Oracle® XML DB Developer's Guide



ORACLE

Oracle XML DB Developer's Guide, 23ai

F46732-05

Copyright © 2002, 2024, Oracle and/or its affiliates.

Primary Author: Maitreyee Chaliha

Contributors: Oracle XML DB development, product management, and quality assurance teams.

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 is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware 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 that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about 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 unless otherwise set forth in an applicable agreement between you and Oracle. 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, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	xliii
Documentation Accessibility	xliii
Diversity and Inclusion	xliv
Related Documents	xliv
Conventions	xlv
Code Examples	xlv
Syntax Descriptions	xlvii

Part I Oracle XML DB Basics

1 Introduction to Oracle XML DB

Overview of Oracle XML DB	1-1
Oracle XML DB Benefits	1-2
Data and Content Unified	1-4
Database Capabilities for Working with XML	1-5
Advantages of Storing Data as XML in the Database	1-6
Data Duality: XML and Relational	1-7
Use XMLType Views If Your Data Is Not XML	1-8
Efficient Storage and Retrieval of Complex XML Documents	1-8
Oracle XML DB Architecture	1-9
Oracle XML DB Features	1-10
XMLType Data Type	1-11
XMLType Storage Models	1-12
XML Schema Support in Oracle XML DB	1-13
DTD Support in Oracle XML DB	1-15
Static Data Dictionary Views Related to XML	1-16
SQL/XML Standard Functions	1-16
Programmatic Access to Oracle XML DB (Java, PL/SQL, and C)	1-17
Oracle XML DB Repository: Overview	1-17
Standards Supported by Oracle XML DB	1-19
Oracle XML DB Technical Support	1-20



Oracle XML DB Examples	1-20
Oracle XML DB Case Studies and Demonstrations on OTN	1-21

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
XML Data Storage	2-3
The Structure of Your XML Data	2-4
Languages Used to Implement Your Application	2-5
XML Processing Options	2-5
Oracle XML DB Repository Access	2-6
Oracle XML DB Cooperates with Other Database Options and Features	2-6

3 Overview of How To Use Oracle XML DB

Creating XMLType Tables and Columns	3-2
Creating XMLType Columns in Shared and Duplicated Tables	3-3
Creating Virtual Columns on XMLType Data Stored as Binary XML	3-4
Partitioning Tables That Contain XMLType Data Stored as Binary XML	3-5
Enforcing XML Data Integrity Using the Database	3-7
Enforcing Referential Integrity Using SQL Constraints	3-8
Loading XML Content into Oracle XML DB	3-12
Loading XML Content Using SQL or PL/SQL	3-12
Loading XML Content Using Java	3-13
Loading XML Content Using C	3-14
Loading Large XML Files that Contain Small XML Documents	3-15
Loading Large XML Files Using SQL*Loader	3-16
Loading XML Documents into the Repository Using DBMS_XDB_REPOS	3-16
Loading Documents into the Repository Using Protocols	3-17
Querying XML Content Stored in Oracle XML DB	3-17
PurchaseOrder XML Document Used in Examples	3-18
Retrieving the Content of an XML Document Using Pseudocolumn OBJECT_VALUE	3-19
Accessing Fragments or Nodes of an XML Document Using XMLQUERY	3-20
Accessing Text Nodes and Attribute Values Using XMLCAST and XMLQUERY	3-21
Searching an XML Document Using XMLEXISTS, XMLCAST, and XMLQUERY	3-22
Performing SQL Operations on XMLType Fragments Using XMLTABLE	3-25
Updating XML Content Stored in Oracle XML DB	3-28
Generating XML Data from Relational Data	3-31
Generating XML Data from Relational Data Using SQL/XML Functions	3-31
Generating XML Data from Relational Data Using DBURITYPE	3-36

Character Sets of XML Documents	3-38
XML Encoding Declaration	3-38
Character-Set Determination When Loading XML Documents into the Database	3-39
Character-Set Determination When Retrieving XML Documents from the Database	3-40
Migrating XMLType Data to Transportable Binary XML (TBX)	3-41

Part II Manipulation of XML Data in Oracle XML DB

4 XQuery and Oracle XML DB

Overview of the XQuery Language	4-1
XPath Expressions Are XQuery Expressions	4-2
XQuery: A Functional Language Based on Sequences	4-3
XQuery Is About Sequences	4-4
XQuery Is Referentially Transparent	4-4
XQuery Update Has Side Effects on Your Data	4-5
XQuery Update Snapshots	4-5
XQuery Full Text Provides Full-Text Search	4-5
XQuery Expressions	4-6
FLWOR Expressions	4-8
Overview of XQuery in Oracle XML DB	4-9
When To Use XQuery	4-10
Predefined XQuery Namespaces and Prefixes	4-10
SQL/XML Functions XMLQUERY, XMLTABLE, XMLExists, and XMLCast	4-11
XMLQUERY SQL/XML Function in Oracle XML DB	4-12
XMLTABLE SQL/XML Function in Oracle XML DB	4-13
Chaining Calls to SQL/XML Function XMLTABLE	4-16
XMLEXISTS SQL/XML Function in Oracle XML DB	4-17
Using XMLExists to Find a Node	4-18
XMLCAST SQL/XML Function in Oracle XML DB	4-19
Using XMLCAST to Extract the Scalar Value of an XML Fragment	4-20
URI Scheme oradb: Querying Table or View Data with XQuery	4-21
Oracle XQuery Extension Functions	4-22
ora:sqrt XQuery Function	4-23
ora:tokenize XQuery Function	4-23
Oracle XQuery Extension-Expression Pragmas	4-23
XQuery Static Type-Checking in Oracle XML DB	4-26
Oracle XML DB Support for XQuery	4-27
Support for XQuery and SQL	4-28
Implementation Choices Specified in the XQuery Standards	4-28
XQuery Features Not Supported by Oracle XML DB	4-29

	XQuery Optional Features	4-29
S	Support for XQuery Functions and Operators	4-29
	XQuery Functions fn:doc, fn:collection, and fn:doc-available	4-30
S	Support for XQuery Full Text	4-30
	XQuery Full Text, XML Schema-Based Data, and Pragma ora:no_schema	4-31
	Restrictions on Using XQuery Full Text with XMLExists	4-31
	Supported XQuery Full Text FTSelection Operators	4-32
	Supported XQuery Full Text Match Options	4-33
	Unsupported XQuery Full Text Features	4-33
	XQuery Full Text Errors	4-33

5 Query and Update of XML Data

Using XQuery with Oracle XML DB	
XQuery Sequences Can Contain Data of Any XQuery Type	5-3
Querying XML Data in Oracle XML DB Repository Using XQuery	5-3
Querying Relational Data Using XQuery and URI Scheme oradb	5-5
Querying XMLType Data Using XQuery	5-10
Using Namespaces with XQuery	5-17
Querying XML Data Using SQL and PL/SQL	5-19
Using the SQL*Plus XQUERY Command	5-24
Using XQuery with PL/SQL, JDBC, and ODP.NET to Access Database Data	5-24
Updating XML Data	5-28
Updating an Entire XML Document	5-28
Replacing XML Nodes	5-29
Updating XML Data to NULL Values	5-35
Inserting Child XML Nodes	5-37
Deleting XML Nodes	5-40
Creating XML Views of Modified XML Data	5-41
Performance Tuning for XQuery	5-42
Rule-Based and Cost-Based XQuery Optimization	5-43
XQuery Optimization over Relational Data	5-44
XQuery Optimization over XML Schema-Based XMLType Data	5-45
Diagnosis of XQuery Optimization: XMLOptimizationCheck	5-47
Performance Improvement for fn:doc and fn:collection on Repository Data	5-48
Use EQUALS_PATH and UNDER_PATH Instead of fn:doc and fn:collection	5-49
Using Oracle XQuery Pragma ora:defaultTable	5-50

6 Indexes for XMLType Data

Oracle XML DB Tasks Involving Indexes	6-1
Overview of Indexing XMLType Data	6-2



XMLIndex	6-3
Advantages of XMLIndex	6-4
XMLIndex Components	6-5
XMLIndex Structured Component	6-6
Ignore the Index Content Tables; They Are Transparent	6-8
Data Type Considerations for XMLIndex Structured Component	6-8
Exchange Partitioning and XMLIndex	6-10
Creating, Dropping, Altering, and Examining an XMLIndex Index	6-12
Use of XMLIndex with a Structured Component	6-14
Using Namespaces and Storage Clauses with an XMLIndex Structured Component	6-15
Adding a Structured Component to an XMLIndex Index	6-16
Using Non-Blocking ALTER INDEX with an XMLIndex Structured Component	6-17
Modifying the Data Type of a Structured XMLIndex Component	6-19
Dropping an XMLIndex Structured Component	6-20
Indexing the Relational Tables of a Structured XMLIndex Component	6-20
How to Tell Whether XMLIndex is Used	6-20
Turning Off Use of XMLIndex	6-22
Guidelines for Using XMLIndex with a Structured Component	6-23
XMLIndex Partitioning and Parallelism	6-23
Collecting Statistics on XMLIndex Objects for the Cost-Based Optimizer	6-25
Data Dictionary Static Public Views Related to XMLIndex	6-25
PARAMETERS Clause for CREATE INDEX and ALTER INDEX	6-27
Using a Registered PARAMETERS Clause for XMLIndex	6-28
PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX	6-28
Usage of XMLIndex_parameters_clause	6-35
Usage of XMLIndex_parameters	6-35
Usage of groups_clause and alter_index_group_clause	6-36
Usage of XMLIndex_xmltable_clause	6-36
Usage of column_clause	6-36
XML Search Index: Indexing for Full Text Search and Ad-hoc Queries	6-36
Creating and Using an XML Search Index	6-37
Maintenance of XML Search Indexes	6-39
Preference Defaults and Recommendations	6-39
Queries using an XML Search Index	6-41
Migrating to Use XML Search Index	6-44
Indexing XMLType Data Stored Object-Relationally	6-46
Indexing Non-Repeating Text Nodes or Attribute Values	6-46
Indexing Repeating (Collection) Elements	6-47

7 Transformation and Validation of XMLType Data

XSL Transformation and Oracle XML DB

7-1

SQL Function XMLTRANSFORM and XMLType Method TRANSFORM()	7-3
XMLTRANSFORM and XMLType.transform(): Examples	7-4
XSL Transformation Using DBUri Servlet	7-9
Validation of XMLType Instances	7-11
Partial and Full XML Schema Validation	7-12
Partial Validation	7-13
Full Validation	7-13
Validating XML Data Stored as XMLType: Examples	7-15

Part III Relational Data To and From XML Data

8 Generation of XML Data from Relational Data

Overview of Generating XML Data	8-1
Generation of XML Data Using SQL Functions	8-2
XMLELEMENT and XMLATTRIBUTES SQL/XML Functions	8-3
Escape of Characters in Generated XML Data	8-6
Formatting of XML Dates and Timestamps	8-7
XMLElement Examples	8-7
XMLFOREST SQL/XML Function	8-11
XMLCONCAT SQL/XML Function	8-12
XMLAGG SQL/XML Function	8-13
XMLPI SQL/XML Function	8-16
XMLCOMMENT SQL/XML Function	8-17
XMLSERIALIZE SQL/XML Function	8-18
XMLPARSE SQL/XML Function	8-20
XMLCOLATTVAL Oracle SQL Function	8-21
XMLCDATA Oracle SQL Function	8-22
Generation of XML Data Using DBMS_XMLGEN	8-23
Using PL/SQL Package DBMS_XMLGEN	8-24
Functions and Procedures of Package DBMS_XMLGEN	8-26
DBMS_XMLGEN Examples	8-31
SYS_XMLAGG Oracle SQL Function	8-48
Ordering Query Results Before Aggregating, Using XMLAGG ORDER BY Clause	8-49
Returning a Rowset Using XMLTABLE	8-50

9 Relational Views over XML Data

Introduction to Creating and Using Relational Views over XML Data	9-1
Creating a Relational View over XML: One Row for Each XML Document	9-2
Creating a Relational View over XML: Mapping XML Nodes to Columns	9-3
Indexing Binary XML Data Exposed Using a Relational View	9-4

10 XMLType Views

What Are XMLType Views?	10-1
CREATE VIEW for XMLType Views: Syntax	10-2
Creating Non-Schema-Based XMLType Views	10-3
Creating XML Schema-Based XMLType Views	10-3
Creating XML Schema-Based XMLType Views Using SQL/XML Publishing Functions	10-4
Using Namespaces with SQL/XML Publishing Functions	10-7
Creating XML Schema-Based XMLType Views Using Object Types or Object Views	10-11
Creating XMLType Employee View, with Nested Department Information	10-12
Creating XMLType Department View, with Nested Employee Information	10-15
Creating XMLType Views from XMLType Tables	10-18
Referencing XMLType View Objects Using SQL Function REF	10-19
Using DML (Data Manipulation Language) on XMLType Views	10-19

Part IV XMLType APIs

11 PL/SQL APIs for XMLType

Overview of PL/SQL APIs for XMLType	11-1
PL/SQL APIs for XMLType: Features	11-2
Lazy Load of XML Data (Lazy Manifestation)	11-2
XMLType Data Type Supports XML Schema	11-2
XMLType Supports Data in Different Character Sets	11-2
PL/SQL APIs for XMLType: References	11-3
PL/SQL DOM API for XMLType (DBMS_XMLDOM)	11-5
Overview of the W3C Document Object Model (DOM) Recommendation	11-6
Oracle XML Developer's Kit Extensions to the W3C DOM Standard	11-7
Supported W3C DOM Recommendations	11-7
Difference Between DOM and SAX	11-8
PL/SQL DOM API for XMLType (DBMS_XMLDOM): Features	11-8
PL/SQL DOM API Support for XML Schema	11-9
Enhanced DOM Performance	11-9
Application Design Using Oracle XML Developer's Kit and Oracle XML DB	11-9
Preparing XML Data to Use the PL/SQL DOM API for XMLType	11-10
XML Schema Types Are Mapped to SQL Object Types	11-11
DOM Fidelity for XML Schema Mapping	11-12
Wrap Existing Data as XML with XMLType Views	11-12
DBMS_XMLDOM Methods Supported by Oracle XML DB	11-12
PL/SQL DOM API for XMLType: Node Types	11-13

PL/SQL Function NEWDOMDOCUMENT and DOMDOCUMENT Nodes	11-14
DOM NodeList and NamedNodeMap Objects	11-15
Overview of Using the PL/SQL DOM API for XMLType (DBMS_XMLDOM)	11-15
PL/SQL DOM API for XMLType – Examples	11-16
Large Node Handling Using DBMS_XMLDOM	11-18
Get-Push Model for Large Node Handling	11-20
Get-Pull Model for Large Node Handling	11-22
Set-Pull Model for Large Node Handling	11-23
Set-Push Model for Large Node Handling	11-24
Determining Binary Stream or Character Stream for Large Node Handling	11-25
PL/SQL Parser API for XMLType (DBMS_XMLPARSER)	11-26
PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)	11-27
PL/SQL XSLT Processor for XMLType: Features	11-28
Using the PL/SQL XSLT Processor API for XMLType (DBMS_XSLPROCESSOR)	11-29

12 PL/SQL Package DBMS_XMLSTORE

Using Package DBMS_XMLSTORE	12-1
Inserting an XML Document Using DBMS_XMLSTORE	12-2
Updating XML Data Using DBMS_XMLSTORE	12-4
Deleting XML Data Using DBMS_XMLSTORE	12-5

13 Java DOM API for XMLType

Overview of Java DOM API for XMLType	13-2
Access to XMLType Data Using JDBC	13-2
Using JDBC to Access XML Documents in Oracle XML DB	13-3
Manipulating XML Database Documents Using JDBC	13-5
Loading a Large XML Document into the Database Using JDBC	13-9
MS Windows Java Security Manager Permissions for Java DOM API with a Thick	
Connection	13-11
Creating XML Schema-Based Documents	13-11
XMLType Instance Representation in Java (JDBC or SQLJ)	13-12
Classes of Java DOM API for XMLType	13-12
Using the Java DOM API for XMLType	13-14
Large XML Node Handling with Java	13-14
Stream Extensions to Java DOM	13-15
Get-Pull Model	13-16
Get-Push Model	13-17
Set-Pull Model	13-17
Set-Push Model	13-18



14 C DOM API for XMLType

Overview of the C DOM API for XMLType	14-1
Access to XMLType Data Stored in the Database Using OCI	14-2
Creating XMLType Instances on the Client	14-3
XML Context Parameter for C DOM API Functions	14-3
OCIXmIDbInitXmICtx() Syntax	14-3
OCIXmIDbFreeXmlCtx() Syntax	14-4
Initializing and Terminating an XML Context	14-4
Using the C API for XML with Binary XML	14-8
Using the Oracle XML Developer's Kit Pull Parser with Oracle XML DB	14-11
Common XMLType Operations in C	14-16

15 Oracle XML DB and Oracle Data Provider for .NET

Oracle XML DB and ODP.NET XML	15-1
Using XMLType Data with ODP.NET	15-1

Part V XML Schema and Object-Relational XMLType

16 Choice of XMLType Storage and Indexing

Introduction to Choosing an XMLType Storage Model and Indexing Approaches	16-1
XMLType Use Case Spectrum: Data-Centric to Document-Centric	16-3
Common Use Cases for XML Data Stored as XMLType	16-4
XMLType Use Case: No XML Fragment Updating or Querying	16-5
XMLType Use Case: Data Integration from Diverse Sources with Different XML Schemas	16-5
XMLType Use Case: Staged XML Data for ETL	16-6
XMLType Use Case: Semi-Structured XML Data	16-6
XMLType Use Case: Business Intelligence Queries	16-7
XMLType Use Case: XML Queries Involving Full-Text Search	16-7
XMLType Storage Model Considerations	16-8
XMLType Indexing Considerations	16-9
XMLType Storage Options: Relative Advantages	16-9

17 XML Schema Storage and Query: Basic

Overview of XML Schema	17-2
XML Schema for Schemas	17-3
XML Schema Features	17-3

XML Instance Documents	17-3
XML Namespaces and XML Schemas	17-3
Overview of Editing XML Schemas	17-3
Overview of Using XML Schema with Oracle XML DB	17-4
Why Use XML Schema with Oracle XML DB?	17-5
Overview of Annotating an XML Schema to Control Naming, Mapping, and Storage	17-6
DOM Fidelity	17-7
XMLType Methods Related to XML Schema	17-8
XML Schema Registration with Oracle XML DB	17-8
XML Schema Registration Actions	17-10
Registering an XML Schema with Oracle XML DB	17-10
SQL Types and Tables Created During XML Schema Registration	17-12
Default Tables for Global Elements	17-13
Database Objects That Depend on Registered XML Schemas	17-13
Local and Global XML Schemas	17-14
Local XML Schema	17-14
Global XML Schema	17-15
Fully Qualified XML Schema URLs	17-16
Deletion of an XML Schema	17-17
Listing All Registered XML Schemas	17-18
Creation of XMLType Tables and Columns Based on XML Schemas	17-19
Specification of XMLType Storage Options for XML Schema-Based Data	17-22
Binary XML Storage of XML Schema-Based Data	17-22
Object-Relational Storage of XML Schema-Based Data	17-25
Ways to Identify XML Schema Instance Documents	17-27
Attributes noNamespaceSchemaLocation and schemaLocation	17-28
XML Schema and Multiple Namespaces	17-28
XML Schema Data Types Are Mapped to Oracle XML DB Storage	17-29

18 XML Schema Storage and Query: Object-Relational Storage

Object-Relational Storage of XML Documents	18-2
How Collections Are Stored for Object-Relational XMLType Storage	18-3
SQL Types Created during XML Schema Registration for Object-Relational Storage	18-4
Default Tables Created during XML Schema Registration	18-5
Do Not Use Internal Constructs Generated during XML Schema Registration	18-6
Generated Names are Case Sensitive	18-6
SYS_XDBPD\$ and DOM Fidelity for Object-Relational Storage	18-6
Oracle XML Schema Annotations	18-7
Common Uses of XML Schema Annotations	18-8
XML Schema Annotation Example	18-9
Annotating an XML Schema Using DBMS_XMLSCHEMA_ANNOTATE	18-13



Available Oracle XML DB XML Schema Annotations	18-14
XML Schema Annotation Guidelines for Object-Relational Storage	18-16
Avoid Creation of Unnecessary Tables for Unused Top-Level Elements	18-17
Provide Your Own Names for Default Tables	18-18
Turn Off DOM Fidelity If Not Needed	18-18
Annotate Time-Related Elements with a Timestamp Data Type	18-19
Add Table and Column Properties	18-19
Store Large Collections Out of Line	18-19
Querying a Registered XML Schema to Obtain Annotations	18-20
You Can Apply Annotations from One XML Schema to Another	18-21
Use DBMS_XMLSCHEMA to Map XML Schema Data Types to SQL Data Types	18-21
Example of Mapping XML Schema Data Types to SQL	18-22
XML Schema Attribute Data Types Mapped to SQL	18-23
You Can Override the SQLType Value in an XML Schema When Declaring Attributes	18-24
XML Schema Element Data Types Mapped to SQL	18-24
Override of the SQLType Value in an XML Schema When Declaring Elements	18-25
How XML Schema simpleType Is Mapped to SQL	18-25
NCHAR, NVARCHAR2, and NCLOB SQLType Values Are Not Supported for	
SQLType	18-28
simpleType: How XML Strings Are Mapped to SQL VARCHAR2 Versus CLOB	18-29
How XML Schema Time Zones Are Mapped to SQL	18-29
How XML Schema complexType Is Mapped to SQL	18-30
Attribute Specification in a complexType XML Schema Declaration	18-30
complexType Extensions and Restrictions in Oracle XML DB	18-31
complexType Declarations in XML Schema: Handling Inheritance	18-31
How a complexType Based on simpleContent Is Mapped to an Object Type	18-34
How any and anyAttribute Declarations Are Mapped to Object Type Attributes	18-34
Creating XML Schema-Based XMLType Columns and Tables	18-35
Overview of Partitioning XMLType Tables and Columns Stored Object-Relationally	18-37
Examples of Partitioning XMLType Data Stored Object-Relationally	18-38
Partition Maintenance for XMLType Data Stored Object-Relationally	18-39
Specification of Relational Constraints on XMLType Tables and Columns	18-40
Adding Unique Constraints to the Parent Element of an Attribute	18-42
Out-Of-Line Storage of XMLType Data	18-43
Setting Annotation Attribute xdb:SQLInline to false for Out-Of-Line Storage	18-44
Storing Collections in Out-Of-Line Tables	18-47
Considerations for Working with Complex or Large XML Schemas	18-49
Circular and Cyclical Dependencies Among XML Schemas	18-50
For Circular XML Schema Dependencies Set Parameter GENTABLES to TRUE	18-51
complexType Declarations in XML Schema: Handling Cycles	18-51
Cyclical References Among XML Schemas	18-54
Support for Recursive Schemas	18-56



defaultTable Shared Among Common Out-Of-Line Elements	18-57
Query Rewrite when DOCID is Present	18-59
DOCID Column Creation Disabling	18-60
XML Fragments Can Be Mapped to Large Objects (LOBs)	18-60
ORA-01792 and ORA-04031: Issues with Large XML Schemas	18-61
Considerations for Loading and Retrieving Large Documents with Collections	18-62
Guidelines for Configuration Parameters xdbcore-loadableunit-size and xdbcore-	
xobmem-bound	18-64
Debugging XML Schema Registration for XML Data Stored Object-Relationally	18-64

19 XPath Rewrite for Object-Relational Storage

Overview of XPath Rewrite for Object-Relational Storage	19-1
Common XPath Expressions that Are Rewritten	19-3
XPath Rewrite for Out-Of-Line Tables	19-4
Guidelines for Using Execution Plans to Analyze and Optimize XPath Queries	19-5
Guideline: Look for underlying tables versus XML functions in execution plans	19-6
Guideline: Name the object-relational tables, so you recognize them in execution plans	19-6
Guideline: Create an index on a column targeted by a predicate	19-7
Guideline: Create indexes on ordered collection tables	19-9
Guideline: Use XMLOptimizationCheck to determine why a query is not rewritten	19-10

20 XML Schema Evolution

Overview of XML Schema Evolution	20-1
Copy-Based Schema Evolution	20-2
Scenario for Copy-Based Evolution	20-3
COPYEVOLVE Parameters and Errors	20-6
Limitations of Procedure COPYEVOLVE	20-8
Guidelines for Using Procedure COPYEVOLVE	20-8
Top-Level Element Name Changes	20-9
User-Created Virtual Columns of Tables Other Than Default Tables	20-9
Ensure That the XML Schema and Dependents Are Not Used by Concurrent Sessions	20-10
Rollback When Procedure DBMS_XMLSCHEMA.COPYEVOLVE Raises an Error	20-10
– Failed Rollback From Insufficient Privileges	20-10
Privileges Needed for XML Schema Evolution	20-10
Update of Existing XML Instance Documents Using an XSLT Stylesheet	20-11
Examples of Using Procedure COPYEVOLVE	20-13
In-Place XML Schema Evolution	20-16
Restrictions for In-Place XML Schema Evolution	20-17
Backward-Compatibility Restrictions	20-17
Other Restrictions on In-Place Evolution	20-19



Supported Operations for In-Place XML Schema Evolution	20-19
Guidelines for Using In-Place XML Schema Evolution	20-21
inPlaceEvolve Parameters	20-22
The diffXML Parameter Document	20-23
diffXML Operations and Examples	20-24

Part VI Oracle XML DB Repository

21 Access to Oracle XML DB Repository Data

Overview of Oracle XML DB Repository	21-2
Oracle XML DB Provides Name-Level Locking	21-4
Two Ways to Access Oracle XML DB Repository Resources	21-5
Database Schema (User Account) XDB and Oracle XML DB Repository	21-6
Repository Terminology and Supplied Resources	21-6
Repository Terminology	21-7
Predefined Repository Files and Folders	21-8
Oracle XML DB Repository Resources	21-9
Where Is Repository Data Stored?	21-9
Names of Generated Tables	21-10
How Object-Relational Storage Is Defined for Repository Resources	21-10
Oracle ASM Virtual Folder	21-10
How Documents are Stored in Oracle XML DB Repository	21-11
Repository Data Access Control	21-12
Repository Path-Name Resolution	21-12
Link Types	21-13
Repository Links and Document Links	21-13
Hard Links and Weak Links	21-14
Creating a Weak Link with No Knowledge of Folder Hierarchy	21-15
How and When to Prevent Multiple Hard Links	21-16
Navigational or Path Access to Repository Resources	21-16
Access to Oracle XML DB Resources Using Internet Protocols	21-19
Where You Can Use Oracle XML DB Protocol Access	21-19
Overview of Protocol Access to Oracle XML DB	21-19
Retrieval of Oracle XML DB Resources	21-20
Storage of Oracle XML DB Resources	21-20
Internet Protocols and XMLType: XMLType Direct Stream Write	21-20
Access to Oracle ASM Files Using Protocols and Resource APIs – For DBAs	21-21
Query-Based Access to Repository Resources	21-23
Servlet Access to Repository Resources	21-24
Operations on Repository Resources	21-24



Accessing the Content of Repository Resources Using SQL	21-31
Access to the Content of XML Schema-Based Documents	21-32
Accessing Resource Content Using Element XMLRef in Joins	21-32
Update of the Content of Repository Documents	21-33
Update of Repository Content Using Internet Protocols	21-34
Update of Repository Content Using SQL	21-35
Updating a Document in the Repository by Updating Its Resource Document	21-35
Updating an XML Schema-Based Document in the Repository by Updating the	
Default Table	21-37
Querying Resources in RESOURCE_VIEW and PATH_VIEW	21-38
Oracle XML DB Hierarchical Repository Index	21-41

22 Configuration of Oracle XML DB Repository

Resource Configuration Files	22-2
Configuring a Resource	22-3
Common Configuration Parameters	22-4
Configuration Element ResConfig	22-4
Configuration Elements defaultChildConfig and configuration	22-5
Configuration Element applicationData	22-6

23 Use of XLink and XInclude with Oracle XML DB

Overview of XLink and XInclude	23-2
Link Types for XLink and XInclude	23-3
XLink and XInclude Links Model Document Relationships	23-3
XLink Link Types and XInclude Link Types	23-3
XInclude: Compound Documents	23-4
Oracle XML DB Support for XLink	23-5
Oracle XML DB Support for XInclude	23-6
Expanding Compound-Document Inclusions	23-7
Validation of Compound Documents	23-8
Update of a Compound Document	23-8
Compound Document Versioning, Locking, and Access Control	23-9
Use View DOCUMENT_LINKS to Examine XLink and XInclude Links	23-9
Querying DOCUMENT_LINKS for XLink Information	23-10
Querying DOCUMENT_LINKS for XInclude Information	23-11
Configuration of Repository Resources for XLink and XInclude	23-12
Configure the Treatment of Unresolved Links: Attribute UnresolvedLink	23-13
Configure the Type of Document Links to Create: Element LinkType	23-14
Configure the Path Format for Retrieval: Element PathFormat	23-14
Configure Conflict-Resolution for XInclude: Element ConflictRule	23-15



Configure the Decomposition of Documents Using XInclude: Element SectionConfig	23-15
XLink and XInclude Configuration Examples	23-16
Manage XLink and XInclude Links Using DBMS_XDB_REPOS.processLinks	23-18

24 Repository Access Using RESOURCE_VIEW and PATH_VIEW

Overview of Oracle XML DB RESOURCE_VIEW and PATH_VIEW	24-2
RESOURCE_VIEW Definition and Structure	24-4
PATH_VIEW Definition and Structure	24-5
The Difference Between RESOURCE_VIEW and PATH_VIEW	24-5
Operations You Can Perform Using UNDER_PATH and EQUALS_PATH	24-6
Oracle SQL Functions That Use RESOURCE_VIEW and PATH_VIEW	24-6
UNDER_PATH SQL Function	24-7
EQUALS_PATH SQL Function	24-8
PATH SQL Function	24-9
DEPTH SQL Function	24-9
Accessing Repository Data Paths, Resources and Links: Examples	24-10
Deleting Repository Resources: Examples	24-17
Deleting Nonempty Folder Resources	24-17
Updating Repository Resources: Examples	24-18
Working with Multiple Oracle XML DB Resources	24-21
Performance Guidelines for Oracle XML DB Repository Operations	24-22
Searching for Resources Using Oracle Text	24-23

25 Resource Versions

Overview of Oracle XML DB Repository Resource Versioning	25-1
Overview of PL/SQL Package DBMS_XDB_VERSION	25-3
Resource Versions and Resource IDs	25-4
Resource Versions and ACLs	25-5
Resource Versioning Examples	25-6

26 PL/SQL Access to Oracle XML DB Repository

DBMS_XDB_REPOS: Access and Manage Repository Resources	26-1
DBMS_XDB_REPOS: ACL-Based Security Management	26-3
DBMS_XDB_CONFIG: Configuration Management	26-7

27 Repository Access Control

Access Control Concepts	27-2
Authentication and Authorization	27-3



Principal: A User or Role	27-3
Database Roles and ACLs Map Privileges to Users	27-4
Principal DAV::owner	27-4
Privilege: A Permission	27-4
Access Control Entry (ACE)	27-5
Access Control List (ACL)	27-6
Database Privileges for Repository Operations	27-6
Privileges	27-7
Atomic Privileges	27-7
Aggregate Privileges	27-8
ACLs and ACEs	27-9
System ACLs	27-10
ACL and ACE Evaluation	27-11
ACL Validation	27-11
Element invert: Complement the Principals in an ACE	27-11
Overview of Working with Access Control Lists (ACLs)	27-12
Creating an ACL Using DBMS_XDB_REPOS.CREATERESOURCE	27-13
Retrieving an ACL Document, Given its Repository Path	27-14
Setting the ACL of a Resource	27-14
Deleting an ACL	27-15
Updating an ACL	27-15
Retrieving the ACL Document that Protects a Given Resource	27-17
Retrieving Privileges Granted to the Current User for a Particular Resource	27-17
Checking Whether the Current User Has Privileges on a Resource	27-18
Checking Whether a User Has Privileges Using the ACL and Resource Owner	27-19
Retrieving the Path of the ACL that Protects a Given Resource	27-19
Retrieving the Paths of All Resources Protected by a Given ACL	27-20
ACL Caching	27-21
Repository Resources and Database Table Security	27-21
Optimization: Do not enforce ACL-based security if you do not need it	27-22
Integration Of Oracle XML DB with LDAP	27-23

28 Repository Access Using Protocols

Overview of Oracle XML DB Protocol Server	28-1
Session Pooling	28-2
Oracle XML DB Protocol Server Configuration Management	28-3
Protocol Server Configuration Parameters	28-4
Configuring Secure HTTP (HTTPS)	28-8
Enabling the HTTP Listener to Use SSL	28-9
Enabling TCPS Dispatcher	28-10
Using Listener Status to Check Port Configuration	28-10

ORACLE

Configuring Protocol Port Parameters after Database Consolidation	28-11
Configuration and Management of Authentication Mechanisms for HTTP	28-11
Nonces for Digest Authentication	28-12
Oracle XML DB Repository and File-System Resources	28-13
Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents	28-13
Event-Based Logging	28-13
Auditing of HTTP and FTP Protocols	28-13
FTP and the Oracle XML DB Protocol Server	28-14
Oracle XML DB Protocol Server: FTP Features	28-14
FTP Features That Are Not Supported	28-15
Supported FTP Client Methods	28-16
FTP Quote Methods	28-16
Uploading Content to Oracle XML DB Repository Using FTP	28-17
Using FTP with Oracle ASM Files	28-19
Using FTP on the Standard Port Instead of the Oracle XML DB Default Port	28-21
Using IPv6 IP Addresses with FTP	28-22
FTP Server Session Management	28-22
Handling Error 421. Modifying the Default Timeout Value of an FTP Session	28-22
FTP Client Failure in Passive Mode	28-23
HTTP(S) and Oracle XML DB Protocol Server	28-23
Oracle XML DB Protocol Server: HTTP(S) Features	28-23
Supported HTTP(S) Client Methods	28-25
Using HTTP(S) on a Standard Port Instead of an Oracle XML DB Default Port	28-25
Use of IPv6 IP Addresses with HTTP(S)	28-26
HTTPS: Support for Secure HTTP	28-26
Control of URL Expiration Time	28-27
Anonymous Access to Oracle XML DB Repository Using HTTP	28-27
Use of Java Servlets with HTTP(S)	28-28
Embedded PL/SQL Gateway	28-28
Transmission of Multibyte Data From a Client	28-29
Characters That Are Not ASCII in URLs	28-30
Character Sets for HTTP(S)	28-30
WebDAV and Oracle XML DB	28-31
Oracle XML DB WebDAV Features	28-31
WebDAV Features That Are Not Supported by Oracle XML DB	28-32
WebDAV Client Methods Supported by Oracle XML DB	28-32
WebDAV and Microsoft Windows	28-33
Creating a WebFolder in Microsoft Windows For Use With Oracle XML DB Repository	28-34
Use of WebDAV with Windows Explorer to Copy Files into Oracle XML DB	00.07
Repository	28-34



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	29-3
Scenario: Metadata for a Photo Collection	29-3
Using XML Schemas to Define Resource Metadata	29-4
Addition, Modification, and Deletion of Resource Metadata	29-5
Adding Metadata Using APPENDRESOURCEMETADATA	29-6
Deleting Metadata Using DELETERESOURCEMETADATA	29-8
Adding Metadata Using SQL DML	29-8
Adding Metadata Using WebDAV PROPPATCH	29-9
Querying XML Schema-Based Resource Metadata	29-10
XML Image Metadata from Binary Image Metadata	29-12
Adding Non-Schema-Based Resource Metadata	29-12
PL/SQL Procedures Affecting Resource Metadata	29-14

30 Oracle XML DB Repository Events

Overview of Repository Events	30-2
Repository Events: Use Cases	30-2
Repository Events and Database Triggers	30-3
Repository Event Listeners and Event Handlers	30-3
Repository Event Configuration	30-3
Possible Repository Events	30-4
Repository Operations and Events	30-6
Repository Event Handler Considerations	30-8
Configuration of Repository Events	30-9
Configuration Element event-listeners	30-10
Configuration Element listener	30-11
Repository Events Configuration Examples	30-12

31 Guidelines for Oracle XML DB Applications in Java

Overview of Oracle XML DB Java Applications	31-2
HTTP(S): Access Java Servlets or Directly Access XMLType Resources	31-2
Use JDBC XMLType Support to Access Many XMLType Object Elements	31-3
Use Servlets to Manipulate and Write Out Data Quickly as XML	31-3
Oracle XML DB Java Servlet Support Restrictions	31-3
Configuration of Oracle XML DB Servlets	31-3
HTTP Request Processing for Oracle XML DB Servlets	31-7
Session Pool and Oracle XML DB Servlets	31-8

ORACLE

Native XML Stream Support	31-8
Oracle XML DB Servlet APIs	31-8
Oracle XML DB Servlet Example	31-9

32 Data Access Using URIs

Overview of Oracle XML DB URI Features	32-2
URIs and URLs	32-2
URIType and its Subtypes	32-3
Overview of DBUris and XDBUris	32-4
URIType PL/SQL Methods	32-5
HTTPURIType PL/SQL Method GETCONTENTTYPE()	32-6
DBURIType PL/SQL Method GETCONTENTTYPE()	32-7
DBURIType PL/SQL Method GETCLOB()	32-7
DBURIType PL/SQL Method GETBLOB()	32-8
Accessing Data Using URIType Instances	32-8
XDBUris: Pointers to Repository Resources	32-11
XDBUri URI Syntax	32-11
Using XDBUri: Examples	32-12
DBUris: Pointers to Database Data	32-14
View the Database as XML Data	32-15
DBUri URI Syntax	32-16
DBUris are Scoped to a Database and Session	32-18
Using DBUris —Examples	32-18
Targeting a Table Using a DBUri	32-19
Targeting a Row in a Table Using a DBUri	32-20
Targeting a Column Using a DBUri	32-20
Retrieving the Text Value of a Column Using a DBUri	32-22
Targeting a Collection Using a DBUri	32-22
Create New Subtypes of URIType Using Package URIFACTORY	32-23
Registering New URIType Subtypes with Package URIFACTORY	32-24
SYS_DBURIGEN SQL Function	32-26
Rules for Passing Columns or Object Attributes to SYS_DBURIGEN	32-27
Using SQL Function SYS_DBURIGEN: Examples	32-27
Inserting Database References Using SYSDBURIGEN	32-28
Returning Partial Results Using SYSDBURIGEN	32-28
Returning URLs to Inserted Objects Using SYS_DBURIGEN	32-29
DBUriServlet	32-30
Overriding the MIME Type Using a URL	32-32
Customizing DBUriServlet	32-32
Using Roles for DBUriServlet Security	32-33
Configuring Package URIFACTORY to Handle DBUris	32-34

33 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	33-3
Enabling Web Services for a Specific User	33-4
Query Oracle XML DB Using a Web Service	33-4
Access to PL/SQL Stored Procedures Using a Web Service	33-6
Using a PL/SQL Function with a Web Service: Example	33-8

Part VII Oracle Tools that Support Oracle XML DB

34 Administration of Oracle XML DB

Upgrade or Downgrade of an Existing Oracle XML DB Installation	34-1
Authentication Considerations for Database Installation, Upgrade and Downgrade	34-2
Authentication Considerations for a Database Installation	34-3
Authentication Considerations for a Database Upgrade	34-3
Authentication Considerations for a Database Downgrade	34-3
Automatic Installation of Oracle XML DB	34-4
Validation of ACL Documents and Configuration File	34-4
Administration of Oracle XML DB Using Oracle Enterprise Manager	34-5
Configuration of Oracle XML DB Using xdbconfig.xml	34-5
Oracle XML DB Configuration File, xdbconfig.xml	34-6
Element xdbconfig (Top-Level)	34-7
Element sysconfig (Child of xdbconfig)	34-7
Element userconfig (Child of xdbconfig)	34-7
Element protocolconfig (Child of sysconfig)	34-8
Element httpconfig (Child of protocolconfig)	34-8
Element servlet (Descendant of httpconfig)	34-9
Oracle XML DB Configuration File Example	34-10
Oracle XML DB Configuration API	34-12
Configuration of Mappings from Default Namespace to Schema Location	34-13
Configuration of XML File Extensions	34-15
Oracle XML DB and Database Consolidation	34-15
Package DBMS_XDB_ADMIN	34-16

35 How to Load XML Data

Load XMLType Data Using SQL*Loader Load XMLType LOB Data Using SQL*Loader Load LOB Data Using Predetermined Size Fields Load LOB Data Using Delimited Fields	35-1
Load LOB Data Using Predetermined Size Fields Load LOB Data Using Delimited Fields	35-2
Load LOB Data Using Delimited Fields	35-2
	35-3
	35-3
Load XML Columns Containing LOB Data from LOBFILEs	35-3
Specify LOBFILEs	35-4
Load XMLType Data Directly from a Control File Using SQL*Loader	35-4
Loading Large XML Documents Using SQL*Loader	35-4

36 Export and Import of Oracle XML DB Data

Overview of Exporting and Importing XMLType Tables	36-1
Export/Import Limitations for Oracle XML DB Repository	36-3
Export/Import Syntax and Examples	36-3
Performing a Table-Mode Export /Import	36-3
Performing a Schema-Mode Export/Import	36-4

37 XML Data Exchange Using Oracle Database Advanced Queuing

XML and Oracle Database Advanced Queuing	37-1
Oracle Database Advanced Queuing and XML Message Payloads	37-2
Advantages of Using Oracle Database Advanced Queuing	37-3
Oracle Database Advanced Queuing	37-3
Message Queuing	37-4
XMLType Attributes in Object Types	37-4
Internet Data Access Presentation (iDAP): SOAP for AQ	37-4
iDAP Architecture	37-5
XMLType Queue Payloads	37-6
Guidelines for Using XML and Oracle Database Advanced Queuing	37-8
Store AQ XML Messages with Many PDFs as One Record	37-9
Add New Recipients After Messages Are Enqueued	37-9
Enqueue and Dequeue XML Messages	37-9
Parse Messages with XML Content from AQ Queues	37-9
Prevent the Listener from Stopping Until an XML Document Is Processed	37-10
HTTPS with AQ	37-10
Store XML in Oracle AQ Message Payloads	37-10
iDAP and SOAP	37-10



Part VIII Appendixes

A Oracle-Supplied XML Schemas and Examples

XDBResource.xsd: XML Schema for Oracle XML DB Resources	A-1
XDBResConfig.xsd: XML Schema for Resource Configuration	A-11
acl.xsd: XML Schema for ACLs	A-15
xdbconfig.xsd: XML Schema for Configuring Oracle XML DB	A-18
xdiff.xsd: XML Schema for Comparing Schemas for In-Place Evolution	A-32
Purchase-Order XML Schemas	A-34
XSLT Stylesheet Example, PurchaseOrder.xsl	A-44
Loading XML Data Using C (OCI)	A-50
Initializing and Terminating an XML Context (OCI)	A-54

B Oracle XML DB Restrictions

C XMLIndex Unstructured Component

Guidelines for Using XMLIndex with an Unstructured Component	C-5
Ignore the Path Table – It Is Transparent	C-7
Column VALUE of an XMLIndex Path Table	C-7
Secondary Indexes on Column VALUE	C-9
XPath Expressions That Are Not Indexed by an XMLIndex Unstructured Component	C-9
Using XMLIndex with an Unstructured Component	C-9
Creating Additional Secondary Indexes on an XMLIndex Path Table	C-11
Asynchronous (Deferred) Maintenance of XMLIndex Indexes	C-16
Syncing an XMLIndex Index in Case of Error ORA-08181	C-18
Advantages of Unstructured XMLIndex	C-18
XMLIndex Path Subsetting: Specifying the Paths You Want to Index	C-19
Examples of XMLIndex Path Subsetting	C-20
XMLIndex Path-Subsetting Rules	C-21
PARAMETERS Clause for CREATE INDEX and ALTER INDEX in Unstructured Index	C-21
Usage of PATHS Clause	C-21
Usage of create_index_paths_clause and alter_index_paths_clause	C-22
Usage of pikey_clause, path_id_clause, and order_key_clause	C-22
Usage of value_clause	C-22
Usage of async_clause	C-22

D Indexing XML Data for Full-Text Queries (pre-23ai)

Creating and Using an XML Search Index

D-2

What To Do If an XML Search Index Is Not Picked Up	D-4
Pragma ora:no_schema: Using XML Schema-Based Data with XQuery Full Text	D-4
Pragma ora:use_xmltext_idx: Forcing the Use of an XML Search Index	D-5
Migrating from Using Oracle Text Index to XML Search Index	D-6

Index



List of Examples

3-1	Creating a Table with an XMLType Column	3-2
3-2	Creating a Table of XMLType	3-3
3-3	Creating a Sharded Table in a System-Sharded Environment	3-4
3-4	Creating a Duplicated Table in a Sharded Environment	3-4
3-5	Creating a Virtual Column for an XML Attribute in an XMLType Table	3-5
3-6	Creating a Virtual Column for an XML Attribute in an XMLType Column	3-5
3-7	Partitioning a Relational Table That Has an XMLType Column	3-6
3-8	Partitioning an XMLType Table	3-7
3-9	Error From Attempting to Insert an Incorrect XML Document	3-8
3-10	Constraining a Binary XML Table Using a Virtual Column	3-9
3-11	Constraining a Binary XML Column Using a Virtual Column: Uniqueness	3-10
3-12	Constraining a Binary XML Column Using a Virtual Column: Foreign Key	3-10
3-13	Enforcing Database Integrity When Loading XML Using FTP	3-11
3-14	Creating a Database Directory	3-13
3-15	Inserting XML Content into an XMLType Table	3-13
3-16	Inserting Content into an XMLType Table Using Java	3-14
3-17	Inserting Content into an XMLType Table Using C	3-14
3-18	Inserting XML Content into the Repository Using CREATERESOURCE	3-16
3-19	PurchaseOrder XML Instance Document	3-18
3-20	Retrieving an Entire XML Document Using OBJECT_VALUE	3-19
3-21	Accessing XML Fragments Using XMLQUERY	3-20
3-22	Accessing a Text Node Value Using XMLCAST and XMLQuery	3-21
3-23	Searching XML Content Using XMLExists, XMLCast, and XMLQuery	3-23
3-24	Joining Data from an XMLType Table and a Relational Table	3-25
3-25	Accessing Description Nodes Using XMLTABLE	3-26
3-26	Counting the Number of Elements in a Collection Using XMLTABLE	3-27
3-27	Counting the Number of Child Elements in an Element Using XMLTABLE	3-28
3-28	Updating a Text Node	3-28
3-29	Replacing an Entire Element Using XQuery Update	3-29
3-30	Changing Text Node Values Using XQuery Update	3-30
3-31	Generating XML Data Using SQL/XML Functions	3-32
3-32	Creating XMLType Views Over Conventional Relational Tables	3-34
3-33	Querying XMLType Views	3-34
3-34	Generating XML Data from a Relational Table Using DBURIType and getXML()	3-37
3-35	Restricting Rows Using an XPath Predicate	3-37
3-36	Restricting Rows and Columns Using an XPath Predicate	3-37

3-37	Creating a CSX Based Table p with the Following Specifications	3-41
3-38	Migrating Table p to TBX Based Table int p Using Online Redefinition	3-42
4-1	Chaining XMLTable Calls	4-17
4-2	Finding a Node Using SQL/XML Function XMLExists	4-18
4-3	Extracting the Scalar Value of an XML Fragment Using XMLCAST	4-20
4-4	Static Type-Checking of XQuery Expressions: oradb URI scheme	4-27
4-5	Static Type-Checking of XQuery Expressions: XML Schema-Based Data	4-27
5-1	Creating Resources for Examples	5-2
5-2	XMLQuery Applied to a Sequence of Items of Different Types	5-3
5-3	FLOWR Expression Using for, let, order by, where, and return	5-4
5-4	FLOWR Expression Using Built-In Functions	5-5
5-5	Querying Relational Data as XML Using XMLQuery	5-7
5-6	Querying Relational Data as XML Using a Nested FLWOR Expression	5-7
5-7	Querying Relational Data as XML Using XMLTable	5-9
5-8	Querying an XMLType Column Using XMLQuery PASSING Clause	5-12
5-9	Using XMLTABLE with XML Schema-Based Data	5-13
5-10	Using XMLQUERY with XML Schema-Based Data	5-14
5-11	Using XMLTABLE with PASSING and COLUMNS Clauses	5-14
5-12	Using XMLTABLE with RETURNING SEQUENCE BY REF	5-15
5-13	Using Chained XMLTABLE with Access by Reference	5-16
5-14	Using XMLTABLE to Decompose XML Collection Elements into Relational Data	5-16
5-15	Using XMLQUERY with a Namespace Declaration	5-17
5-16	Using XMLTABLE with the XMLNAMESPACES Clause	5-18
5-17	Querying XMLTYPE Data	5-19
5-18	Querying Transient XMLTYPE Data Using a PL/SQL Cursor	5-20
5-19	Extracting XML Data and Inserting It into a Relational Table Using SQL	5-21
5-20	Extracting XML Data and Inserting It into a Table Using PL/SQL	5-22
5-21	Searching XML Data Using SQL/XML Functions	5-23
5-22	Extracting Fragments Using XMLQUERY	5-23
5-23	Using the SQL*Plus XQUERY Command	5-24
5-24	Using XQuery with PL/SQL	5-25
5-25	Using XQuery with JDBC	5-26
5-26	Using XQuery with ODP.NET and C#	5-26
5-27	Updating XMLType Data Using SQL UPDATE	5-29
5-28	Updating XMLTYPE Data Using SQL UPDATE and XQuery Update	5-30
5-29	Updating Multiple Text Nodes and Attribute Nodes	5-31
5-30	Updating Selected Nodes within a Collection	5-32

5-31	Incorrectly Updating a Node That Occurs Multiple Times in a Collection	5-33
5-32	Correctly Updating a Node That Occurs Multiple Times in a Collection	5-34
5-33	NULL Updates – Element and Attribute	5-36
5-34	NULL Updates – Text Node	5-37
5-35	Inserting an Element into a Collection	5-38
5-36	Inserting an Element that Uses a Namespace	5-39
5-37	Inserting an Element Before an Element	5-39
5-38	Inserting an Element as the Last Child Element	5-40
5-39	Deleting an Element	5-40
5-40	Creating a View Using Updated XML Data	5-41
5-41	Optimization of XMLQuery over Relational Data	5-44
5-42	Optimization of XMLTable over Relational Data	5-44
5-43	Optimization of XMLQuery with Schema-Based XMLType Data	5-45
5-44	Optimization of XMLTable with Schema-Based XMLType Data	5-46
5-45	Unoptimized Repository Query Using fn:doc	5-49
5-46	Optimized Repository Query Using EQUALS_PATH	5-49
5-47	Repository Query Using Oracle XQuery Pragma ora:defaultTable	5-50
6-1	Making Query Data Compatible with Index Data – SQL Cast	6-10
6-2	Making Query Data Compatible with Index Data – XQuery Cast	6-10
6-3	Exchange-Partitioning Tables That Have an XMLIndex Structured Component	6-11
6-4	Exchange-Partitioning Reference-Partitioned Tables That Use XMLIndex	6-11
6-5	Data Used in Example of Exchange-Partitioning for Reference-Partitioned Tables	6-12
6-6	Creating an XMLIndex Index	6-13
6-7	Obtaining the Name of an XMLIndex Index on a Particular Table	6-13
6-8	Renaming and Dropping an XMLIndex Index	6-14
6-9	XMLIndex with a Structured Component, Using Namespaces and Storage Options	6-16
6-10	XMLIndex with a Structured Component, Specifying TABLESPACE at the Index Level	6-16
6-11	XMLIndex Index: Adding a Structured Component	6-17
6-12	Using DBMS_XMLINDEX.PROCESS_PENDING To Index XML Data	6-18
6-13	Dropping an XMLIndex Structured Component	6-20
6-14	Creating a B-tree Index on an XMLIndex Index Content Table	6-20
6-15	Using a Structured XMLIndex Component for a Query with Multilevel Chaining	6-21
6-16	Turning Off XMLIndex Using Optimizer Hints	6-23
6-17	Creating an XMLIndex Index in Parallel	6-24
6-18	Using Different PARALLEL Degrees for XMLIndex Internal Objects	6-24
6-19	Automatic Collection of Statistics on XMLIndex Objects	6-25
6-20	XQuery Full Text Query	6-42



6-21	Execution Plan for XQuery Full Text Query	6-42
6-22	A Range Search Query in an XMLExists predicate answered by an XML Search Index	6-43
6-23	Execution Plan for a Range Search Query in an XMLExists predicate answered by an XML	
	Search Index	6-43
6-24	CREATE INDEX Using XMLCAST and XMLQUERY on a Singleton Element	6-47
6-25	CREATE INDEX Using EXTRACTVALUE on a Singleton Element	6-47
7-1	XSLT Stylesheet Example: PurchaseOrder.xsl	7-2
7-2	Registering an XML Schema and Inserting XML Data	7-4
7-3	Using SQL Function XMLTRANSFORM to Apply an XSL Stylesheet	7-6
7-4	Using XMLType Method TRANSFORM() with a Transient XSL Stylesheet	7-7
7-5	Using XMLTRANSFORM to Apply an XSL Stylesheet Retrieved Using XDBURIType	7-8
7-6	Error When Inserting Incorrect XML Document (Partial Validation)	7-13
7-7	Forcing Full XML Schema Validation Using a CHECK Constraint	7-14
7-8	Enforcing Full XML Schema Validation Using a BEFORE INSERT Trigger	7-14
7-9	Validating XML Using Method ISSCHEMAVALID() in SQL	7-15
7-10	Validating XML Using Method ISSCHEMAVALID() in PL/SQL	7-15
7-11	Validating XML Using Method SCHEMAVALIDATE() within Triggers	7-16
7-12	Checking XML Validity Using XMLISVALID Within CHECK Constraints	7-16
8-1	XMLELEMENT: Formatting a Date	8-8
8-2	XMLELEMENT: Generating an Element for Each Employee	8-8
8-3	XMLELEMENT: Generating Nested XML	8-8
8-4	XMLELEMENT: Generating Employee Elements with Attributes ID and Name	8-9
8-5	XMLELEMENT: Characters in Generated XML Data Are Not Escaped	8-9
8-6	Creating a Schema-Based XML Document Using XMLELEMENT with Namespaces	8-9
8-7	XMLELEMENT: Generating an Element from a User-Defined Data-Type Instance	8-10
8-8	XMLFOREST: Generating Elements with Attribute and Child Elements	8-11
8-9	XMLFOREST: Generating an Element from a User-Defined Data-Type Instance	8-12
8-10	XMLCONCAT: Concatenating XMLType Instances from a Sequence	8-13
8-11	XMLCONCAT: Concatenating XML Elements	8-13
8-12	XMLAGG: Generating a Department Element with Child Employee Elements	8-14
8-13	XMLAGG: Using GROUP BY to Generate Multiple Department Elements	8-15
8-14	XMLAGG: Generating Nested Elements	8-15
8-15	Using SQL/XML Function XMLPI	8-17
8-16	Using SQL/XML Function XMLCOMMENT	8-18
8-17	Using SQL/XML Function XMLSERIALIZE	8-19
8-18	Using SQL/XML Function XMLPARSE	8-20
8-19	XMLCOLATTVAL: Generating Elements with Attribute and Child Elements	8-21

8-20	Using Oracle SQL Function XMLCDATA	8-22
8-21	DBMS_XMLGEN: Generating Simple XML	8-33
8-22	DBMS_XMLGEN: Generating Simple XML with Pagination (Fetch)	8-33
8-23	DBMS_XMLGEN: Generating XML Using Object Types	8-35
8-24	DBMS_XMLGEN: Generating XML Using User-Defined Data-Type Instances	8-36
8-25	DBMS_XMLGEN: Generating an XML Purchase Order	8-38
8-26	DBMS_XMLGEN: Generating a New Context Handle from a REF Cursor	8-42
8-27	DBMS_XMLGEN: Specifying NULL Handling	8-43
8-28	DBMS_XMLGEN: Generating Recursive XML with a Hierarchical Query	8-44
8-29	DBMS_XMLGEN: Binding Query Variables Using SETBINDVALUE()	8-46
8-30	Using XMLAGG ORDER BY Clause	8-49
8-31	Returning a Rowset Using XMLTABLE	8-50
9-1	Creating a Relational View of XML Content	9-2
9-2	Accessing Individual Members of a Collection Using a View	9-4
9-3	XMLIndex Index that Matches Relational View Columns	9-5
9-4	XMLTable Expression Returned by PL/SQL Function getSIDXDefFromView	9-5
9-5	Querying Master Relational View of XML Data	9-6
9-6	Querying Master and Detail Relational Views of XML Data	9-6
9-7	Business-Intelligence Query of XML Data Using a View	9-6
10-1	Creating an XMLType View Using XMLELEMENT	10-3
10-2	Registering XML Schema emp_simple.xsd	10-5
10-3	Creating an XMLType View Using SQL/XML Publishing Functions	10-6
10-4	Querying an XMLType View	10-6
10-5	Using Namespace Prefixes with SQL/XML Publishing Functions	10-7
10-6	XML Schema with No Target Namespace	10-8
10-7	Creating a View for an XML Schema with No Target Namespace	10-8
10-8	Using SQL/XML Functions in XML Schema-Based XMLType Views	10-9
10-9	Creating Object Types for Schema-Based XMLType Views	10-12
10-10	Creating and Registering XML Schema emp_complex.xsd	10-13
10-11	Creating XMLType View emp_xml Using Object Type emp_t	10-14
10-12	Creating an Object View and an XMLType View Based on the Object View	10-15
10-13	Creating Object Types	10-15
10-14	Registering XML Schema dept_complex.xsd	10-16
10-15	Creating XMLType View dept_xml Using Object Type dept_t	10-17
10-16	Creating XMLType View dept_xml Using Relational Data Directly	10-18
10-17	Creating an XMLType View by Restricting Rows from an XMLType Table	10-18
10-18	Creating an XMLType View by Transforming an XMLType Table	10-19

10-19	Determining Whether an XMLType View Is Implicitly Updatable, and Updating It	10-19
11-1	Creating and Manipulating a DOM Document	11-17
11-2	Creating an Element Node and Obtaining Information About It	11-18
11-3	Creating a User-Defined Subtype of SYS.util_BinaryOutputStream()	11-21
11-4	Retrieving Node Value with a User-Defined Stream	11-21
11-5	Get-Pull of Binary Data	11-22
11-6	Get-Pull of Character Data	11-23
11-7	Set-Pull of Binary Data	11-24
11-8	Set-Push of Binary Data	11-25
11-9	Parsing an XML Document	11-27
11-10	Transforming an XML Document Using an XSL Stylesheet	11-30
12-1	Inserting Data with Specified Columns	12-2
12-2	Updating Data with Key Columns	12-4
12-3	DBMS_XMLSTORE.DELETEXML Example	12-5
13-1	Querying an XMLType Table Using JDBC	13-3
13-2	Selecting XMLType Data Using getString() and getCLOB()	13-3
13-3	Returning XMLType Data Using getSQLXML()	13-4
13-4	Returning XMLType Data Using an Output Parameter	13-4
13-5	Updating an XMLType Column Using SQL Constructor XMLType and Java String	13-6
13-6	Updating an XMLType Column Using SQLXML	13-6
13-7	Retrieving Metadata About an XMLType Column Using JDBC	13-6
13-8	Updating an XMLType Column Using JDBC	13-6
13-9	Updated Purchase-Order Document	13-8
13-10	Inserting an XMLType column using JDBC	13-10
13-11	Converting an XML String to an OracleClob Instance	13-10
13-12	Policy File Granting Permissions for Java DOM API	13-11
13-13	Creating a DOM Object with the Java DOM API	13-12
13-14	Using the Java DOM API with a Binary XML Column	13-20
14-1	Using OCIXMLDBINITXMLCTX() and OCIXMLDBFREEXMLCTX()	14-5
14-2	Using the C API for XML with Binary XML	14-9
14-3	Using the Oracle XML DB Pull Parser	14-11
14-4	Using the DOM to Count Ordered Parts	14-16
15-1	Retrieve XMLType Data to .NET	15-2
17-1	Registering an XML Schema Using DBMS_XMLSCHEMA.REGISTERSCHEMA	17-12
17-2	Objects Created During XML Schema Registration	17-12
17-3	Registering a Local XML Schema	17-15
17-4	Registering a Global XML Schema	17-16

17-5	Deleting an XML Schema with DBMS_XMLSCHEMA.DELETESCHEMA	17-18
17-6	Data Dictionary Table for Registered Schemas	17-18
17-7	Creating XML Schema-Based XMLType Tables and Columns	17-21
17-8	Creating an Object-Relational XMLType Table with Default Storage	17-26
17-9	Specifying Object-Relational Storage Options for XMLType Tables and Columns	17-26
17-10	Using STORE ALL VARRAYS AS	17-20
18-1	SQL Object Types for Storing XMLType Tables	18-4
18-2	Default Table for Global Element PurchaseOrder	18-4
18-3	Using Common Schema Annotations	18-5
18-4	Registering an Annotated XML Schema	18-11
18-5		18-14
18-6	Querying View USER_XML_SCHEMAS for a Registered XML Schema	18-20
18-7	Querying Metadata from a Registered XML Schema	18-20
18-8	Mapping XML Schema Data Types to SQL Data Types Using Attribute SQLType	18-22
18-9	XML Schema Inheritance: complexContent as an Extension of complexTypes	18-32
18-10	Inheritance in XML Schema: Restrictions in complexTypes	18-33
18-11	XML Schema complexType: Mapping complexType to simpleContent	18-34
18-12	XML Schema: Mapping complexType to any/anyAttribute	18-35
18-13	Creating an XMLType Table that Conforms to an XML Schema	18-36
18-14	Creating an XMLType Table for Nested Collections	18-37
18-15	Using DESCRIBE with an XML Schema-Based XMLType Table	18-37
18-16	Specifying Partitioning Information During XML Schema Registration	18-39
18-17	Specifying Partitioning Information During Table Creation	18-39
18-18	Integrity Constraints and Triggers for an XMLType Table Stored Object-Relationally	18-41
18-19	Adding a Unique Constraint to the Parent Element of an Attribute	18-42
18-20	Setting SQLInline to False for Out-Of-Line Storage	18-45
18-21	Generated XMLType Tables and Types	18-45
18-22	Querying an Out-Of-Line Table	18-46
18-23	Storing a Collection Out of Line	18-47
18-24	Generated Out-Of-Line Collection Type	18-48
18-25	Renaming an Intermediate Table of REF Values	18-48
18-26	XPath Rewrite for an Out-Of-Line Collection	18-48
18-27	XPath Rewrite for an Out-Of-Line Collection, with Index on REFs	18-49
18-28	An XML Schema with Circular Dependency	18-50
18-29	XML Schema: Cycling Between complexTypes	18-53
18-30	XML Schema: Cycling Between complexTypes, Self-Reference	18-53
18-31	An XML Schema that Includes a Non-Existent XML Schema	18-55
•		20 00

18-32	Using the FORCE Option to Register XML Schema xm40.xsd	18-55
18-33	Trying to Create a Table Using a Cyclic XML Schema	18-55
18-34	Using the FORCE Option to Register XML Schema xm40a.xsd	18-56
18-35	Recursive XML Schema	18-57
18-36	Out-of-line Table	18-58
18-37	Invalid Default Table Sharing	18-58
18-38	Oracle XML DB XML Schema: Mapping complexType XML Fragments to LOBs	18-60
19-1	XPath Rewrite	19-2
19-2	XPath Rewrite for an Out-Of-Line Table	19-4
19-3	Using an Index with an Out-Of-Line Table	19-5
19-4	Execution Plan Generated When XPath Rewrite Does Not Occur	19-6
19-5	Analyzing an Execution Plan to Determine a Column to Index	19-8
19-6	Using DBMS_XMLSTORAGE_MANAGE.XPATH2TABCOLMAPPING	19-8
19-7	Creating an Index on a Column Targeted by a Predicate	19-8
19-8	Creating a Function-Based Index for a Column Targeted by a Predicate	19-8
19-9	Execution Plan Showing that Index Is Picked Up	19-9
19-10	Creating a Function-Based Index for a Column Targeted by a Predicate	19-9
19-11	Execution Plan for a Selection of Collection Elements	19-10
19-12	Creating an Index for Direct Access to an Ordered Collection Table	19-10
20-1	Revised Purchase-Order XML Schema	20-3
20-2	evolvePurchaseOrder.xsl: XSLT Stylesheet to Update Instance Documents	20-11
20-3	Loading Revised XML Schema and XSLT Stylesheet	20-15
20-4	Updating an XML Schema Using DBMS_XMLSCHEMA.COPYEVOLVE	20-15
20-5	Splitting a Complex Type into Two Complex Types	20-18
20-6	diffXML Parameter Document	20-25
21-1	Querying PATH_VIEW to Determine Link Type	21-15
21-2	Obtaining the OID Path of a Resource	21-15
21-3	Creating a Weak Link Using an OID Path	21-16
21-4	Accessing a Text Document in the Repository Using XDBURITYPE	21-31
21-5	Accessing Resource Content Using RESOURCE_VIEW	21-31
21-6	Accessing XML Documents Using Resource and Namespace Prefixes	21-32
21-7	Querying Repository Resource Data Using SQL Function REF and Element XMLRef	21-33
21-8	Selecting XML Document Fragments Based on Metadata, Path, and Content	21-33
21-9	Updating a Text Document Using UPDATE and XQuery Update on the Resource	21-35
21-10	Updating an XML Node Using UPDATE and XQuery Update on the Resource	21-36
21-11	Updating XML Schema-Based Documents in the Repository	21-37
21-12	Accessing Resources Using EQUALS_PATH and RESOURCE_VIEW	21-38

21-13	Determining the Path to XSLT Stylesheets Stored in the Repository	21-39
21-14	Counting Resources Under a Path	21-40
21-15	Listing the Folder Contents in a Path	21-40
21-16	Listing the Links Contained in a Folder	21-40
21-17	Finding Paths to Resources that Contain Purchase-Order XML Documents	21-41
21-18	Execution Plan Output for a Folder-Restricted Query	21-41
22-1	Resource Configuration File	22-5
22-2	applicationData Element	22-6
23-1	XInclude Used in a Book Document to Include Parts and Chapters	23-5
23-2	Expanding Document Inclusions Using XDBURIType	23-8
23-3	Querying Document Links Mapped From XLink Links	23-10
23-4	Querying Document Links Mapped From XInclude Links	23-11
23-5	Mapping XInclude Links to Hard Document Links, with OID Retrieval	23-17
23-6	Mapping XLInk Links to Weak Links, with Named-Path Retrieval	23-17
23-7	Configuring XInclude Document Decomposition	23-17
23-8	Repository Document, Showing Generated xi:include Elements	23-18
24-1	Determining Paths Under a Path: Relative	24-10
24-2	Determining Paths Under a Path: Absolute	24-10
24-3	Determining Paths Not Under a Path	24-11
24-4	Determining Paths Using Multiple Correlations	24-11
24-5	Relative Path Names for Three Levels of Resources	24-12
24-6	Extracting Resource Metadata Using UNDER_PATH	24-12
24-7	Using Functions PATH and DEPTH with PATH_VIEW	24-13
24-8	Extracting Link and Resource Information from PATH_VIEW	24-13
24-9	All Repository Paths to a Certain Depth Under a Path	24-14
24-10	Locating a Repository Path Using EQUALS_PATH	24-15
24-11	Retrieve RESID of a Given Resource	24-15
24-12	Obtaining the Path Name of a Resource from its RESID	24-15
24-13	Folders Under a Given Path	24-16
24-14	Joining RESOURCE_VIEW with an XMLType Table	24-16
24-15	Deleting Resources	24-17
24-16	Deleting Links to Resources	24-17
24-17	Deleting a Nonempty Folder	24-18
24-18	Updating a Resource	24-19
24-19	Updating a Path in the PATH_VIEW	24-20
24-20	Updating Resources Based on Attributes	24-21
24-21	Finding Resources Inside a Folder	24-21

24-22	Copying Resources	24-22
24-23	Find All Resources Containing "Paper"	24-24
24-24	Find All Resources Containing "Paper" that are Under a Specified Path	24-24
25-1	Creating a Repository Resource	25-7
25-2	Creating a Version-Controlled Resource	25-7
25-3	Retrieving Resource Content by Referencing the Resource ID	25-8
25-4	Checking Out a Version-Controlled Resource	25-8
25-5	Updating Resource Content	25-8
25-6	Checking In a Version-Controlled Resource	25-9
25-7	Retrieving Resource Version Content Using XDBURITYPE and CREATEOIDPATH	25-9
25-8	Retrieving Resource Version Content Using GETCONTENTSCLOBBYRESID	25-9
25-9	Retrieving Resource Version Metadata Using GETRESOURCEBYRESID	25-10
25-10	Canceling a Check-Out Using UNCHECKOUT	25-11
26-1	Managing Resources Using DBMS_XDB_REPOS	26-3
26-2	Using DBMS_XDB_REPOS.GETACLDOCUMENT	26-4
26-3	Using DBMS_XDB_REPOS.SETACL	26-5
26-4	Using DBMS_XDB_REPOS.CHANGEPRIVILEGES	26-5
26-5	Using DBMS_XDB_REPOS.GETPRIVILEGES	26-6
26-6	Using DBMS_XDB_CONFIG.CFG_GET	26-8
26-7	Using DBMS_XDB_CONFIG.CFG_UPDATE	26-9
27-1	Simple Access Control Entry (ACE) that Grants a Privilege	27-5
27-2	Simple Access Control List (ACL) that Grants a Privilege	27-6
27-3	Complementing a Set of Principals with Element invert	27-12
27-4	Creating an ACL Using CREATERESOURCE	27-13
27-5	Retrieving an ACL Document, Given its Repository Path	27-14
27-6	Setting the ACL of a Resource	27-15
27-7	Deleting an ACL	27-15
27-8	Updating (Replacing) an Access Control List	27-16
27-9	Appending ACEs to an Access Control List	27-16
27-10	Deleting an ACE from an Access Control List	27-16
27-11	Retrieving the ACL Document for a Resource	27-17
27-12	Retrieving Privileges Granted to the Current User for a Particular Resource	27-17
27-13	Checking If a User Has a Certain Privileges on a Resource	27-18
27-14	Checking User Privileges Using ACLCheckPrivileges	27-19
27-15	Retrieving the Path of the ACL that Protects a Given Resource	27-20
27-16	Retrieving the Paths of All Resources Protected by a Given ACL	27-20
27-17	ACL Referencing an LDAP User	27-24

27-18	ACL Referencing an LDAP Group	27-24
28-1	Listener Status with FTP and HTTP(S) Protocol Support Enabled	28-10
28-2	Uploading Content to the Repository Using FTP	28-18
28-3	Navigating Oracle ASM Folders	28-20
28-4	Transferring Oracle ASM Files Between Databases with FTP proxy Method	28-20
28-5	FTP Connection Using IPv6	28-22
28-6	Modifying the Default Timeout Value of an FTP Session	28-23
29-1	Registering an XML Schema for Technical Photo Information	29-4
29-2	Registering an XML Schema for Photo Categorization	29-5
29-3	Add Metadata to a Resource – Technical Photo Information	29-7
29-4	Add Metadata to a Resource – Photo Content Categories	29-7
29-5	Delete Specific Metadata from a Resource	29-8
29-6	Adding Metadata to a Resource Using DML with RESOURCE_VIEW	29-8
29-7	Adding Metadata Using WebDAV PROPPATCH	29-10
29-8	Query XML Schema-Based Resource Metadata	29-11
29-9	Add Non-Schema-Based Metadata to a Resource	29-12
30-1	Resource Configuration File for Java Event Listeners with Preconditions	30-13
30-2	Resource Configuration File for PL/SQL Event Listeners with No Preconditions	30-14
30-3	PL/SQL Code Implementing Event Listeners	30-14
30-4	Java Code Implementing Event Listeners	30-16
30-5	Invoking Event Handlers	30-18
31-1	An Oracle XML DB Servlet	31-9
31-2	Registering and Mapping an Oracle XML DB Servlet	31-10
32-1	Using HTTPURIType PL/SQL Method GETCONTENTTYPE()	32-7
32-2	Creating and Querying a URI Column	32-9
32-3	Using Different Kinds of URI, Created in Different Ways	32-10
32-4	Access a Repository Resource by URI Using an XDBUri	32-12
32-5	Using PL/SQL Method GETXML with XMLCAST and XMLQUERY	32-14
32-6	Targeting a Complete Table Using a DBUri	32-19
32-7	Targeting a Particular Row in a Table Using a DBUri	32-20
32-8	Targeting a Specific Column Using a DBUri	32-21
32-9	Targeting an Object Column with Specific Attribute Values Using a DBUri	32-21
32-10	Retrieve Only the Text Value of a Node Using a DBUri	32-22
32-11	Targeting a Collection Using a DBUri	32-23
32-12	URIFACTORY: Registering the ECOM Protocol	32-25
32-13	SYS_DBURIGEN: Generating a DBUri that Targets a Column	32-26
32-14	Passing Columns with Single Arguments to SYS_DBURIGEN	32-27

32-15	Inserting Database References Using SYS_DBURIGEN	32-28
32-16	Creating the Travel Story Table	32-29
32-17	A Function that Returns the First 20 Characters	32-29
32-18	Creating a Travel View for Use with SYS_DBURIGEN	32-29
32-19	Retrieving a URL Using SYS_DBURIGEN in RETURNING Clause	32-30
32-20	Changing the Installation Location of DBUriServlet	32-33
32-21	Restricting Servlet Access to a Database Role	32-34
32-22	Registering a Handler for a DBUri Prefix	32-35
33-1	Adding a Web Services Configuration Servlet	33-3
33-2	Verifying Addition of Web Services Configuration Servlet	33-3
33-3	XML Schema for Database Queries To Be Processed by Web Service	33-5
33-4	Input XML Document for SQL Query Using Query Web Service	33-6
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	33-8
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	33-10
33-9	Output XML Document for PL/SQL Query Using Web Service	33-10
34-1	Oracle XML DB Configuration File	34-10
34-2	Updating the Configuration File Using CFG_UPDATE and CFG_GET	34-12
35-1	Data File filelist.dat: List of XML Files to Load	35-5
35-2	Control File load_datra.ctl, for Loading Purchase-Order XML Documents	35-5
35-3	Loading XML Data Using Shell Command sqlldr	35-5
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	Unannotated Purchase-Order XML Schema	A-35
A-2	Annotated Purchase-Order XML Schema	A-37
A-3	Revised Annotated Purchase-Order XML Schema	A-40
A-4	PurchaseOrder.xsl XSLT Stylesheet	A-44
A-5	Inserting XML Data into an XMLType Table Using C	A-50

A-6	Using OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx()	A-54
C-1	Path Table Contents for Two Purchase Orders	C-3
C-2	Naming the Path Table of an XMLIndex Index	C-10
C-3	Determining the System-Generated Name of an XMLIndex Path Table	C-10
C-4	Specifying Storage Options When Creating an XMLIndex Index	C-10
C-5	Dropping an XMLIndex Unstructured Component	C-11
C-6	Determining the Names of the Secondary Indexes of an XMLIndex Index	C-11
C-7	Obtaining the Name of an XMLIndex Index from Its Path-Table Name	C-13
C-8	Extracting Data from an XML Fragment Using XMLIndex	C-13
C-9	Creating a Function-Based Index on Path-Table Column VALUE	C-15
C-10	Trying to Create a Numeric Index on Path-Table Column VALUE Directly	C-15
C-11	Creating a Numeric Index on Column VALUE with Procedure createNumberIndex	C-15
C-12	Creating a Date Index on Column VALUE with Procedure createDateIndex	C-15
C-13	Creating an Oracle Text CONTEXT Index on Path-Table Column VALUE	C-15
C-14	Showing All Secondary Indexes on an XMLIndex Path Table	C-15
C-15	Specifying Deferred Synchronization for XMLIndex	C-17
C-16	Manually Synchronizing an XMLIndex Index Using SYNCINDEX	C-17
C-17	XMLIndex Path Subsetting with CREATE INDEX	C-20
C-18	XMLIndex Path Subsetting with ALTER INDEX	C-20
C-19	XMLIndex Path Subsetting Using a Namespace Prefix	C-21
D-1	Creating an XML Search Index	D-3
D-2	XQuery Full Text Query	D-3
D-3	Execution Plan for XQuery Full Text Query	D-3
D-4	XQuery Full Text Query with XML Schema-Based Data: Error ORA-18177	D-5
D-5	Using XQuery Pragma ora:no_schema with XML Schema-Based Data	D-5
D-6	Full-Text Query with XQuery Pragma ora:use_xmltext_idx	D-5

List of Figures

1-1	Oracle XML DB Benefits	1-3
1-2	Unifying Data and Content: Some Common XML Architectures	1-4
1-3	XMLType Storage	1-10
1-4	Oracle XML DB Repository Architecture	1-19
2-1	Oracle XML DB Storage Options for XML Data	2-4
3-1	Loading Content into the Repository Using Windows Explorer	3-17
4-1	XMLQUERY Syntax	4-12
4-2	XMLTABLE Syntax	4-13
4-3	XMLExists Syntax	4-17
4-4	XMLCast Syntax	4-19
7-1	XMLTRANSFORM Syntax	7-4
7-2	Using XMLTRANSFORM	7-4
7-3	Database XSL Transformation of a PurchaseOrder Using DBUri Servlet	7-10
7-4	Database XSL Transformation of Departments Table Using DBUri Servlet	7-11
8-1	XMLELEMENT Syntax	8-4
8-2	XMLAttributes Clause Syntax (XMLATTRIBUTES)	8-5
8-3	XMLFOREST Syntax	8-11
8-4	XMLCONCAT Syntax	8-13
8-5	XMLAGG Syntax	8-14
8-6	XMLPI Syntax	8-17
8-7	XMLComment Syntax	8-17
8-8	XMLSerialize Syntax	8-18
8-9	XMLParse Syntax	8-20
8-10	XMLCOLATTVAL Syntax	8-21
8-11	XMLCDATA Syntax	8-22
8-12	Using PL/SQL Package DBMS_XMLGEN	8-25
8-13	SYS_XMLAGG Syntax	8-48
10-1	Creating XMLType Views Clause: Syntax	10-3
11-1	Using the PL/SQL DOM API for XMLType	11-16
11-2	Using the PL/SQL Parser API for XMLType	11-26
11-3	Using the PL/SQL XSLT Processor for XMLType	11-29
13-1	Using the Java DOM API for XMLType	13-14
16-1	XML Use Cases and XMLType Storage Models	16-3
17-1	XMLSpy Graphical Representation of a Purchase-Order XML Schema	17-4
17-2	XMLSpy Support for Oracle XML DB Schema Annotations	17-7
17-3	Creating an XMLType Table – CREATE TABLE Syntax	17-20

17-4	Creating an XMLType Table – XMLType_table Syntax	17-20
17-5	Creating an XMLType Table – table_properties Syntax	17-20
17-6	Creating an XMLType Table – XMLType_virtual_columns Syntax	17-21
17-7	How Oracle XML DB Maps XML Schema-Based XMLType Tables	17-30
18-1	simpleType Mapping: XML Strings to SQL VARCHAR2 or CLOB	18-26
18-2	Mapping complexType to SQL for Out-Of-Line Storage	18-44
18-3	Cross Referencing Between Different complexTypes in the Same XML Schema	18-52
18-4	Self-Referencing Complex Type within an XML Schema	18-52
18-5	Cyclical References Between XML Schemas	18-54
18-6	Mapping complexType XML Fragments to CLOB Instances	18-60
21-1	A Folder Tree, Showing Hierarchical Structures in the Repository	21-3
21-2	Oracle XML DB Folders in Windows Explorer	21-17
21-3	Accessing Repository Data Using HTTP(S)/WebDAV and a Web Browser	21-18
21-4	Path-Based Access Using HTTP and a URL	21-18
21-5	Oracle ASM Virtual Folder Hierarchy	21-22
21-6	Updating and Editing Content Stored in Oracle XML DB Using Microsoft Word	21-34
24-1	Accessing Repository Resources Using RESOURCE_VIEW and PATH_VIEW	24-3
24-2	RESOURCE_VIEW and PATH_VIEW Structure	24-3
24-3	RESOURCE_VIEW and PATH_VIEW Explained	24-6
24-4	UNDER_PATH Syntax	24-7
24-5	EQUALS_PATH Syntax	24-9
24-6	PATH Syntax	24-9
28-1	Oracle XML DB Architecture: Protocol Server	28-3
28-2	Creating a WebFolder in Microsoft Windows	28-34
28-3	Copying Files into Oracle XML DB Repository	28-35
32-1	A DBUri Corresponds to an XML Visualization of Relational Data	32-15
32-2	SYS_DBURIGEN Syntax	32-26
37-1	Oracle Database Advanced Queuing and XML Message Payloads	37-3
37-2	iDAP Architecture for Performing AQ Operations Using HTTP(S)	37-5
C-1	XML Use Cases and XML Indexing	C-3

List of Tables

1-1	Static Data Dictionary Views Related to XML	1-16
3-1	SQL*Loader – Conventional and Direct-Path Load Modes	3-16
4-1	Common XPath Constructs	4-2
4-2	Predefined Namespaces and Prefixes	4-10
4-3	oradb Expressions: Column Types for Comparisons	4-22
6-1	Basic XML Indexing Tasks	6-2
6-2	Tasks Involving XMLIndex Indexes with a Structured Component	6-2
6-3	Tasks involving XML Search Index	6-2
6-4	XML Use Cases and XML Indexing	6-6
6-5	XML and SQL Data Type Correspondence for XMLIndex	6-9
6-6	XMLIndex Static Public Views	6-25
8-1	DBMS_XMLGEN Functions and Procedures	8-26
11-1	PL/SQL APIs Related to XML	11-3
11-2	XML and HTML DOM Node Types and Their Child Node Types	11-13
13-1	Java DOM API for XMLType: Classes	13-13
14-1	OCIXmIDbInitXMICtx() Parameters	14-3
14-2	Common XMLType Operations in C	14-16
16-1	XMLType Storage Model Considerations	16-8
16-2	XMLType Indexing Considerations	16-9
16-3	XMLType Storage Models: Relative Advantages	16-10
17-1	XMLType Methods Related to XML Schema	17-8
17-2	CREATE TABLE Encoding Options for Binary XML	17-24
18-1	Annotations in Elements	18-15
18-2	Annotations in Elements Declaring Global complexType Elements	18-16
18-3	XML Schema String Data Types Mapped to SQL	18-26
18-4	XML Schema Binary Data Types (hexBinary/base64Binary) Mapped to SQL	18-26
18-5	Default Mapping of Numeric XML Schema Primitive Types to SQL	18-27
18-6	XML Schema Date and Time Data Types Mapped to SQL	18-27
18-7	Default Mapping of Other XML Schema Primitive and Derived Data Types to SQL	18-27
19-1	Sample of XPath Expressions that Are Rewritten to Underlying SQL Constructs	19-3
20-1	Parameters of Procedure DBMS_XMLSCHEMA.COPYEVOLVE	20-6
20-2	Errors Associated with Procedure DBMS_XMLSCHEMA.COPYEVOLVE	20-7
20-3	XML Schema Evolution: XMLType Table Temporary Table Columns	20-14
20-4	XML Schema Evolution: XMLType Column Temporary Table Columns	20-14
20-5	Procedure COPYEVOLVE Mapping Table	20-15
20-6	Parameters of Procedure DBMS_XMLSCHEMA.INPLACEEVOLVE	20-22

21-1	Synonyms for Oracle XML DB Repository Terms	21-8
21-2	Differences Between PATH_VIEW and RESOURCE_VIEW	21-23
21-3	Accessing Oracle XML DB Repository: API Options	21-25
24-1	Structure of RESOURCE_VIEW	24-5
24-2	Structure of PATH_VIEW	24-5
24-3	UNDER_PATH SQL Function Signature	24-7
25-1	Oracle XML DB Versioning Terms	25-2
25-2	PL/SQL Functions and Procedures in Package DBMS_XDB_VERSION	25-3
26-1	DBMS_XDB_REPOS Resource Access and Management Subprograms	26-2
26-2	DBMS_XDB_REPOS: Security Management Subprograms	26-4
26-3	DBMS_XDB_CONFIG: Configuration Management Subprograms	26-7
27-1	Database Privileges Needed for Operations on Oracle XML DB Resources	27-6
27-2	Atomic Privileges	27-8
27-3	Aggregate Privileges	27-8
28-1	Common Protocol Configuration Parameters	28-5
28-2	Configuration Parameters Specific to FTP	28-6
28-3	Configuration Parameters Specific to HTTP(S)/WebDAV (Except Servlet)	28-6
30-1	Predefined Repository Events	30-5
30-2	Oracle XML DB Repository Operations and Events	30-7
31-1	XML Elements Defined for Servlet Deployment Descriptors	31-3
31-2	Java Servlet 2.2 Methods that Are Not Implemented	31-8
32-1	URIType PL/SQL Methods	32-5
32-2	URIFACTORY PL/SQL Methods	32-23
32-3	DBUriServlet: Optional Arguments	32-31
33-1	Web Service Mapping Between XML and Oracle Database Data Types	33-7
34-1	DBMS_XDB_ADMIN Management Procedures	34-17
C-1	XMLIndex Path Table	C-2
C-2	Tasks Involving XMLIndex Indexes with an Unstructured Component	C-2
C-3	Index Synchronization	C-16
D-1	Migrating Oracle-Specific XML Queries to XQuery Full Text	D-6

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 Database Advanced Queuing XMLType support.

Audience

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

- Documentation Accessibility
- Diversity and Inclusion
- Related Documents
- Conventions
- 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.
- Syntax Descriptions
 Syntax descriptions are provided for various SQL, PL/SQL, or other command-line constructs in graphic form or Backus Naur Form (BNF).

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

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

For more information, see these Oracle resources:

- Oracle Database XML Java API Reference
- Oracle XML Developer's Kit Programmer's Guide
- Oracle Database Error Messages
- Oracle Text Application Developer's Guide
- Oracle Text Reference
- Oracle Database Concepts
- Oracle Database Java Developer's Guide
- Oracle Database Development Guide
- Oracle Database Advanced Queuing User's Guide
- Oracle Database PL/SQL Packages and Types Reference

Many of the examples in this book use the Oracle Database sample schemas. Refer to *Oracle Database Sample Schemas* for information about how these database 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 OTN Registration

For additional information, see:

- Extensible Markup Language (XML) 1.0
- XML Schema and XML Schema resources
- XML Schema Part 0: Primer
- XML Schema Part 1: Structures
- XML Schema Part 2: Datatypes



- XML Schemas reference list
- XML and MIME Media-Types
- XML Pointer Language (XPointer)
- XML Path Language (XPath) Version 1.0
- XML Path Language (XPath) 2.0
- XPath Tutorial
- Unicode in XML and other Markup Languages, Unicode Technical Report #20
- Namespaces in XML 1.0
- XML Information Set
- Document Object Model (DOM)
- XSL Transformations (XSLT) Version 1.0
- Extensible Stylesheet Language (XSL) Version 1.1
- XSL references
- XSLT Tutorial
- Web Services Activity
- FTP Protocol Specification, IETF RFC959
- ISO/IEC 13249-2:2000, Information technology Database languages SQL Multimedia and Application Packages - Part 2: Full-Text, International Organization For Standardization, 2000

Note:

Throughout this manual, **XML Schema** refers to the XML Schema 1.0 recommendation.

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.

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 SQL is case-insensitive. XML is case-sensitive. Keep this in mind when you mix SQL and XML.

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 XSLT stylesheet 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 stylesheet can also be found in 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

SQL is case-insensitive. XML is case-sensitive. Keep this in mind when you mix SQL and XML.

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 quotation marks (").
- 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 quotation marks, then you must refer to it in XML code as "MY TABLE".

Syntax Descriptions

Syntax descriptions are provided 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.



Part I Oracle XML DB Basics

The basics of Oracle XML DB are introduced.

- Introduction to Oracle XML DB
 Oracle XML DB provides Oracle Database with native XML support by encompassing both the SQL and XML data models in an interoperable way.
- Getting Started with Oracle XML DB Some preliminary design criteria are presented for consideration when planning your Oracle XML DB solution.
- Overview of How To Use Oracle XML DB An overview of the various ways of using Oracle XML DB is presented.



1 Introduction to Oracle XML DB

Oracle XML DB provides Oracle Database with native XML support by encompassing both the SQL and XML data models in an interoperable way.

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 Benefits

Oracle XML DB supports all major XML, SQL, Java, and Internet standards. It provides high performance and scalability for XML operations. It brings database features such as transaction control, data integrity, replication, reliability, availability, security, and scalability to the world of XML.

Oracle XML DB Architecture

Oracle XML DB gives you protocol and programmatic access to XML data in the form of local and remote XMLType tables and views. It provides a WebDAV repository with resource versioning and access control.

Oracle XML DB Features

Oracle XML DB 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.

- Standards Supported by Oracle XML DB Oracle XML DB supports all major XML, SQL, Java, and Internet standards.
- Oracle XML DB Technical Support

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

Oracle XML DB Examples

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

 Oracle XML DB Case Studies and Demonstrations on OTN Visit Oracle Technology Network (OTN) to view Oracle XML DB examples, white papers, 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 Oracle9i Release 2 (9.2).



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.

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. 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 and XQuery.
- 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 Fusion Middleware or Oracle Database.

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

Related Topics

XMLType Data Type

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

See Also:

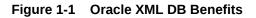
- Oracle XML DB on OTN for the latest news and white papers about Oracle XML DB
- Oracle XML Developer's Kit Programmer's Guide

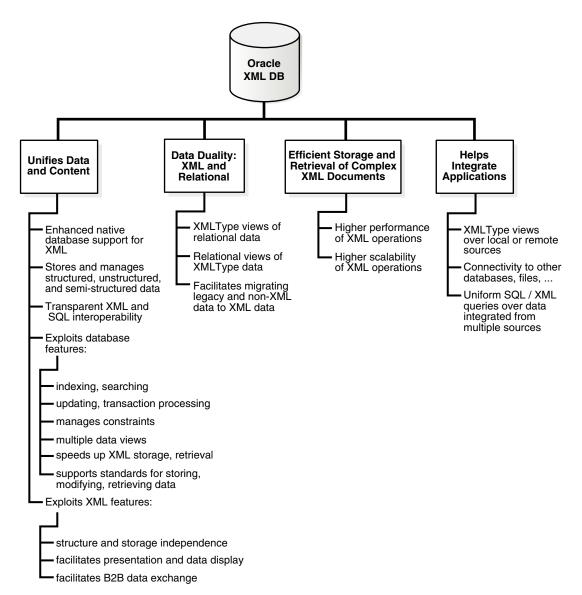
Oracle XML DB Benefits

Oracle XML DB supports all major XML, SQL, Java, and Internet standards. It provides high performance and scalability for XML operations. It brings database features such as transaction control, data integrity, replication, reliability, availability, security, and scalability to the world of XML.



Figure 1-1 presents an overview of the standards supported by Oracle XML DB.





Data and Content Unified

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.

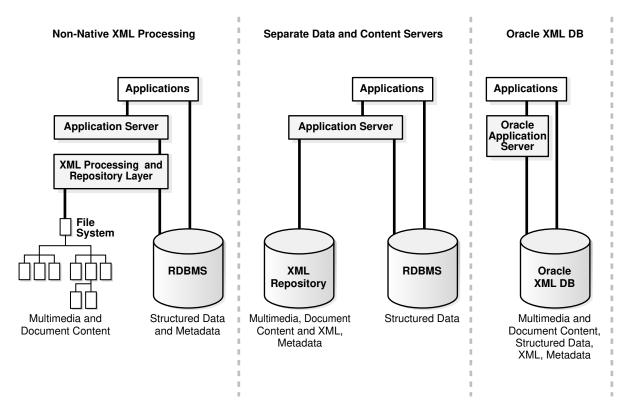
- Data Duality: XML and Relational Oracle XML DB presents a symmetric, dual view of data: as XML and as relational.
- Efficient Storage and Retrieval of Complex XML Documents Oracle XML DB provides high performance and scalability for XML operations, letting you manage the storage and retrieval of complex, large, or many XML documents.

Data and Content Unified

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.

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 1-2 Unifying Data and Content: Some Common XML Architectures



Organizations often manage their structured data and unstructured data differently:

- Unstructured data, stored in tables, makes document access transparent and table access complex.
- Structured data, often stored in binary large objects (such as in BLOB instances), makes access more complex and table access transparent.
- Database Capabilities for Working with XML

Oracle Database provides database capabilities for working with XML: indexing and search; updates and transaction processing; managing relationships using constraints; multiple data views; high performance and scalability. It supports XML Schema, XQuery, XPath, and DOM.



• Advantages of Storing Data as XML in the Database Storing data as XML in the database provides these advantages: storage independence, ease of presentation, and ease of interchange.

Database Capabilities for Working with XML

Oracle Database provides database capabilities for working with XML: indexing and search; updates and transaction processing; managing relationships using constraints; multiple data views; high performance and scalability. It supports XML Schema, XQuery, XPath, and DOM.

• Indexing and search – Just as your database data can be more or less structured, so can your queries. One query can look for all product definitions created between March and April 2014. Another query can look for products whose descriptions contain the words "wireless" and "router" but not the term "wireless router".

A query such as the former targets structured data, and it is typically supported by a B-tree index on a date column. A query such as the latter targets unstructured data, and for Oracle Database it is typically supported by an Oracle Text (full-text) index. Applications can of course combine structured and unstructured queries, and targeted data can be a mix of structured and unstructured data.

For XML data the situation is similar. Oracle XML DB provides indexing features that let you target the gamut of XML possibilities, from data and queries that are highly structured to those that are highly unstructured.

See Also:

- Query and Update of XML Data
- Generation of XML Data from Relational Data
- Indexing XML Data for Full-Text Queries (pre-23ai)
- Oracle Text Application Developer's Guide
- Oracle Text Reference
- 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:

Query and Update of XML Data

 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:

- XML Schema Storage and Query: Basic
- The purchase-order examples in Query and Update of XML Data
- 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.



 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:

- Getting Started with Oracle XML DB
- Overview of How To Use 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:

- XQuery and Oracle XML DB
- Java DOM API for XMLType
- Repository Access Using RESOURCE_VIEW and PATH_VIEW
- PL/SQL Access to Oracle XML DB Repository

Advantages of Storing Data as XML in the Database

Storing data as XML in the database provides these advantages: storage independence, ease of presentation, and ease of interchange.

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.



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.



- Transformation and Validation of XMLType Data.
- Generation of XML Data from Relational Data.
- XMLType Views.
- 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.



- Transformation and Validation of XMLType Data
- XMLType Views

Data Duality: XML and Relational

Oracle XML DB presents a symmetric, dual view of data: as XML and as relational.

A key feature of Oracle XML DB is that it lets you work with XML data as if it were relational data and relational data as if it were XML data. You can leverage the power of the relational model when working with XML content, and you can leverage the flexibility of XML when working with relational content. You can use the most appropriate tools for different aspects of a particular business problem.

This 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 XQuery, the DOM, or XSL transformation. Access and processing techniques are independent of the underlying storage method.

These features can provide simple solutions to common business problems:



- You can generate XML data directly from a SQL query. You can 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 and spatial data operations on XML content.

Oracle OLAP deprecated in Oracle Database 23ai.

 Use XMLType Views If Your Data Is Not XML XMLType views provide a way for you to wrap existing relational or object-relational data in XML format.

Use XMLType Views If Your Data Is Not XML

XMLType views provide a way for you to wrap existing relational or object-relational data in XML format.

This can be especially useful if 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 or the stored data.

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.

Related Topics

XMLType Views

You can create XMLType views over relational and object-relational data.

Efficient Storage and Retrieval of Complex XML Documents

Oracle XML DB provides high performance and scalability for XML operations, letting you manage the storage and retrieval of complex, large, or many XML documents.

These are the major performance features of Oracle XML DB:

- Native XMLType. Abstract data type XMLType has two storage models, each optimized to work efficiently for a particular set of use cases. See Query and Update of XML Data and Choice of XMLType Storage and Indexing
- Optimized processing of XQuery, XPath, and XSLT. See Performance Tuning for XQuery and Transformation and Validation of XMLType Data.
- Indexing XML data for structured or full-text search. See Indexes for XMLType Data.
- A lazily evaluated virtual DOM. See PL/SQL DOM API for XMLType (DBMS_XMLDOM).
- A hierarchical index over Oracle XML DB Repository. See Performance Guidelines for Oracle XML DB Repository Operations.
- Parallelism and Oracle Exadata Smart Scan. Query and update of XML data can be carried out in parallel. Oracle Exadata Smart Scan is enabled automatically for XML data.



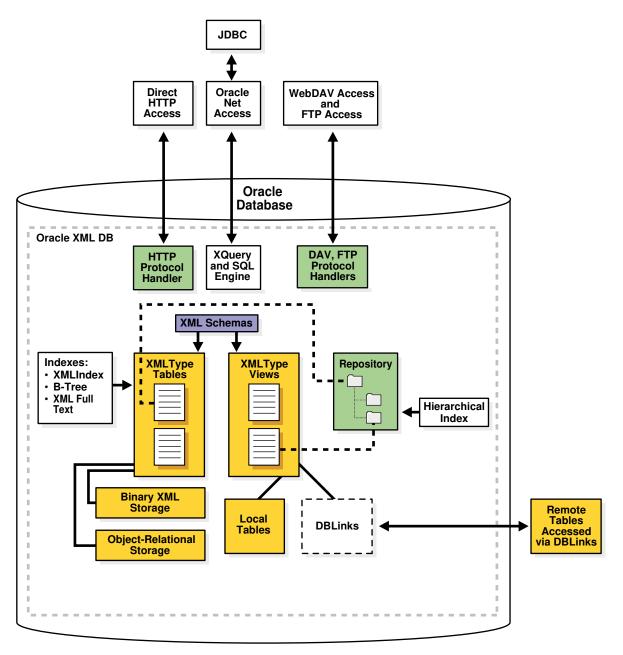
Oracle XML DB Architecture

Oracle XML DB gives you protocol and programmatic access to XML data in the form of local and remote XMLType tables and views. It provides a WebDAV repository with resource versioning and access control.

Figure 1-3 shows 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.
- Support for XQuery, including XQuery Update and XQuery Full Text.
- 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)





Oracle XML DB Features

Oracle XML DB 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 a 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.

XMLType Data Type

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

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.

XML Schema Support in Oracle XML DB

Support for the World Wide Web Consortium (W3C) XML Schema Recommendation is a key feature in Oracle XML DB.

DTD Support in Oracle XML DB

An XML schema is in general a much more powerful way to define XML document structure than is a DTD. You can nevertheless use DTDs to some extent with Oracle XML DB.

- Static Data Dictionary Views Related to XML Several static data dictionary views are related to XML.
- SQL/XML Standard Functions Oracle XML DB provides the SQL functions that are defined in the SQL/XML standard.
- Programmatic Access to Oracle XML DB (Java, PL/SQL, and C) All Oracle XML DB functionality is accessible from C, PL/SQL, and Java.
- Oracle XML DB Repository: Overview Oracle XML DB Repository is a component of Oracle Database that lets you handle XML data using a file/folder/URL metaphor.

XMLType Data Type

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

XMLType is an abstract native SQL data type for XML data. It provides PL/SQL and Java constructors for creating an XMLType instance from a VARCHAR2, CLOB, BLOB, or BFILE instance. And it provides PL/SQL methods for various XML operations.

You can use XMLType as you would any other SQL data type. For example, you can create an XMLType table or view, or an XMLType column in a relational table.



You can use XMLType in PL/SQL stored procedures for parameters, return values, and variables.

You can also manipulate XMLType data using application programming interfaces (APIs) for the Java and C languages, including Java Database Connectivity (JDBC), and Oracle Data Provider for .NET (ODP.NET).

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.

You can constrain XMLType tables or columns to conform to an XML schema, in which case the database ensures that only XML data that validates against the XML schema is stored in the column or table. invalid documents are excluded.

🖍 See Also:

- Oracle Database Object-Relational Developer's Guide for information about Oracle Database object types and object-relational storage
- Oracle Database PL/SQL Packages and Types Reference for information about XMLType constructors and methods

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.

XMLType tables and columns can be stored in in the following ways:

 Transportable Binary XML (TBX) storage (default, recommended) – XMLType data is stored in a self-contained binary format. Oracle Database 23ai introduces Transportable Binary XML (TBX) which does not store the metadata used to encode or decode XML data in a central table. This simplifies the XML data storage and makes it easier to transport. Oracle supports sharding and XML search index for Transportable Binary XML.

Transportable Binary XML can only be stored using SecureFile LOB. If a store as basicfile clause is specified for TBX, then an error is raised. The only exceptions are SYS and XDB users, for which the basicfile clause may be used.

 Binary XML (Non-Transportable) storage – XMLType data is stored in a post-parse, binary format designed specifically for XML data. Binary XML is compact, post-parse, XML schema-aware XML data and stores the metadata used to encode or decode XML data in a central table.

For non-transportable binary XML storages, SecureFiles is the default storage option. However, if either of the following is true then it is not possible to use SecureFiles LOB storage. In that case, BasicFiles 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 SecureFiles LOB storage. For example, see parameter DB SECUREFILE.



 Object-relational storage – XMLType data is stored as a set of objects. This is also referred to as structured storage and object-based persistence.

With the use of appropriate indexes, both transportable and non-transportable binary XML storages offer good performance for most use cases.

You can change XMLType storage from one model to another using database import/export or online redefinition. Your application code need not change. You can change XML storage options when tuning your application.

Note:

- Starting with Oracle Database 12c Release 1 (12.1.0.1), the unstructured (CLOB) storage model for XMLType is *deprecated*. Use binary XML storages instead.
- Starting with Oracle Database 23ai, Transportable Binary XML is the default and recommended storage type for all XMLType data
- While TBX is the recommended storage type, certain applications may still rely
 on the old CSX or not-transportable binary xml. Use parameter XML_PARAMS to
 switch the default type from TBX to CSX.

Related Topics

- Choice of XMLType Storage and Indexing Important design choices for your application include what XMLType storage model to use and which indexing approaches to use.
- Export and Import of Oracle XML DB Data You can use Oracle Data Pump to export and import XMLType tables for use with Oracle XML DB.

See Also:

- Oracle Database Reference for more information about XML PARAMS.
- 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 SecureFiles and BasicFiles
- Oracle Database Administrator's Guide for information about automatic segment
 space management
- Oracle Database Reference for information about parameter DB_SECUREFILE

XML Schema Support in Oracle XML DB

Support for the World Wide 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/soap12-part0/.

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.

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 one of the Oracle XML DB XMLType storage models: compact schema-aware binary XML storage, transportable binary XML storage, or object-relational storage. Non-schema-based XML data can be stored only using binary XML storages.

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.

Related Topics

 Choice of XMLType Storage and Indexing Important design choices for your application include what XMLType storage model to use and which indexing approaches to use.

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.

XML Schema Storage and Query: Basic

XML Schema is a standard for describing the content and structure of XML documents. You can register, update, and delete an XML schema used with Oracle XML DB. You can define storage structures to use for your XML schema-based data and map XML Schema data types to SQL data types.

DTD Support in Oracle XML DB

An XML schema is in general a much more powerful way to define XML document structure than is a DTD. You can nevertheless use DTDs to some extent with Oracle XML DB.

Like an XML schema, 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. They can be associated with an XML document by using DTD element DOCTYPE or by using an external file through a DOCTYPE reference.

Oracle XML DB uses XML Schema, not DTDs, to define structured mappings to XMLType storage, but XML processors can still access and interpret your DTDs.

Note:

You can use a DTD to obtain the XML entities defined in it. The entities are the only information used from the DTD. The structural and type information in the DTD is not used by Oracle XML DB.

Inline DTD Definitions

When an XML instance document has an inline DTD definition, that definition is used during document parsing. Any DTD validations and entity declaration handling are 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 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.

Related Topics

Access to Oracle XML DB Repository Data

There are several ways to access and manipulate data in Oracle XML DB Repository, including using standard protocols such as FTP and HTTP(S)/WebDAV; Oracle XML DB resource Application Program Interfaces (APIs); and the repository views RESOURCE_VIEW and PATH VIEW.



Static Data Dictionary Views Related to XML

Several static data dictionary views are related to XML.

Table 1-1 lists these views. Information about a given view can be obtained by using SQL command DESCRIBE:

DESCRIBE USER XML SCHEMAS

Table 1-1 Static Data Dictionary 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

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 update XML content as part of normal SQL operations. In this book, these are called SQL/XML query and update 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

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 that take advantage of Oracle XML DB 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 Repository: Overview

Oracle XML DB Repository is a component of Oracle Database that lets you handle XML data using a file/folder/URL metaphor.

Oracle XML DB Repository contains **resources**, which can be either **folders** (directories, containers) or files.

A resource, whether folder or file, 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:

- SQL Using views RESOURCE VIEW and PATH VIEW
- Standard protocols FTP, HTTP(S), and WebDAV
- PL/SQL Using PL/SQL package DBMS XDB REPOS
- Java Using the Oracle XML DB resource API for Java

Besides providing APIs for accessing and manipulating repository data, Oracle XML DB provides APIs for the following repository services, which are based on IETF WebDAV:

- Versioning Using PL/SQL package DBMS XDB VERSION
- ACL Security Using access control lists (ACLs)
- Foldering Using repository path names

Figure 1-4 illustrates the architecture of Oracle XML DB Repository.



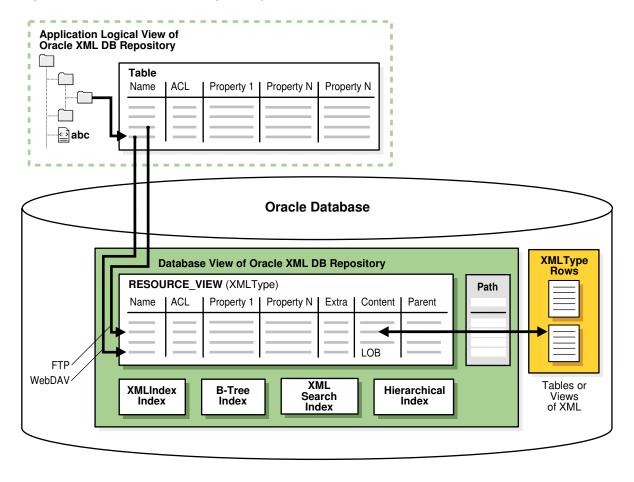


Figure 1-4 Oracle XML DB Repository Architecture

Related Topics

Oracle XML DB Repository

Oracle XML DB Repository lets you version your data, implement and manage security, and use APIs to access and manipulate repository resources.

Standards Supported by Oracle XML DB

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

These include the following:

- 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.
- ANSI/ISO/IEC 9075-14:2011, Information technology—Database languages—SQL—Part 14: XML-Related Specifications (SQL/XML).
- 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.



- Java Database Connectivity (JDBC) API. Provides Java access to XML data.
- W3C XSL 1.0 Recommendation. You can transform XML documents at the server using XSLT.
- 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 Oracle Fusion Middleware, using WSDL and UDDI. You can use Oracle Database 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 for more information about the SQL/XML functions
- Oracle Database SQL Language Reference for information about Oracle support for the SQL/XML standard
- Use of XLink and XInclude with Oracle XML DB for more information about XLink and XPointer support
- Repository Access Using Protocols for more information about protocol support
- Guidelines for Oracle XML DB Applications in Java for information about using the Java servlet
- XML Data Exchange Using Oracle Database Advanced Queuing and Oracle Database Advanced Queuing User's Guide for information about using SOAP

Oracle XML DB Technical Support

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

Oracle XML DB Examples

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

Related Topics

Oracle-Supplied XML Schemas and Examples

Full listings are provided here for the Oracle XML DB-supplied XML schemas, purchaseorder XML schemas and an XSLT stylesheet used in various examples, and C-language



(OCI) examples for loading XML content into Oracle XML DB and initializing and terminating an XML context.

Oracle XML DB Case Studies and Demonstrations on OTN

Visit Oracle Technology Network (OTN) to view Oracle XML DB examples, white papers, case studies, and demonstrations.

Oracle XML DB is presented on OTN at Oracle XML DB on OTN.

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

Detailed Oracle XML DB case studies available on OTN 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 Overview of How To Use Oracle XML DB.
- 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 Webbased application.



2 Getting Started with Oracle XML DB

Some preliminary design criteria are presented for consideration when planning your Oracle XML DB solution.

- Oracle XML DB Installation
 Oracle XML DB is installed automatically if Database Configuration Assistant (DBCA) is used to build Oracle Database using the general-purpose template.
- Oracle XML DB Use Cases
 Oracle XML DB is suited for any application where some or all of the data processed is
 represented using XML.
- Application Design Considerations for Oracle XML DB
 When planning an Oracle XML DB application it can be worthwhile to consider some preliminary design criteria.

Oracle XML DB Installation

Oracle XML DB is installed automatically if Database Configuration Assistant (DBCA) is used to build Oracle Database using the general-purpose template.

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:

- Administration of Oracle XML DB for information about installing Oracle XML DB manually
- Oracle Database Security Guide

Oracle XML DB Use Cases

Oracle XML DB is suited for any application where some or all of the data processed 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 at least one 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

Application Design Considerations for Oracle XML DB

When planning an Oracle XML DB application it can be worthwhile to consider some preliminary design criteria.

These include the following:

- The ways that you intend to store your XML data
- The structure of your XML data
- The languages used to implement your application
- The ways you intend to process your XML data

However, in general Oracle recommends that you start with the following Oracle XML DB features. For most use cases they are all that you need to consider.

- Storage model compact schema-aware binary XML or transportable binary XML
- Indexing XML search index, XMLIndex with structured component
- Database language SQL, with SQL/XML functions
- XML languages XQuery and XSLT
- Client APIs OCI, thin JDBC, SQL .NET



- XML Data Storage There are several ways to store XML data in Oracle Database.
- The Structure of Your XML Data How structured your XML data is, and whether it is based on an XML schema, can influence how you store it.
- Languages Used to Implement Your Application You can program your Oracle XML DB applications in Java (JDBC, Java Servlets) or PL/ SQL.
- XML Processing Options Oracle XML DB offers a full range of XML processing options.
- Oracle XML DB Repository Access
 Design considerations for applications that use Oracle XML DB Repository include access
 method, security needs, and whether you need versioning.
- Oracle XML DB Cooperates with Other Database Options and Features Oracle XML DB is an integrated part of Oracle Database, and works well with other database options and features.

Related Topics

- Choice of XMLType Storage and Indexing Important design choices for your application include what XMLType storage model to use and which indexing approaches to use.
- SQL/XML Standard Functions Oracle XML DB provides the SQL functions that are defined in the SQL/XML standard.
- XQuery and Oracle XML DB The XQuery language is one of the main ways that you interact with XML data in Oracle XML DB. Support for the language includes SQL*Plus commandxQUERY and SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast.
- Transformation and Validation of XMLType Data

There are several Oracle SQL functions and XMLType APIs for transforming XMLType data using XSLT stylesheets and for validating XMLType instances against an XML schema.

- C DOM API for XMLType The C DOM API for XMLType lets you operate on XMLType instances using a DOM in C.
- Java DOM API for XMLType The Java DOM API for XMLType lets you operate on XMLType instances using a DOM. You can use it to manipulate XML data in Java, including fetching it through Java Database Connectivity (JDBC).
- Oracle XML DB and Oracle Data Provider for .NET
 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.

XML Data Storage

There are several ways to store XML data in Oracle Database.

Storage of XMLType tables and views is outlined in Figure 2-1.



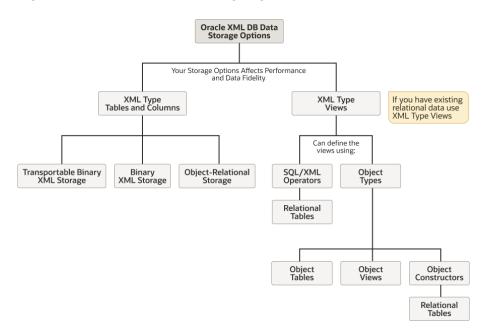


Figure 2-1 Oracle XML DB Storage Options for XML Data

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 Generation of XML Data from Relational Data and XQuery and 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 how frequently you query or update your data and what APIs your application uses.

Related Topics

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.

The Structure of Your XML Data

How structured your XML data is, and whether it is based on an XML schema, can influence how you store it.

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 you can store it as a file in an Oracle XML DB Repository folder. You cannot store it object-relationally.

If your XML data is XML Schema-based then you must store it as binary XML or objectrelationally.



Related Topics

 Choice of XMLType Storage and Indexing Important design choices for your application include what XMLType storage model to use and which indexing approaches to use.

Languages Used to Implement Your Application

You can program your Oracle XML DB applications in Java (JDBC, Java Servlets) or PL/SQL.

Related Topics

- Java DOM API for XMLType The Java DOM API for XMLType lets you operate on XMLType instances using a DOM. You can use it to manipulate XML data in Java, including fetching it through Java Database Connectivity (JDBC).
- Guidelines for Oracle XML DB Applications in Java
 Design guidelines are presented for writing Oracle XML DB applications in Java. This
 includes guidelines for writing and configuring Java servlets for Oracle XML DB.
- PL/SQL APIs for XMLType There are several PL/SQL packages that provide APIs for XMLType.
- PL/SQL Access to Oracle XML DB Repository
 PL/SQL packages DBMS_XDB_CONFIG and DBMS_XDB_REPOS together provide the Oracle
 XML DB resource application program interface (API) for PL/SQL. You use the former to
 configure Oracle XML DB and its repository. You use the latter to perform other, non configuration operations on the repository.

XML Processing Options

Oracle XML DB offers a full range of XML processing options.

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

- XML Generation and XMLType views. Whether you need to generate (or regenerate) XML data. See Generation of XML Data from Relational Data.
- Whether your application is data-centric or document-centric, or both. See Overview of How To Use Oracle XML DB.
- DOM fidelity, document fidelity. XMLType storage, whether object-relational, compact schema-aware binary XML, or transportable binary XML, preserves DOM fidelity. That is, A DOM created from an XML document stored as XMLType is identical to a DOM created from the original document. However, there could be differences in insignificant whitespace. See DOM Fidelity, SYS_XDBPD\$ and DOM Fidelity for Object-Relational Storage, and PL/SQL APIs for XMLType.

If you need to preserve *document fidelity* (insignificant whitespace) in addition to DOM fidelity, then store two copies of your original document: one as an XMLType instance for database use and XML processing, the other as a CLOB instance to provide document fidelity.

- XPath searching. You can use XPath syntax embedded in a SQL statement to query XML content in the database. See Query and Update of XML Data, Access to Oracle XML DB Repository Data, and Repository Access Using RESOURCE_VIEW and PATH_VIEW.
- How often XML documents are accessed, updated, and manipulated. See Query and Update of XML Data.



- Whether 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. See Updating XML Data.
- Which kinds of indexing best suit your application and data. See Indexes for XMLType Data.
- XSLT. Whether you need to transform the XML data to HTML, WML, or other languages, and, if so, how your application does this. While storing XML documents in Oracle XML DB, you can optionally ensure that their structure complies with (validates against) specific XML schemas. See Transformation and Validation of XMLType Data.

Oracle XML DB Repository Access

Design considerations for applications that use Oracle XML DB Repository include access method, security needs, and whether 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 Repository Access Using RESOURCE_VIEW and PATH_VIEW.
 - PL/SQL access using package DBMS_XDB or packages DBMS_XDB_ADMIN,
 DBMS_XDB_CONFIG and DBMS_XDB_REPOS. See PL/SQL Access to Oracle XML DB
 Repository .
 - Protocol-based access using HTTP(S)/WebDAV or FTP, most suited to contentoriented applications. See Repository Access 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 PL/SQL APIs for XMLType.

These options for accessing repository data are also discussed in Access to Oracle XML DB Repository Data.

You can also consider the following access criteria:

- What levels of security you need. See Repository Access Control.
- Whether you need to version the data. See Resource Versions.

Oracle XML DB Cooperates with Other Database Options and Features

Oracle XML DB is an integrated part of Oracle Database, and works well with other database options and features.

- Oracle Database Advanced Queuing (AQ) merge XML payloads. See XML Data Exchange Using Oracle Database Advanced Queuing and Oracle Database Advanced Queuing User's Guide
- Oracle GoldenGate and Oracle Active Data Guard replicate and safeguard XML data, or perform a rolling upgrade. See Oracle GoldenGate and Oracle Data Guard Concepts and Administration



- Oracle Exadata Storage Server Software high-performance, scalable, and highly available use of XML data. See Oracle Exadata Storage Server Software User's Guide.
- Oracle Real Application Clusters (Oracle RAC) Use XML data with clusters of database instances. See Oracle Real Application Clusters Administration and Deployment Guide
- Oracle Multitenant option Use XML data with a multitenant architecture, where each pluggable database has its own Oracle XML DB Repository. See Oracle Multitenant Administrator's Guide
- Compression and Encryption You can compress or encrypt binary XML data that uses SecureFiles LOB storage. For XML data stored object-relationally, you can compress or encrypt XML elements and attributes individually.
- Parallel Execution Execution of the following operations can be carried out in parallel:
 - A query of XMLType data
 - DML for XMLType data stored as binary XML using SecureFiles LOBs
 - A direct load for an XMLType table on which an Oracle Text CONTEXT index is defined

See Also:

Oracle Database Concepts



3 Overview of How To Use Oracle XML DB

An overview of the various ways of using Oracle XML DB is presented.

This overview illustrates how to do the following: create and partition XMLType tables and columns; enforce data integrity, load, query, and update database XML content; and generate XML data from relational data. It also explains how Oracle XML DB determines which character sets are used for XML documents.

Purchase Order Documents Illustrate Key XML Schema Features

Many of the examples presented in this chapter illustrate techniques for accessing and managing XML content in purchase-order documents. Purchase orders are highly structured documents, but you can also use the techniques shown here to work with XML documents that have little structure.

The purchase-order documents used for the examples here conform to a purchase-order XML schema that 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

Note:

Binary XML feature description in this chapter is applicable to both compact schemaaware binary XML (CSX) and transportable binary XML (TBX), unless indicated otherwise.

- Creating XMLType Tables and Columns
 Creating a table or column of XMLType is straightforward because it is an abstract data type.
- Creating XMLType Columns in Shared and Duplicated Tables
- Creating Virtual Columns on XMLType Data Stored as Binary XML
 You can create virtual columns only for XMLType data that is stored as binary XML. Such columns are useful for partitioning or constraining the data.
- Partitioning Tables That Contain XMLType Data Stored as Binary XML You can partition a table that contains XMLType data stored as binary XML.
- Enforcing XML Data Integrity Using the Database You can combine the power of SQL and XML with the ability of the database to enforce rules.



- Loading XML Content into Oracle XML DB There are several ways to load XML content into Oracle XML DB.
- Querying XML Content Stored in Oracle XML DB There are many ways to query XML content in Oracle XML DB and retrieve it.
- Updating XML Content Stored in Oracle XML DB You can update XML content, replacing either the entire contents of a document or only particular parts of a document.
- Generating XML Data from Relational Data You can use Oracle XML DB to generate XML data from relational data.
- Character Sets of XML Documents
 There are a few ways in which Oracle XML DB determines which character sets are used
 for XML documents
- Migrating XMLType Data to Transportable Binary XML (TBX)

See Also:

- Application Design Considerations for Oracle XML DB for recommended Oracle XML DB features for most uses
- XMLType APIs, XML Schema and Object-Relational XMLType, and Oracle XML DB Repository for information about more advanced Oracle XML DB features
- Purchase-Order XML Schemas for the purchase-order XML schemas used for examples in this chapter

Creating XMLType Tables and Columns

Creating a table or column of XMLType is straightforward because it is an abstract data type.

The basic CREATE TABLE statement, specifying no storage options and no XML schema, stores XMLType data as binary XML before 23ai.¹ Starting from 23ai, when the database compatibility is set to 23.0.0.0 or above, XMLType data is stored as Transportable Binary XML by default.

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);

This is equivalent to specifying transportable binary xml explicitly:

CREATE TABLE mytable1 (key_column VARCHAR2(10) PRIMARY KEY, xml_column XMLType) XMLType xml column store as transportable binary xml;

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 object-relational storage is used.

Example 3-2 Creating a Table of XMLType

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

Or,

CREATE TABLE mytable2 OF XMLType XMLType store as transportable binary xml;

```
Note:
TBX is highly recommended. However, if for some reason the application still expects
the old not transportable binary XML, the following workarounds may be used.
    CREATE TABLE mytable1 (key column VARCHAR2(10) PRIMARY KEY,
    xml column XMLType)
        XMLType xml column store as not transportable binary xml;
    Or
    CREATE TABLE mytable1 (key column VARCHAR2(10) PRIMARY KEY,
    xml column XMLType)
        XMLType xml column store as binary xml;
    If the usage of non-ASSM tablespace cannot be avoided, specify basicfile
    explicitly:
    CREATE TABLE mytable1 (key column VARCHAR2(10) PRIMARY KEY,
    xml column XMLType)
        XMLType xml column store as basicfile binary xml;
    Use parameter XML PARAMS for lesser modifications:
    alter session SET XML PARAMS = "xml default storage tbx=false";
    See Oracle Database Reference for more information about XML PARAMS.
```

Related Topics

Creation of XMLType Tables and Columns Based on XML Schemas

You can create XMLType tables and columns that are constrained to a global element defined by an XML schema. After an XMLType column has been constrained to a particular element and a particular schema, it can only contain documents that are compliant with the schema definition of that element.

Creating XMLType Columns in Shared and Duplicated Tables

Transportable Binary XML (TBX) storage is a variant built on top of Non-Transportable Binary XML with inline token information. TBX is scalable and supports sharding, TBX data replication, and search index. User can create sharded tables with TBX columns, but not

sharded TBX tables. User can also create virtual TBX columns in sharded tables, but they cannot be a sharded key.

Since sharding is required for tables to be transported across different remote databases, the only viable storage type for XMLType is TBX. This is the only storage type allowed for sharding architectures and therefore is the defaulted storage for sharded and duplicated tables.

If other storage option is specified for sharded or duplicated tables, the following error is thrown:

ORA-19072: Only STORE AS TRANSPORTABLE BINARY XML is supported with XMLType on sharded tables.

Example 3-3 Creating a Sharded Table in a System-Sharded Environment

CREATE SHARDED TABLE TAB (ID NUMBER, X XMLTYPE, CONSTRAINT ST_PK PRIMARY KEY(ID)) PARTITION BY CONSISTENT HASH (ID) PARTITIONS AUTO TABLESPACE SET TBSSET1 XMLTYPE X STORE AS TRANSPORTABLE BINARY XML;

Example 3-4 Creating a Duplicated Table in a Sharded Environment

CREATE DUPLICATED TABLE DUPTAB (ID NUMBER, X XMLTYPE) XMLTYPE X STORE AS TRANSPORTABLE BINARY XML;

Creating Virtual Columns on XMLType Data Stored as Binary XML

You can create virtual columns only for XMLType data that is stored as binary XML. Such columns are useful for partitioning or constraining the data.

You create virtual columns for XML data the same way you create them for other data types, but you use a slightly different syntax. (In particular, you cannot specify constraints in association with the column definition.)

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 3-5 and Example 3-6. The XQuery expression argument to function XMLQuery must be a simple XPath expression that uses only the child and attribute axes.

Example 3-5 creates XMLType table po_binaryxml, stored as binary XML. It creates virtual column date col, which represents the XML data in attribute /PurchaseOrder/@orderDate.

Example 3-6 creates relational table rel_tab, which has two columns: VARCHAR2 column key col for the primary key, and XMLType column xml col for the XML data.

Because XMLType is an abstract data type, if you create virtual columns on an XMLType table or column 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. Virtual columns made using XMLType columns are shown in the DESCRIBE operation.



Note:

If you use a virtual column for interval partitioning then it *must* have data type NUMBER or DATE, otherwise an error is raised. Use SQL/XML functions XMLCast and XMLQuery to cast to the proper data type.

See Also:

Oracle Database SQL Language Reference for information about creating tables with virtual columns

Example 3-5 Creating a Virtual Column for an XML Attribute in an XMLType Table

```
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)));
```

Example 3-6 Creating a Virtual Column for an XML Attribute in an XMLType Column

For this scenario, if you describe reltab you will be able to see date col.

Related Topics

- Partitioning Tables That Contain XMLType Data Stored as Binary XML You can partition a table that contains XMLType data stored as binary XML.
- Enforcing Referential Integrity Using SQL Constraints
 You can use SQL constraints and database triggers to ensure data-integrity properties such as uniqueness and foreign-key relations.

Partitioning Tables That Contain XMLType Data Stored as Binary XML

You can partition a table that contains XMLType data stored as binary XML.

There are two possibilities:

- The table is relational, with an XMLType column and a non-XMLType column.
- The table is of data type XMLType.



In the case of an XMLType column, you use the non-XMLType column as the partitioning key. This is illustrated in Example 3-7.

This case presents nothing new or specific with respect to XML data. The fact that one of the columns contains XMLType data is irrelevant. Things are different for the other case: partitioning an XMLType table.

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

Therefore, to partition binary 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.

The technique is as follows:

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

This is illustrated in Example 3-8: virtual column date_col targets the orderDate attribute of element PurchaseOrder in a purchase-order document. This column is used as the partitioning key.

For best performance using a partitioned table containing XML data, Oracle recommends that you use an XMLType column rather than an XMLType table, and you therefore partition using a non-XMLType column.

Note:

- You can partition an XMLType table using a virtual column only if the storage model is compact schema-aware binary XML or transportable binary XML. Range, hash, and list partitioning are supported.
- Partitioning of XMLType tables stored as XML 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.
- If a relational table has an XMLType *column*, you cannot partition the table using that column to define virtual columns of XML data.

Example 3-7 Partitioning a Relational Table That Has an XMLType Column



Example 3-8 Partitioning an XMLType Table

```
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));
```

Related Topics

- XMLIndex Partitioning and Parallelism
 If you partition an XMLType table, or a table with an XMLType column, using range, list, or
 hash partitioning, you can also create an XMLIndex index on the table. You can optionally
 ensure that index creation and maintenance are carried out in parallel.
- Overview of Partitioning XMLType Tables and Columns Stored Object-Relationally When you partition an object-relational XMLType table or a table with an XMLType column that is stored object-relationally and you use list, range, or hash partitioning, any ordered collection tables (OCTs) or out-of-line tables within the data are automatically partitioned accordingly, by default.

Enforcing XML Data Integrity Using the Database

You can combine the power of SQL and XML with the ability of the database to enforce rules.

You can use SQL to supplement the functionality provided by XML schema. 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 then 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 3-9 illustrates this.

Such an error occurs only 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.

```
Example 3-9 Error From Attempting to Insert an Incorrect XML Document
```

Enforcing Referential Integrity Using SQL Constraints

You can use SQL constraints and database triggers to ensure data-integrity properties such as uniqueness and foreign-key relations.

Related Topics

Partial and Full XML Schema Validation
 When you insert XML Schema-based documents into the database they can be validated partially or fully.

Enforcing Referential Integrity Using SQL Constraints

You can use SQL constraints and database triggers to ensure data-integrity properties such as uniqueness and foreign-key relations.

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 as those you use to enforce integrity on relational data. Simple rules, such as uniqueness and foreign-key relationships, can be enforced by specifying SQL 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.

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



Therefore, to constrain binary 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 that you need.

The technique is as follows:

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

The binary XML data can be in an XMLType table or an XMLType column of a relational table. In the former case, you can include creation of the constraint as part of the CREATE TABLE statement, if you like. For the latter case, you must create the constraint using an ALTER TABLE statement, after the relational table has been created.

See also: Oracle Database Error Messages Reference

Example 3-10 Constraining a Binary XML Table Using a Virtual Column

This example illustrates the technique for an XMLType table. It defines virtual column c_xtabref using the Reference element in a purchase-order document. It defines uniqueness constraint reference_is_unique on that column, which ensures that the value of node /PurchaseOrder/ Reference/text() is unique across all documents that are stored in the table. It fills the table with the data from OE.purchaseorder. It then tries to insert a duplicate document, DuplicateReference.xml, which violates the uniqueness constraint, raising an error.

```
CREATE TABLE po_binaryxml OF XMLType
(CONSTRAINT reference_is_unique UNIQUE (c_xtabref))
XMLTYPE STORE AS BINARY XML
VIRTUAL COLUMNS
(c_xtabref 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.
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 3-11 Constraining a Binary XML Column Using a Virtual Column: Uniqueness

This example illustrates the technique for an XMLType column of a relational table. It defines virtual column c_xcolref and uniqueness constraint fk_ref, which references the uniqueness constraint defined in Example 3-10. As in Example 3-10, this ensures that the value of node / PurchaseOrder/Reference/text() is unique across all documents that are stored in XMLType column po binxml col.

The example fills the XMLType column with the same data from OE.purchaseorder. It then tries to insert duplicate document, DuplicateReference.xml, which violates the uniqueness constraint, raising an error.

ORA-00001: unique constraint (OE.REFERENCE IS UNIQUE) violated

Example 3-12 Constraining a Binary XML Column Using a Virtual Column: Foreign Key

This example is similar to Example 3-11, but it uses a foreign-key constraint, fk_ref, which references the column with the uniqueness constraint defined in Example 3-10. Insertion of the document in file DuplicateReference.xml succeeds here, since that document is in (virtual) column c_tabref of table po_binaryxml. Insertion of a document that does not match any document in table po_binaryxml.

```
CREATE TABLE po_reltab (po_binxml_col XMLType)

XMLTYPE po_binxml_col STORE AS BINARY XML

VIRTUAL COLUMNS

(c_xcolref AS (XMLCast (XMLQuery('/PurchaseOrder/Reference'

PASSING po_binxml_col

RETURNING CONTENT)

AS VARCHAR2(32))));

ALTER TABLE po_reltab ADD CONSTRAINT fk_ref

FOREIGN KEY (c_xcolref)

REFERENCES po_binaryxml(c_xtabref);

INSERT INTO po_reltab

VALUES (XMLType(bfilename('XMLDIR', 'DuplicateReference.xml'),

nls_charset_id('AL32UTF8')));

INSERT INTO po reltab
```

VALUES (

```
'<PurchaseOrder><Reference>Not Compliant</Reference></PurchaseOrder>');
INSERT INTO po_reltab VALUES ('<PurchaseOrder><Reference>Not Compliant
</Reference></PurchaseOrder>')
*
ERROR at line 1:
```

ORA-02291: integrity constraint (OE.FK_REF) violated - parent key not found

Example 3-13 Enforcing Database Integrity When Loading XML Using FTP

Integrity rules defined using constraints and triggers are also enforced when XML schemabased XML content is loaded into Oracle XML DB Repository. This example 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. In this case, additional constraints, besides uniqueness, were also violated.

```
$ 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 SOL 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 using 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 a command line FTP tool, report the error returned by Oracle XML DB, while others, such as Microsoft Windows Explorer, report a generic error message.

Related Topics

- Specification of Relational Constraints on XMLType Tables and Columns
 For XMLType data stored object-relationally, you can specify typical relational constraints for
 elements and attributes that occur only once in an XML document.
- Creating Virtual Columns on XMLType Data Stored as Binary XML You can create virtual columns only for XMLType data that is stored as binary XML. Such columns are useful for partitioning or constraining the data.

Loading XML Content into Oracle XML DB

There are several ways to load XML content into Oracle XML DB.

- Loading XML Content Using SQL or PL/SQL
- Loading XML Content Using Java With a DOM you can use Java to load a SQLXML instance.
- Loading XML Content Using C With a DOM you can use C code to load an XMLType instance.
- 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.
- Loading Large XML Files Using SQL*Loader You can use SQL*Loader to load large amounts of XML data into Oracle Database.
- Loading XML Documents into the Repository Using DBMS_XDB_REPOS You can use PL/SQL package DBMS_XDB_REPOS to load XML documents into Oracle XML DB Repository. You can access repository documents (resources) using path-based rather than table-based techniques.
- Loading Documents into the Repository Using Protocols You can load documents, including XML documents, from a local file system into Oracle XML DB Repository using popular 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.



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.

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.

Example 3-15 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 PL/SQL Packages and Types Reference for a description of the XMLType constructors
- Oracle Database SQL Language Reference, under GRANT

Example 3-14 Creating a Database Directory

CREATE DIRECTORY xmldir AS path_to_folder_containing_XML_file;

Example 3-15 Inserting XML Content into an XMLType Table

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

Related Topics

- Query and Update of XML Data There are many ways for applications to query and update XML data that is in Oracle Database, both XML schema-based and non-schema-based.
- PL/SQL APIs for XMLType: References The PL/SQL Application Programming Interfaces (APIs) for XMLType are described.
- Considerations for Loading and Retrieving Large Documents with Collections Oracle XML DB configuration file xdbconfig.xml has parameters that control the amount of memory used by the loading operation: xdbcore-loadableunit-size and xdbcorexobmem-bound.

Loading XML Content Using Java

With a DOM you can use Java to load a SQLXML instance.

Example 3-16 shows how to load XML content into Oracle XML DB by first creating a SQLXML instance in Java, given a Document Object Model (DOM).



A simple bulk loader application is available at Oracle XML DB on OTN. It shows 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.

Example 3-16 Inserting Content into an XMLType Table Using Java

```
public void doInsert(Connection conn, Document doc)
throws Exception
{
   String query = "INSERT INTO purchaseorder VALUES (?)";
   SQLXML sx = conn.createSQLXML();
   DOMResult dom = sx.setResult(DOMResult.class);
   dom.setNode(doc);
   PreparedStatement statement = conn.prepareStatement(query);
   statement.setSQLXML(1, sx);
   statement.execute();
}
```

Loading XML Content Using C

With a DOM you can use C code to load an XMLType instance.

Example 3-17 shows how to insert XML content into an XMLType table using C code, by creating an XMLType instance given a DOM (see *Oracle XML Developer's Kit Programmer's Guide*). See Loading XML Data Using C (OCI) for a complete listing of this example.

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.

Example 3-17 Inserting Content into an XMLType Table Using C

```
. . .
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();
```

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:

- SAX Project for information about SAX
- Oracle XML DB on OTN, for an application example that loads large files using SAX



}

Loading Large XML Files Using SQL*Loader

You can 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 <i>default</i> 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.
<i>Disadvantage:</i> This loads data slower than with the direct load mode.	<i>Disadvantage:</i> 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.

Related Topics

 How to Load XML Data The main way to load XML data into Oracle XML DB is to use SQL*Loader.

Loading XML Documents into the Repository Using DBMS_XDB_REPOS

You can use PL/SQL package DBMS_XDB_REPOS to load XML documents into Oracle XML DB Repository. You can access repository documents (resources) 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_REPOS.createResource. Example 3-18 illustrates this.

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.

Example 3-18 Inserting XML Content into the Repository Using CREATERESOURCE



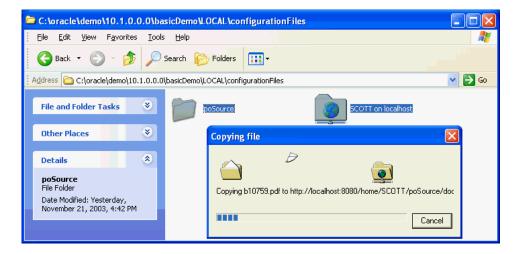
Loading Documents into the Repository Using Protocols

You can load documents, including XML documents, from a local file system into Oracle XML DB Repository using popular 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 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 stylesheets.

Querying XML Content Stored in Oracle XML DB

There are many ways to query XML content in Oracle XML DB and retrieve it.

Note:

For efficient query performance you typically need to create indexes. For information about indexing XML data, see Indexes for XMLType Data.

- PurchaseOrder XML Document Used in Examples An XML schema defines the purchase-order documents used in examples.
- 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.



- Accessing Fragments or Nodes of an XML Document Using XMLQUERY You can use SQL/XML function XMLQuery to extract the nodes that match an XQuery expression. The result is returned as an instance of XMLType.
- 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.
- Searching an XML Document Using XMLEXISTS, XMLCAST, and XMLQUERY You can use SQL/XML standard functions XMLExists, XMLCast, and XMLQuery in a SQL WHERE clause to limit query results.
- Performing SQL Operations on XMLType Fragments Using XMLTABLE You can use SQL/XML function XMLTable to perform SQL operations on a set of nodes that match an XQuery expression.

PurchaseOrder XML Document Used in Examples

An XML schema defines the purchase-order documents used in examples.

Examples presented here are based on the PurchaseOrder XML document shown in Example 3-19.

Example 3-19 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 Database 10g Release 1.)

In Example 3-20, 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 3-20 Retrieving an Entire XML Document Using OBJECT_VALUE

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-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>

</LineItem>

</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 XQuery expression. The result is returned as an instance of XMLType.

Example 3-21 illustrates this with several queries.

Example 3-21 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:

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.

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.

Related Topics

 Performing SQL Operations on XMLType Fragments Using XMLTABLE You can use SQL/XML function XMLTable to perform SQL operations on a set of nodes that match an XQuery expression.

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 XQuery expression passed to XMLQuery must uniquely identify a *single* text node or attribute value within the document – that is, a *leaf* node. Example 3-22 illustrates this using several queries.

See Also:

XQuery and Oracle XML DB for information on SQL/XML functions ${\tt XMLQuery}$ and ${\tt XMLCast}$

Example 3-22 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.

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(
```

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].

```
XMLCAST(XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[1]/DESCRIPTION'PASSINGOBJECT_VALUEAS"P"
```

A Night to Remember

1 row selected.

Searching an XML Document Using XMLEXISTS, XMLCAST, and XMLQUERY

You can use SQL/XML standard functions XMLExists, XMLCast, and XMLQuery in a SQL WHERE clause to limit query results.

SQL/XML standard function XMLExists evaluates whether or not a given document contains a node that matches a W3C XPath expression. It 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, functions XMLCast and XMLQuery let you limit query results to documents that satisfy some property. Example 3-23 illustrates the use of XMLExists, XMLCast, and XMLQuery to search for documents.



Example 3-24 performs a join based on the values of a node in an XML document and data in another, relational table.

See Also: XQuery and Oracle XML DB for information about SQL/XML functions XMLQuery, XMLExists, and XMLCast

Example 3-23 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 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.

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.

XMLCAST (XMLQUERY ('\$P/PURCHASEORDER

SBELL-20021009123336231PDT SBELL-20021009123336331PDT SKING-20021009123336321PDT ...

36 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.

SBELL-20021009123338304PDT SBELL-2002100912333505PDT SBELL-20021009123335771PDT SBELL-20021009123335280PDT SBELL-2002100912333763PDT SBELL-20021009123336362PDT SBELL-20021009123336532PDT SBELL-20021009123338204PDT SBELL-2002100912337673PDT



13 rows selected.

The following query 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.

1 row selected.

Example 3-24 Joining Data from an XMLType Table and a Relational 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.

Performing SQL Operations on XMLType Fragments Using XMLTABLE

You can use SQL/XML function XMLTable to perform SQL operations on a set of nodes that match an XQuery expression.

Example 3-21 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 3-25
- When the target XPath expression ends in a *wildcard*, in which case the nodes in the fragment can be of different types – see Example 3-27

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.

See Also:

XQuery and Oracle XML DB for more information about SQL/XML function XMLTable

Example 3-25 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 3-21 into multiple text nodes.

Example 3-26 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 3-27 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 3-25 Accessing Description Nodes Using XMLTABLE

<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, the example 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");
```

3 rows selected.

The COLUMNS clause lets you specify precise SQL data types, which can make static typechecking 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 9-2 for an example.

Example 3-26 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
                               21
SBELL-20021009123335771PDT
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 this example 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 3-27 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.
```

Updating XML Content Stored in Oracle XML DB

You can update XML content, replacing either the entire contents of a document or only particular 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.

You can make multiple changes to a document in a single operation. Each change uses an XQuery expression to identify a node to be updated, and specifies the new value for that node.

Example 3-28 updates the text node associated with element User.

Example 3-29 replaces an entire element within an XML document. The XQuery expression references the element, and the replacement value is passed as an XMLType object.

You can make multiple changes to a document in one statement. Example 3-30 changes the values of text nodes belonging to elements CostCenter and SpecialInstructions in a single SQL UPDATE statement.

Example 3-28 Updating a Text Node

```
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 =
   XMLQuery('copy $i := $p1 modify
               (for $j in $i/PurchaseOrder/User
                return replace value of node $j with $p2)
             return $i'
            PASSING OBJECT VALUE AS "p1", 'SKING' AS "p2"
            RETURNING CONTENT)
    WHERE XMLExists(
           '$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 3-29 Replacing an Entire Element Using XQuery Update

```
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 =

XMLQuery(
```



```
'copy $i := $p1 modify
             (for $j in $i/PurchaseOrder/LineItems/LineItem[1]
              return replace node $j with $p2)
           return $i'
          PASSING OBJECT VALUE AS "p1",
                  XMLType('<LineItem ItemNumber="1">
                             <Description>The Lady Vanishes</Description>
                             <Part Id="37429122129" UnitPrice="39.95"
                                   Quantity="1"/>
                           </LineItem>') AS "p2"
          RETURNING CONTENT)
        WHERE XMLExists(
                '$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");
XMLOUERY ( '$P/PURCHAS
_____
<LineItem ItemNumber="1">
  <Description>The Lady Vanishes</Description>
```

```
<Part Id="37429122129" UnitPrice="39.95" Quantity="1"/> </LineItem>
```

```
1 row selected.
```

Example 3-30 Changing Text Node Values Using XQuery Update

```
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.
UPDATE purchaseorder
  SET OBJECT VALUE =
       XMLQuery('copy $i := $p1 modify
                    ((for $j in $i/PurchaseOrder/CostCenter
                     return replace value of node $j with $p2),
```

```
(for $j in $i/PurchaseOrder/SpecialInstructions
                     return replace value of node $j with $p3))
                return $i'
                PASSING OBJECT VALUE AS "p1",
                        'B40' AS "p2",
                        'Priority Overnight Service' AS "p3"
                RETURNING CONTENT)
       WHERE XMLExists(
                '$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.
```

Generating XML Data from Relational Data

You can use Oracle XML DB to generate XML data from relational data.

- Generating XML Data from Relational Data Using SQL/XML Functions You can use standard SQL/XML functions to generate one or more XML documents.
- Generating XML Data from Relational Data Using DBURITYPE You can generate XML data from relational data using SQL function DBURIType.

Related Topics

XQuery and Oracle XML DB

The XQuery language is one of the main ways that you interact with XML data in Oracle XML DB. Support for the language includes SQL*Plus commandXQUERY and SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast.

 Generation of XML Data from Relational Data Oracle XML DB provides features for generating (constructing) XML data from relational data in the database. There are both SQL/XML standard functions and Oracle-specific functions and packages for generating XML data from relational content.

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 3-31 uses these functions to generate an XML document that contains information from the tables departments, locations, countries, employees, and jobs.

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.

You can also create and query an XMLType view that is built using the SQL/XML generation functions. Example 3-32 and Example 3-33 illustrate this. Such an XMLType view has the effect of persisting relational data as XML content. Rows in XMLType views can also be persisted as documents in Oracle XML DB Repository.

In Example 3-33, the XPath expression passed to SQL/XML function XMLExists restricts the query result set to the node that contains the Executive department information. The result is shown pretty-printed here for clarity.

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.

Example 3-31 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 = 1.location id
  AND l.country id = c.country id;
```

The query returns the following XML:

```
XMT.
                            _____
<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>CastName>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>
Example 3-32 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 = 1.location id
     AND l.country id = c.country id;
```

Example 3-33 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="Execut:	ive"]' PASSING OBJECT	_VALUE AS	"p");	
PLAN_TABLE_OUTPUT				
Plan hash value: 2414180351				
Id Operation	Name	Rows	Bytes	Cost (%CPU) Time
0 SELECT STATEMENT 1 SORT AGGREGATE * 2 HASH JOIN * 3 HASH JOIN		1 1 10 10	115	7 (15) 00:00:01

Ι	4	TABLE ACCESS BY INDEX ROWID BATCHED	EMPLOYEES	I	10	690	2	(0) 00:00:01
*	5	INDEX RANGE SCAN	EMP_DEPARTMENT	IX	10	1	1	(0) 00:00:01
	6	TABLE ACCESS FULL	JOBS	1	19	513	2	(0) 00:00:01
	7	TABLE ACCESS FULL	EMPLOYEES	1	107	2033	2	(0) 00:00:01
	8	NESTED LOOPS		1	1	80	3	(0) 00:00:01
	9	NESTED LOOPS	1		1	68	3	(0) 00:00:01
*	10	TABLE ACCESS FULL	DEPARTMENTS	1	1	19	2	(0) 00:00:01
	11	TABLE ACCESS BY INDEX ROWID	LOCATIONS	1	1	49	1	(0) 00:00:01
*	12	INDEX UNIQUE SCAN	LOC_ID_PK	- I	1		0	(0) 00:00:01
*	13	INDEX UNIQUE SCAN	COUNTRY_C_ID_P	K	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.

Generating XML Data from Relational Data Using DBURITYPE

You can generate XML data 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 3-34 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 3-35 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.

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 3-36 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.

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.

Example 3-34 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 3-35 Restricting Rows Using an XPath Predicate

```
1 row selected.
```

Example 3-36 Restricting Rows and Columns Using an XPath Predicate



<DEPARTMENT NAME>Administration</DEPARTMENT NAME>

1 row selected.

Character Sets of XML Documents

There are a few ways in which Oracle XML DB determines which character sets are used for XML documents

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

You can use an XML encoding declaration to explicitly specify the character encoding to use for a given XML entity.

- Character-Set Determination When Loading XML Documents into the Database Except for XML data obtained from a CLOB or VARCHAR value, character encoding is determined by an encoding declaration when a document is loaded into the database.
- Character-Set Determination When Retrieving XML Documents from the Database Except for XML data stored in a CLOB or VARCHAR value, you can specify the encoding to be used when it is retrieved from Oracle XML DB using a SQL client, programmatic APIs, or transfer protocols.

XML Encoding Declaration

You can use an XML encoding declaration to explicitly specify the character encoding to use for a given XML entity.

Each XML document is composed of units called entities. Each entity in an XML document can 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

Except for XML data obtained from a CLOB or VARCHAR value, character encoding is determined by an encoding declaration when a document is loaded into the database.

For XML data obtained from a CLOB or VARCHAR value, any encoding declaration present 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_REPOS.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.

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

Except for XML data stored in a CLOB or VARCHAR value, you can specify the encoding to be used when it is retrieved from Oracle XML DB using a SQL client, programmatic APIs, or transfer protocols.

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 AMERICA._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.

 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
```

• 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
```



Related Topics

• FTP Quote Methods

Oracle Database supports several FTP ${\tt quote}$ methods, which provide information directly to Oracle XML DB.

Character Sets for HTTP(S)
 You can control the character sets used for data that is transferred using HTTP(S).

See Also:

Oracle Database Globalization Support Guide for information about NLS LANG

Migrating XMLType Data to Transportable Binary XML (TBX)

Transportable Binary XML (TBX) is decoupled from central token tables. Hence, Data Pump in convention mode does not have to convert TBX data into self-contained forms during export and later re-encode using import-side token IDs. TBX data can be exported or imported like BLOBs, greatly improving export/import performance. To simplify the migration to TBX from other storage options use option TBX CLAUSE for the TRANSOFRM parameter:

• TRANSFORM=XMLTYPE STORAGE CLAUSE: "TRANSPORTABLE BINARY XML"

Forces the TRANSPORTABLE clause to be present in table creation DDLs for Binary XML data

TRANSFORM=XMLTYPE STORAGE CLAUSE: "BINARY XML" (the default value)

Forces the NOT TRANSPORTABLE clause to be present in table creation DDLs for Binary XML data

• TRANSFORM=XMLTYPE STORAGE CLAUSE: 'CLOB'

NONE clause is not an option. If a transformation is not desired, then do not use the TRANSFORM parameter.

To migrate legacy storage options to TBX, Oracle recommends that you use Online Redefinition, because it incurs no application downtime. You can use Online Redefinition for the following migration tasks:

- From XML/OR to TBX
- From XML/CLOB to TBX
- From non-schema-based CSX to TBX

Example 3-37 Creating a CSX Based Table p with the Following Specifications



Example 3-38 Migrating Table p to TBX Based Table int_p Using Online Redefinition

```
declare
  error_count pls_integer;
begin
  DBMS_REDEFINITION.CAN_REDEF_TABLE('SCOTT', 'P', DBMS_REDEFINITION.CONS_USE_ROWID);
  DBMS_REDEFINITION.START_REDEF_TABLE('SCOTT', 'P', 'INT_P', options_flag
=>DBMS_REDEFINITION.CONS_USE_ROWID );
  DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS('SCOTT', 'P', 'INT_P', 1, true, true, true, true, error_count, true);
  DBMS_REDEFINITION.SYNC_INTERIM_TABLE('SCOTT', 'P', 'INT_P');
  DBMS_REDEFINITION.FINISH_REDEF_TABLE('SCOTT', 'P', 'INT_P');
  end;
/
```



Part II

Manipulation of XML Data in Oracle XML DB

The following are covered here: XQuery, XMLType operations, and indexing of XML data.

- XQuery and Oracle XML DB The XQuery language is one of the main ways that you interact with XML data in Oracle XML DB. Support for the language includes SQL*Plus commandXQUERY and SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast.
- Query and Update of XML Data There are many ways for applications to query and update XML data that is in Oracle Database, both XML schema-based and non-schema-based.

• Indexes for 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. There are various ways that you can index XMLType data, whether it is XML schema-based or non-schema-based, and regardless of the XMLType storage model you use.

Transformation and Validation of XMLType Data

There are several Oracle SQL functions and XMLType APIs for transforming XMLType data using XSLT stylesheets and for validating XMLType instances against an XML schema.

4 XQuery and Oracle XML DB

The XQuery language is one of the main ways that you interact with XML data in Oracle XML DB. Support for the language includes SQL*Plus commandXQUERY and SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast.

- Overview of the XQuery Language
 XQuery is the W3C language designed for querying and updating XML data.
- 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, XMLTable, XMLExists, and XMLCast. 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.
- SQL/XML Functions XMLQUERY, XMLTABLE, XMLExists, and XMLCast 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.
- URI Scheme oradb: Querying Table or View Data with XQuery You can use XQuery function fn:collection to query data that is in database tables and views.
- 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.
- 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 several such pragmas.

- XQuery Static Type-Checking in Oracle XML DB When possible, Oracle XML DB performs static (compile time) type-checking of queries.
- Oracle XML DB Support for XQuery Oracle XML DB support for the XQuery language includes SQL support and support for XQuery functions and operators.

Overview of the XQuery Language

XQuery is the W3C language designed for querying and updating XML data.

Oracle XML DB supports the following W3C XQuery standards:

- XQuery 1.0 Recommendation
- XQuery Update Facility 1.0 Recommendation
- XQuery and XPath Full Text 1.0 Recommendation



This section presents an overview of the XQuery language. For more information, consult a recent book on the language or refer to the standards documents that define it, all of which are available at http://www.w3c.org/.

- XPath Expressions Are XQuery Expressions
 The XPath language is a W3C Recommendation for navigating XML documents. It is a subset of the XQuery language: an XPath expression is also an XQuery expression.
- XQuery: A Functional Language Based on Sequences
 XQuery 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.
- XQuery Expressions

XQuery expressions are case-sensitive. An XQuery expression is either a *simple* expression or an *updating* expression, the latter being an expression that represents data modification. More precisely, these are the possible XQuery expressions:

FLWOR Expressions

Just as for XQuery in general, there is a lot to learn about FLWOR expressions in particular. This section provides a brief overview.

XPath Expressions Are XQuery Expressions

The XPath language is a W3C Recommendation for navigating XML documents. It is a subset of the XQuery language: an XPath expression is also an XQuery expression.

XPath models an XML document as a tree of nodes. It provides a set of operations that walk this tree and apply predicates and node-test functions. Applying an XPath expression to an XML document results in a set of nodes. For example, the expression /PO/PONO selects all PONO child elements under the PO root element of a document.

Table 4-1 lists some common constructs used in XPath.

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.
/	Used as a path separator to identify the child element nodes of a given element node. For example, /PurchaseOrder/Reference identifies Reference elements that are children of PurchaseOrder elements that are children 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.

Table 4-1 Common XPath Constructs



XPath Construct	Description		
Functions	XPath and XQuery support a set of built-in functions such as <pre>substring</pre> , <pre>round</pre> , and <pre>not</pre> . In addition, these languages provide for extension functions through the use of namespaces. Oracle XQuery extension functions use the namespace prefix ora, for namespace <pre>http:// xmlns.oracle.com/xdb</pre> . See Oracle XQuery Extension Functions .		
	An XPath expression must identify a single node or a set of element, text, or attribute nodes. The result of evaluating an XPath expression is never a Boolean expression.		
	You can select XMLType data using PL/SOL_C_or Java_You can also use XMLType method		

Table 4-1 (Cont.) Common XPath Constructs

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 value.

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.

XQuery: A Functional Language Based on Sequences

XQuery 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, the **XQuery Data Model** (XDM), which is based on *sequences*. Another name for an XQuery sequence is an **XDM instance**.

XQuery Is About Sequences

XQuery is all about manipulating sequences. This makes XQuery similar to a setmanipulation 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.

• XQuery Is Referentially Transparent XQuery is a *functional* language. As such, it consists of a set of possible *expressions* that are *evaluated* and whose evaluation returns *values* (results).

XQuery Update Has Side Effects on Your Data

Referential transparency applies to the evaluation of XQuery expressions. It does not imply that this evaluation never has a *side effect* on your *data*. In particular, you use XQuery Update to modify your data. That modification is a side effect of evaluating an XQuery updating expression.



XQuery Update Snapshots

An XQuery expression (query) can call for more than one update operation. XQuery Update performs all such operations for the same query as an *atomic* operation: either they all succeed or none of them do (if an error is raised).

XQuery Full Text Provides Full-Text Search

The XQuery and XPath Full Text 1.0 Recommendation (XQuery Full Text) defines XQuery support for full-text searches in queries. It defines full-text selection operators that perform the search and return instances of the AllMatches model, which complements the XQuery Data Model (XDM).

XQuery Is About Sequences

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 Referentially Transparent

XQuery is a *functional* language. As such, it consists of a set of possible *expressions* that are *evaluated* and whose evaluation returns *values* (results).

The result of evaluating an XQuery expression has two parts, at least one of which is empty: (a) a sequence (an XDM instance) and (b) a **pending update list**. Informally, the sequence is sometimes spoken of as the expression value, especially when the pending update list is empty, meaning that no data updates are involved.

As a functional language, XQuery is also **referentially transparent**. This means that 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.



XQuery Update is not in the list; it does *not* present an exception to referential transparency. See XQuery Update Has Side Effects on Your Data.

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 Update Has Side Effects on Your Data

Referential transparency applies to the evaluation of XQuery expressions. It does not imply that this evaluation never has a *side effect* on your *data*. In particular, you use XQuery Update to modify your data. That modification is a side effect of evaluating an XQuery updating expression.

The side effect is one thing; the expression value is another. The value returned from evaluation includes the pending update list that describes the updates to carry out. For a given XQuery expression, this description is the same regardless of the context in which evaluation occurs (with the above-mentioned exceptions).

The XQuery Update standard defines how the XDM instances of your data are updated. How those updates are propagated to persistent data stores (for example XMLType tables and columns) is implementation-dependent.

XQuery Update Snapshots

An XQuery expression (query) can call for more than one update operation. XQuery Update performs all such operations for the same query as an *atomic* operation: either they all succeed or none of them do (if an error is raised).

The unit of change is thus an entire XQuery query. To effect this atomic update behavior, before evaluating your query XQuery Update takes a **snapshot** of the data (XDM instances) whose modification is called for by the query. It also adds the update operations called for by the query to the pending update list. The snapshot is an evaluation context for an XDM instance that is the update target.

As the last step of XQuery expression evaluation, the pending update list is processed, applying the indicated update operations in an atomic fashion, and terminating the snapshot.

The atomic nature of snapshot semantics means that a set of update operations used in a given query are not necessarily applied in the order written. In fact, the order of applying update operations is fixed and specified by the XQuery Update Feature standard.

This means that *an update operation does not see the result of any other update operation for the same query*. There is no notion of an intermediate or interim update state – all updates for a query are applied together, atomically.

XQuery Full Text Provides Full-Text Search

The XQuery and XPath Full Text 1.0 Recommendation (XQuery Full Text) defines XQuery support for full-text searches in queries. It defines full-text selection operators that perform the search and return instances of the AllMatches model, which complements the XQuery Data Model (XDM).

An AllMatches instance describes all possible solutions to a full-text query for a given search context item. Each solution is described by a Match instance, which contains the search-

context tokens (StringInclude instances) that must be included and those (StringExclude instances) that must be excluded.

In short, XQuery Full Text adds a full-text contains expression to the XQuery language. You use such an expression in your query to search the text of element nodes and their descendent elements (you can also search the text of attribute nodes).

See Also:

Case-Sensitive Indexing and Querying for information about case-sensitive search.

XQuery Expressions

XQuery expressions are case-sensitive. An XQuery expression is either a *simple* expression or an *updating* expression, the latter being an expression that represents data modification. More precisely, these are the possible XQuery expressions:

- **Basic updating expression** an insert, delete, replace, or rename expression, or a call to an *updating function* (see the XQuery Update Facility 1.0 Recommendation).
- Updating expression a basic updating expression or an expression (other than a transform expression) that contains another updating expression (this is a recursive definition).
- Simple expression An XQuery 1.0 expression. It does not call for any updating.

The pending update list that results from evaluating a simple expression is empty. The sequence value that results from evaluating an updating expression is empty.

Simple 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. Sequencemanipulating 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 constructs the XML element <a>33.



 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 (Boolean¹) 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.)
- 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 42. (It is not, strictly speaking, a type expression, because its value does not represent a type.)

• **Full-text contains expression** – An XQuery expression that represents a full-text search. This expression is provided by the XQuery and XPath Full Text 1.0 Recommendation. A full-text contains expression (FTContainsExpr) supported by Oracle has these parts: a

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.

search context that specifies the items to search, and a full-text selection that filters those items, selecting matches.

The selection part is itself composed of the following:

- **Tokens and phrases** used for matching.
- Optional **match options**, such as the use of stemming.
- Optional Boolean operators for combining full-text selections.
- Optional constraint operators, such as positional filters (e.g. ordered window).

See Support for XQuery Full Text.

FLWOR Expressions

Just as for XQuery in general, there is a lot to learn about FLWOR expressions in particular. This section provides 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. Only the return clause can contain an updating expression; the other clauses cannot.

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 searlier that is listed before a variable slater in the for list, can be used in the binding of variable slater. For example, during its second iteration, this expression binds si to 4 and sj to 6 (2+4):

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.

If the return clause contains an updating expression then that expression is evaluated for each tuple generated by the other clauses. The pending update lists from these evaluations are then merged as the result of the FLWOR expression.

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.

Using a FLWOR expression (with order by) is the *only* way to construct an XQuery sequence in any order other than document order.



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, XMLTable, XMLExists, and XMLCast. 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.

Note:

Oracle XML Developer's Kit (XDK) supports XQuery on the mid-tier. You do not need access to Oracle Database to use XQuery. XDK lets you evaluate XQuery expressions using XQuery API for Java (XQJ).

• 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.

Predefined XQuery Namespaces and Prefixes
 Several namespaces and prefixes are predefined for use with XQuery in Oracle XML DB.

Related Topics

- SQL/XML Functions XMLQUERY, XMLTABLE, XMLExists, and XMLCast 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.
- Using the SQL*Plus XQUERY Command You can evaluate an XQuery expression using the SQL*Plus XQUERY command.
- Query and Update of XML Data There are many ways for applications to query and update XML data that is in Oracle Database, both XML schema-based and non-schema-based.



See Also: XQuery API for Java (XQJ) 1.0 Specification, March 2009 This specification is quite concrete and helpful, with understandable examples. Oracle XQuery Extension Functions for Oracle-specific XQuery functions that extend the language Oracle XML DB Support for XQuery for details about Oracle XML DB support for XQuery Oracle XML Developer's Kit Programmer's Guide for information about using XQJ

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.

A general pattern of use is that XQuery is often used when the focus is the world of XML data, and 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 XQuery Namespaces and Prefixes

Several namespaces and prefixes are predefined for use with XQuery in Oracle XML DB.

Table 4-2 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 XQueryexpression prolog. You can redefine any of them *except* xml in the prolog. All of these prefixes except ora are predefined in the XQuery standard.

SQL/XML Functions XMLQUERY, XMLTABLE, XMLExists, and XMLCast

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.

They are referred to in this book as SQL/XML *query and update* functions. 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 chapter. 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.

- XMLQUERY SQL/XML Function in Oracle XML DB Use SQL/XML function XMLQuery to construct or query XML data.
- 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 insert this data into a pre-existing database table, or you can query it using SQL — in a join expression, for example.
- XMLEXISTS SQL/XML Function in Oracle XML DB SQL/XML standard function XMLExists checks whether a given XQuery expression returns a non-empty XQuery sequence. If so, the function returns TRUE. Otherwise, it returns FALSE.
- Using XMLExists to Find a Node You can use SQL/XML standard function XMLExists to find a given node. You can create function-based indexes using XMLExists. You can also create an XMLIndex index to help speed up arbitrary XQuery searching.
- XMLCAST SQL/XML Function in Oracle XML DB You can use SQL/XML function XMLCast to cast an XQuery value to a SQL data type.
- Using XMLCAST to Extract the Scalar Value of an XML Fragment You can use standard SQL/XML function XMLCast to extract the scalar value of an XML fragment.



See Also:

- Oracle Database SQL Language Reference for information about Oracle support for the SQL/XML standard
- http://www.w3.org/TR/xquery-30/ for information about the XQuery language
- Generation of XML Data Using SQL Functions for information about using other SQL/XML functions with Oracle XML DB

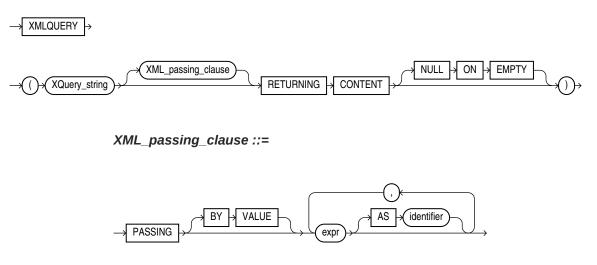
XMLQUERY SQL/XML Function in Oracle XML DB

Use SQL/XML function XMLQuery to construct or query XML data.

The 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.





- 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.

Note:

Prior to Oracle Database 11g 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 11g Release 2.

See Also:

Oracle Database SQL Language Reference for reference information about SQL/XML function XMLQuery in Oracle Database

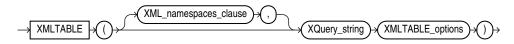
XMLTABLE SQL/XML Function in Oracle XML DB

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 insert this data into a pre-existing database table, or you can query it using SQL — in a join expression, for example.

SeeExample 5-9.

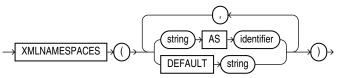
You use XMLTable in a SQL FROM clause.

Figure 4-2 XMLTABLE Syntax





XML_namespaces_clause ::=

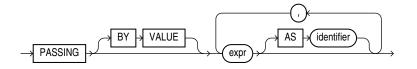


Note: You can specify at most one DEFAULT string clause.

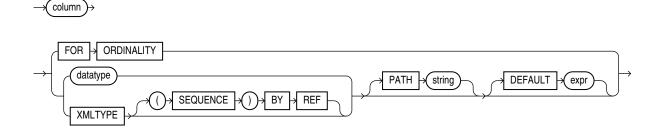
XMLTABLE_options ::=



XML_passing_clause ::=



XML_table_column ::=



- *XQuery_string* is sometimes called the **row pattern** of the XMLTable call. It 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). The row numbers start with 1. There must be at most one FOR ORDINALITY clause.
 - For each resulting *column* except the FOR ORDINALITY column, you must specify the column data type, which can be XMLType or any other SQL data type (called *datatype* in the syntax description). The resulting column content is an instance of the data type specified.
 - For data type XMLType, if you also include the specification (SEQUENCE) BY REF then a reference to the source data targeted by the PATH expression (*string*) is returned as the *column* content. Otherwise, *column* contains a *copy* of that targeted data.

Returning the XMLType data by reference lets you specify other columns whose paths target nodes in the source data that are outside those targeted by the PATH expression for *column*. See Example 5-13.

 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. This XQuery expression is sometimes called the **column pattern**. 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:

Oracle Database SQL Language Reference for reference information about SQL/XML function XMLTable in Oracle Database



Note:

Prior to Oracle Database 11g 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 11g Release 2.

See Oracle Database SQL Language Reference for information about the SQL TABLE collection expression.

Chaining Calls to SQL/XML Function XMLTABLE

When you need to expose data contained at multiple levels in an XMLType table as individual rows in a relational table (or view), you use the same general approach as for breaking up a single level: Use SQL/XML function XMLTable to define the columns making up the table and map the XML nodes to those columns.

Chaining Calls to SQL/XML Function XMLTABLE

When you need to expose data contained at multiple levels in an XMLType table as individual rows in a relational table (or view), you use the same general approach as for breaking up a single level: Use SQL/XML function XMLTable to define the columns making up the table and map the XML nodes to those columns.

But in this case you apply function XMLTable to each document level that is to be broken up and stored in relational columns. Use this technique of **chaining** multiple XMLTable calls whenever there is a one-to-*many* (1:N) relationship between documents in the XMLType table and the rows in the relational table.

You pass one level of XMLType data from one XMLTable call to the next, specifying its column type as XMLType.

When you chain two XMLTable calls, the *row pattern* of each call should target the *deepest node that is a common ancestor* to all of the nodes that are referenced in the *column patterns* of that call.

This is illustrated in Example 4-1.

Each PurchaseOrder element in XMLType table po_binaryxml 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 use XMLTable to project the collection of LineItem elements.

When element PurchaseOrder is decomposed by the first call to XMLTable, its descendant LineItem element is mapped to a column of type XMLType, which contains an XML fragment. That column is then passed to a second call to XMLTable to be broken by it into its various parts as multiple columns of relational values.

The first call to XMLTable uses /PurchaseOrder as the row pattern, because PurchaseOrder is the deepest common ancestor node for the column patterns, Reference and LineItems/LineItem.



The second call to XMLTable uses /LineItem as its row pattern, because that node is the deepest common ancestor node for each of its column patterns (@ItemNumber, Description, Part/@Id, and so on).

The column pattern (LineItems/LineItem) for the column (po.lineitem) that is passed from the first XMLTable call t o the second ends with the repeating element (LineItem) that the second XMLTable call decomposes. That repeating element, written with a leading slash (/), is used as the first element of the row pattern for the second XMLTable call.

The row pattern in each case is thus expressed as an *absolute* path; that is, it starts with /. It is the starting point for decomposition by XMLTable. Column patterns, on the other hand, *never* start with a slash (/); they are always relative to the row pattern of the same XMLTable call.

Example 4-1 Chaining XMLTable Calls

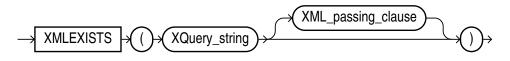
```
SELECT po.reference, li.*
 FROM po binaryxml p,
      XMLTable('/PurchaseOrder' PASSING p.OBJECT VALUE
               COLUMNS
                 reference VARCHAR2(30) PATH 'Reference',
                 lineitem XMLType
                                       PATH 'LineItems/LineItem') po,
      XMLTable('/LineItem' PASSING po.lineitem
               COLUMNS
                                          PATH '@ItemNumber',
                 itemno
                           NUMBER(38)
                 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;
```

XMLEXISTS SQL/XML Function in Oracle XML DB

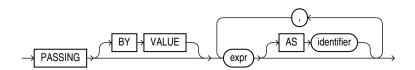
SQL/XML standard function XMLExists checks whether a given XQuery expression returns a non-empty XQuery sequence. If so, the function returns TRUE. Otherwise, it returns FALSE.

Figure 4-3 describes the syntax for function XMLExists.

Figure 4-3 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 XQuery Namespaces and Prefixes.



• 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

Note:

Prior to Oracle Database 11g 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 11g 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.

Using XMLExists to Find a Node

You can use SQL/XML standard function XMLExists to find a given node. You can create function-based indexes using XMLExists. You can also create an XMLIndex index to help speed up arbitrary XQuery searching.

Example 4-2 uses XMLExists to select rows with SpecialInstructions set to Expedite.

Example 4-2 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"</pre>
```

13 rows selected.

XMLCAST SQL/XML Function in Oracle XML DB

You can use SQL/XML function XMLCast to cast an XQuery value to a SQL data type.

Figure 4-4 describes the syntax for SQL/XML standard function XMLCast.





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.

Note:

- Prior to Oracle Database 11g Release 2, some users employed Oracle SQL function <code>extractValue</code> to do some of what can be done better using SQL/XML functions XMLQuery and XMLCast. Function <code>extractValue</code> is *deprecated* in Oracle Database 11g 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.

Related Topics

- Indexing XMLType Data Stored Object-Relationally You can effectively index XMLType data that is stored object-relationally by creating B-tree indexes on the underlying database columns that correspond to XML nodes.
- XMLIndex

SKING-20021009123336822PDT AWALSH-20021009123336101PDT WSMITH-20021009123336412PDT AWALSH-20021009123337954PDT SKING-20021009123338294PDT

Using XMLCAST to Extract the Scalar Value of an XML Fragment

You can use standard SQL/XML function XMLCast to extract the scalar value of an XML fragment.

The query in Example 4-3 extracts the scalar value of node Reference.

Example 4-3 Extracting the Scalar Value of an XML Fragment Using XMLCAST

WSMITH-20021009123338154PDT TFOX-20021009123337463PDT

13 rows selected.

URI Scheme oradb: Querying Table or View Data with XQuery

You can use XQuery function fn:collection to query data that is in database tables and views.

Besides using 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), you can use 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/ROWPRED/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 4-3. Use of any other type raises an error.



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	<pre>xs:date, xs:time, or xs:dateTime</pre>
INTERVAL YEAR TO MONTH	xs:yearMonthDuration
INTERVAL DAY TO SECOND	xs:dayTimeDuration
RAW	xs:hexBinary
ROWID	xs:string or string literal

Table 4-3 oradb Expressions: Column Types for Comparisons

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:

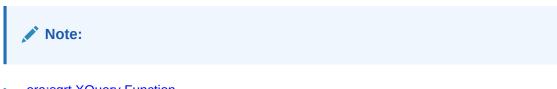
fn:collection('oradb:/OE/WAREHOUSES/ROW[WAREHOUSE_ID < 6]/WAREHOUSE_SPEC')</pre>

Related Topics

• Querying Relational Data Using XQuery and URI Scheme oradb Examples are presented that use XQuery to query relational table or view data as if it were XML data. The examples use XQuery function fn:collection, passing as argument a URI that uses the URI-scheme name oradb together with the database location of the data.

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.



ora:sqrt XQuery Function

Oracle 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 Oracle XQuery function ora:tokenize lets you use a regular expression to split the input string target string into a sequence of strings.



ora:sqrt XQuery Function

Oracle 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:sqrt Syntax

ora:sqrt (number)

ora:tokenize XQuery Function

Oracle XQuery function ora:tokenize lets you use a regular expression to split the input string target_string into a sequence of strings.

ora:tokenize Syntax

ora:tokenize (target string, match pattern [, match parameter])

Function ora:tokenize treats each substring that matches the regular-expression <code>match_pattern</code> as a separator indicating where to split. It returns the sequence of tokens as an XQuery value of type <code>xs:string*</code> (a sequence of <code>xs:string</code> values). If <code>target_string</code> is the empty sequence, it is returned. Optional argument <code>match_parameter</code> is a code that qualifies matching: case-sensitivity and so on.

The argument types are as follows:

- target string xs:string?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 several such pragmas.

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:child-element-name name #) - Specify the name to use for a child element that is inserted. In general, without this pragma the name of the element to be inserted is

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.



unknown at compile time. Specifying the name allows for compile-time optimization, to improve runtime performance.

As an example, the following SQL statement specifies LineItem as the name of the element node that is inserted as a child of element LineItems. The element data to be inserted is the value of XQuery variable p2, which comes from bind variable :1.

This pragma applies to XMLType data stored either object-relationally or as binary XML.

- (#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.
- (#ora:invalid_path empty #) Treat an invalid XPath expression as if its targeted nodes do not exist. For example:

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></ROW>:

(#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</ROW>:

• (#ora:no_xmlquery_rewrite #) - 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 for information about optimizer hint NO_XML_QUERY_REWRITE

• (#ora:xmlquery_rewrite #)⁴ - 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:no_xmlquery_rewrite and ora:xmlquery_rewrite, 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:xmlquery_rewrite will be optimized, if possible.

 (#ora:no_schema #) - Do not raise an error if an XQuery Full Text expression is used with XML Schema-based XMLType data. Instead, implicitly cast the data to non XML-Schemabased data. In particular, this means ignore XML Schema data types.

Oracle supports XQuery Full Text only for XMLType data stored as binary XML, so this pragma applies only for the same case.

• (#ora:use_xmltext_idx #) - Use an XML search index, if available, to evaluate the query. Do not use an XMLIndex index or streaming evaluation.

⁴ Prior to Oracle Database 12c Release 1 (12.1.0.1), pragmas ora:no_xmlquery_rewrite and ora:xmlquery_rewrite were named ora:xq_proc and ora:xq_qry, respectively. They were renamed for readability, with no change in meaning.

An XML search index applies only to XMLType data stored as binary XML, so this pragma does also.

 (#ora:transform_keep_schema #) – Retain XML Schema information for the documents returned by the XQuery expression that follows the pragma. This is useful for XQuery Update, which uses copy semantics.

Without the pragma, when XML schema-based data is copied during an XQuery Update operation, the XML schema information is lost. This is the behavior specified by the XQuery Update standard. If you then try to insert the updated data into an XML schema-based column or table then an error is raised: the data to be inserted is untyped, so it does not conform to the XML schema.

If you use the pragma then the data retains its XML schema information, preventing the insertion error. Here is an example of using the pragma:

```
SELECT XMLQuery('declare default element namespace
    "http://xmlns.oracle.com/xdb/xdbconfig.xsd"; (: :)
    (#ora:transform_keep_schema#)
    {copy $NEWXML :=
        $XML modify (for $CFG in $NEWXML/xdbconfig//httpconfig
            return (replace value of node
            $CFG/http-port with
xs:int($PORTNO)))
            return $NEWXML}'
        PASSING CFG AS "XML", 81 as "PORTNO" RETURNING CONTENT)
        FROM DUAL;
```

XQuery Static Type-Checking in Oracle XML DB

When possible, Oracle XML DB performs static (compile time) type-checking of queries.

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 typechecking 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.
- Using XQuery Update with a transform expression whose input is from an XMLType table or column that is based on an XML schema.

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 4-4 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.

In Example 4-5, 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 4-4 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)
```

Example 4-5 Static Type-Checking of XQuery Expressions: XML Schema-Based Data

```
-- 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)
```

Oracle XML DB Support for XQuery

Oracle XML DB support for the XQuery language includes SQL support and support for XQuery functions and operators.

 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. Oracle XML DB is a general XQuery implementation, but it is in addition specifically designed to make relational and XQuery queries work well together.

• 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 a few exceptions.



• Support for XQuery Full Text

Oracle XML DB supports XQuery Full Text for XMLType data that is stored as binary XML. Oracle Text technology provides the full-text indexing and search that is the basis of this support.

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. 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 Standards The XQuery standards specify several aspects of language processing that are to be defined by the implementation.
- XQuery Features Not Supported by Oracle XML DB The features specified by the XQuery standard that are not supported by Oracle XML DB are specified.
- XQuery Optional Features The optional XQuery features that are not supported by Oracle XML DB are specified.

Related Topics

 Support for XQuery Full Text Oracle XML DB supports XQuery Full Text for XMLType data that is stored as binary XML. Oracle Text technology provides the full-text indexing and search that is the basis of this support.

Implementation Choices Specified in the XQuery Standards

The XQuery standards specify several aspects of language processing that are 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.
- copy-namespaces default value The default value for a copy-namespaces declaration (used in XQuery Update) is inherit.
- Revalidation mode The default mode for XQuery Update transform expression revalidation is skip. However, if the result of a transform expression is an update to XML schema-based data in an XMLType table or column, then XML schema validation is enforced.



XQuery Features Not Supported by Oracle XML DB

The features specified by the XQuery standard that are not supported by Oracle XML DB are specified.

- 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 Update function fn:put.

XQuery Optional Features

The optional XQuery features that are not supported by Oracle XML DB are specified.

The XQuery standard specifies that some features are *optional* for a given implementation. The following optional XQuery features are *not* supported by Oracle XML DB:

- Schema Validation Feature
- Module Feature

The following optional XQuery features are supported by Oracle XML DB:

- XQuery Static Typing Feature
- XQuery Update Static Typing Feature

Related Topics

 XQuery Static Type-Checking in Oracle XML DB When possible, Oracle XML DB performs static (compile time) type-checking of queries.

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 a few exceptions.

Oracle XML DB does not support the following XQuery functions and operators:

- Function fn:tokenize. Use Oracle XQuery function ora:tokenize instead.
- 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:docavailable for all resources in Oracle XML DB Repository.

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:docavailable. It returns true if its document argument exists, false if not.

See Also: XQuery 3.0 Functions and Operators

Support for XQuery Full Text

Oracle XML DB supports XQuery Full Text for XMLType data that is stored as binary XML. Oracle Text technology provides the full-text indexing and search that is the basis of this support.

Refer to the XQuery and XPath Full Text 1.0 Recommendation (hereafter XQuery Full Text, or XQFT) for information about any terms that are not detailed here.

Oracle supports XQuery Full Text only for XMLType data that is stored as binary XML. You can perform a full-text search of XMLType data that is stored object-relationally using an Oracle Text index, but not using XQuery Full Text.

A general rule for understanding Oracle support for XQuery Full Text is that the Oracle implementation of XQFT is based on Oracle Text, which provides full-text indexing and search for Oracle products and for applications developed using them. The XQFT support details provided in this section are a consequence of this Oracle Text based implementation.

XQuery Full Text, XML Schema-Based Data, and Pragma ora:no_schema Use Oracle pragma ora:no schema with XQuery Full Text to query XML Schema-based XMLType data that is stored as binary XML. The data is treated as if it were non XML Schema-based.



- Restrictions on Using XQuery Full Text with XMLExists Restrictions are specified for using XQuery Full Text with SQL/XML function XMLExists.
- Supported XQuery Full Text FTSelection Operators
 Oracle XML DB supports a subset of the XQuery Full Text FTSelection operators.
- Supported XQuery Full Text Match Options Oracle XML DB supports a subset of the XQuery Full Text match options.
- Unsupported XQuery Full Text Features The XQuery Full Text features that are not supported by Oracle XML DB are specified.
- XQuery Full Text Errors Compile-time errors that can be raised when you use XQuery Full Text are described.

See Also:

- Oracle Text Application Developer's Guide
- Oracle Text Reference

XQuery Full Text, XML Schema-Based Data, and Pragma ora:no_schema

Use Oracle pragma ora:no_schema with XQuery Full Text to query XML Schema-based XMLType data that is stored as binary XML. The data is treated as if it were non XML Schema-based.

You can use XQuery Full Text to query XMLType data that is stored as binary XML. However, if you use it with XML Schema-based data then you must also use the XQuery extension-expression pragma ora:no schema in your query, or else an error is raised.

And if you use ora:no_schema then, for purposes of XQuery Full Text, the XML data is implicitly cast to non XML Schema-based data. In other words, Oracle support of XQuery Full Text treats all XML data as if it were not based on an XML schema.

In particular, this means that if you include in your query an XQuery Full Text condition that makes use of XML Schema data types, such type considerations are ignored. A comparison of two XML Schema date values, for instance, is handled as a simple string comparison. Oracle support for XQuery Full Text is not XML Schema-aware.

Related Topics

Pragma ora:no_schema: Using XML Schema-Based Data with XQuery Full Text

Oracle recommends in general that you use *non* XML Schema-based XMLType data when you use XQuery Full Text and an XML search index. But you can in some circumstances use XML Schema-based XMLType data that is stored as binary XML. Oracle XQuery pragma ora:no schema can be useful in this context.

Restrictions on Using XQuery Full Text with XMLExists

Restrictions are specified for using XQuery Full Text with SQL/XML function XMLExists.

You can pass only one XMLType instance as a SQL expression in the PASSING clause of SQL/XML function XMLExists, and each of the other, non-XMLType SQL expressions in that clause must be either a *compile-time constant* of a SQL built-in data type or a *bind variable*



that is bound to an instance of such a data type. If this restriction is not respected then compile-time error ORA-18177 is raised.

Supported XQuery Full Text FTSelection Operators

Oracle XML DB supports a subset of the XQuery Full Text FTSelection operators.

Oracle XML DB supports *only* the following XQuery Full Text FTSelection operators. Any applicable restrictions are noted. Use of the terms "must" and "must not" means that an error is raised if the specified restriction is not respected. Use of any operators not listed here raises an error.

- FTAnd (ftand)
- FTMildNot (not in)

Each operand for operator FTMildNot must be either a term or a phrase, that is, an instance of FTWords. It must not be an expression. Oracle handles FTMildNot the same way it handles Oracle Text operator MNOT.

- FTOr (ftor)
- FTOrder (ordered)

Oracle supports the use of FTOrder *only* when used in the context of a window (FTWindow). Otherwise, it is not supported. For example, you can use ordered window 5 words, but you cannot use only ordered without also window. Oracle handles FTOrder the same way it handles Oracle Text operator NEAR with a TRUE value for option ORDER.

• FTUnaryNot (ftnot)

FTUnaryNot must be used with FTAnd. You cannot use FTUnaryNot by itself. For example, you can use ftand ftnot, but you cannot use only ftnot without also ftand. Oracle handles FTUnaryNot the same way it handles Oracle Text operator NOT.

• FTWindow (window)

Oracle handles FTWindow the same way it handles Oracle Text operator NEAR. You must specify the window size only in words, not in sentences or paragraphs (for example, window 2 paragraphs), and you must specify it as a numeric constant that is less than or equal to 100.

FTWords

FTWordsValue must be an XQuery literal string or a SQL bind variable whose value is passed to SQL function XMLExists or XMLQuery from a SQL expression whose evaluation returns a non-XMLType value.

In addition, FTAnyallOption, if present, must be any. That is, FTWords must correspond to a sequence with only one item.

Note:

Even though FTWords corresponds to a sequence of only one item, you can still search for a phrase of multiple words, by using a single string for the entire phrase. So for example, although Oracle XML DB does not support using {"found" "necklace"} for FTWords, you can use "found necklace".

Supported XQuery Full Text Match Options

Oracle XML DB supports a subset of the XQuery Full Text match options.

Oracle XML DB supports *only* the following XQuery Full Text match options. Any applicable restrictions are noted. Use of the terms "must" and "must not" means that an error is raised if the specified restriction is not respected. Use of any match options not listed here raises an error.

• FTStemOption (stemming, no stemming)

The default behavior specified in the XQuery and XPath Full Text 1.0 Recommendation is used for each unsupported match option, with the following exceptions:

- FTLanguage (unsupported) The language used is the language defined by the *default lexer*, which means the language that was used when the database was installed.
- FTStopWordOption (unsupported) The stoplist used is the stoplist defined for that language.

💉 See Also:

- Oracle Text Reference for information about the default lexer
- Oracle Text Reference for information about the stoplist used for each supported language

Unsupported XQuery Full Text Features

The XQuery Full Text features that are not supported by Oracle XML DB are specified.

In addition to all FTSelection operators not mentioned in Supported XQuery Full Text FTSelection Operators and all match options not mentioned in Supported XQuery Full Text Match Options, Oracle XML DB does *not* support the following XQuery Full Text features:

- FTIgnoreOption
- FTWeight (weight declarations, used with FTPrimaryWithOptions)
- FTScoreVar (score variables, used with XQuery ForClause and LetClause or with XPath 2.0 SimpleForClause)

XQuery Full Text Errors

Compile-time errors that can be raised when you use XQuery Full Text are described.

A compile-time error is raised whenever you use an XQuery Full Text (XQFT) feature that Oracle does not support.

In addition, compile-time error ORA-18177 is raised whenever you use a supported XQFT expression in a SQL WHERE clause (typically in XMLExists), if you did not create a corresponding XML search index or if that index is not picked up.



Related Topics

 Unsupported XQuery Full Text Features The XQuery Full Text features that are not supported by Oracle XML DB are specified.

See Also:

- Indexing XML Data for Full-Text Queries (pre-23ai) for information about creating an XML search index and handling error ORA-18177
- Performance Tuning for XQuery for information about axes other than forward and descendent
- Oracle Database SQL Language Reference for information about SQL built-in data types



5

Query and Update of XML Data

There are many ways for applications to query and update XML data that is in Oracle Database, both XML schema-based and non-schema-based.

Using XQuery with Oracle XML DB

XQuery is a very general and expressive language, and SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast combine that power of expression and computation with the strengths of SQL.

- Querying XML Data Using SQL and PL/SQL
 You can guery XML data from XMLType columns and tables in various ways.
- Using the SQL*Plus XQUERY Command You can evaluate an XQuery expression using the SQL*Plus XQUERY command.
- Using XQuery with PL/SQL, JDBC, and ODP.NET to Access Database Data You can use XQuery with the Oracle APIs for PL/SQL, JDBC, and Oracle Data Provider for .NET (ODP.NET).
- Updating XML Data

There are several ways you can use Oracle XML DB features to update XML data, whether it is transient or stored in database tables.

Performance Tuning for XQuery

A SQL query that involves XQuery expressions can often be automatically rewritten (optimized) in one or more ways. This optimization is referred to as **XML query rewrite** or optimization. When this happens, the XQuery expression is, in effect, evaluated directly against the XML document without constructing a DOM in memory.

See Also:

- Overview of How To Use Oracle XML DB for XMLType storage recommendations
- XML Schema Storage and Query: Basic for how to work with XML schemabased XMLType tables and columns
- XQuery and Oracle XML DB for information about updating XML data using XQuery Update

Using XQuery with Oracle XML DB

XQuery is a very general and expressive language, and SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast combine that power of expression and computation with the strengths of SQL.

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 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 Relational Data Using XQuery and URI Scheme oradb.

• Query XMLType data, possibly decomposing the resulting XML into relational data using function XMLTable.

See Querying XMLType Data Using XQuery.

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_REPOS.createResource('/public/emps.xml', empsxmlstring);
res := DBMS_XDB_REPOS.createResource('/public/empsns.xml', empsxmlnsstring);
  res := DBMS XDB REPOS.createResource('/public/depts.xml', deptsxmlstring);
END;
```

XQuery Sequences Can Contain Data of Any XQuery Type

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).

- Querying XML Data in Oracle XML DB Repository Using XQuery Examples are presented that use 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.
- Querying Relational Data Using XQuery and URI Scheme oradb Examples are presented that use XQuery to query relational table or view data as if it were XML data. The examples use XQuery function fn:collection, passing as argument a URI that uses the URI-scheme name oradb together with the database location of the data.

- Querying XMLType Data Using XQuery Examples are presented that use XQuery to query XMLType data.
- 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.

XQuery Sequences Can Contain Data of Any XQuery Type

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. It 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

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, $\langle A \rangle 33 \langle A \rangle$).

Example 5-2 XMLQuery Applied to a Sequence of Items of Different Types

Querying XML Data in Oracle XML DB Repository Using XQuery

Examples are presented that use 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 here 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.



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 5-4 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 5-3 FLOWR Expression Using for, let, order by, where, and return

XMLQUERY('FOR\$EINDOC("/PUBLIC/EMPS.XML")/EMPS/EMPLET\$D:=DOC("/PUBLIC/DEPTS.XML")

<emp ename="Jack" dept="Administration"></emp><emp ename="Jill" dept="Marketing"</pre>

></emp>

1 row selected.

In this example, 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.) 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. 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 FLOWR Expression Using Built-In Functions

1 row selected.

In this example, 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.

Related Topics

• XQuery Functions fn:doc, fn:collection, and fn:doc-available Oracle XML DB supports XQuery functions fn:doc, fn:collection, and fn:docavailable for all resources in Oracle XML DB Repository.

Querying Relational Data Using XQuery and URI Scheme oradb

Examples are presented that use XQuery to query relational table or view data as if it were XML data. The examples use XQuery function fn:collection, passing as argument a URI that uses the URI-scheme name oradb together with the database location of the 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).

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.

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.

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, in this example the value of pseudocolumn COLUMN_VALUE for each virtual-table row is an XML fragment (of type XMLType) with a single Warehouse element.

See Also:

- Example 5-41 for the execution plan of Example 5-6
- Example 5-42 for the execution plan of Example 5-7

Example 5-5 Querying Relational Data as XML Using XMLQuery

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.

Example 5-6 Querying Relational Data as XML Using a Nested FLWOR Expression

CONNECT hr Enter password: password

Connected.



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

```
<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.

Example 5-7 Querying Relational Data as XML Using XMLTable

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.
```

Querying XMLType Data Using XQuery

Examples are presented that use XQuery to query XMLType 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.

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.

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.

See Also:

Example 5-43 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.

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.

Example 5-11 does not use the clause RETURNING SEQUENCE BY REF, which means that the XQuery sequence returned and then used by the COLUMNS clause is passed by *value*, not by reference. That is, a copy of the targeted nodes is returned, not a reference to the actual nodes.

When the returned sequence is passed by value, the columns specified in a COLUMNS clause cannot refer to any data that is not in that returned copy. In particular, they cannot refer to data that *precedes* the targeted nodes in the source data.

To be able to refer to an arbitrary part of the source data from column specifications in a COLUMNS clause, you need to use the clause RETURNING SEQUENCE BY REF, which causes the sequence resulting from the XQuery expression to be returned by *reference*.



Example 5-12 shows the use of clause RETURNING SEQUENCE BY REF, which allows column reference to refer to a node that is outside the nodes targeted by the XQuery expression. Because the sequence of LineItem nodes is returned by reference, the code has access to the complete tree of nodes, so it can navigate upward and then back down to node Reference.

Clause RETURNING SEQUENCE BY REF lets you specify that the result of evaluating the top-level XQuery expression used to generate rows for XMLTable be returned by reference. The same kind of choice is available for the result of evaluating a PATH expression in a COLUMNS clause. To specify that such a result be returned by reference you use XMLType (SEQUENCE) BY REF as the column data type.

Example 5-13 illustrates this. It chains together two XMLTable tables, t1 and t2, returning XML data from the source document by reference:

- For column reference of the top-level table, t1, because it corresponds to a node outside element LineItem (just as in Example 5-12)
- For column part of table t1, because it is passed to table t2, whose column item targets data outside node Part

In table t1, the type used for column part is XMLType (SEQUENCE) BY REF, so that the part data is a reference to the source data targeted by its PATH expression, LineItem/Part. This is needed because the PATH expression for column item in table t2 targets attribute ItemNumber of the parent of element Part, LineItem. Without specifying that part is a reference, it would be a copy of just the Part element, so that using PATH expression .../@ItemNumber would raise an error.

Example 5-14 uses SQL/XML function XMLTable to break up the XML data in an XMLType collection element, LineItem, into separate columns of a virtual table.

See Also:

- Example 5-44 for the execution plan of Example 5-14
- Creating a Relational View over XML: Mapping XML Nodes to Columns, for an example of applying XMLTable to multiple document levels (multilevel chaining)

Example 5-8 Querying an XMLType Column Using XMLQuery PASSING Clause



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.

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-2002100912333894PDT"></A10po>
```



<A10po pono="SMCCAIN-20021009123337403PDT"></A10po>

10 rows selected.

Example 5-10 Using XMLQUERY with XML 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 <A10po pono="SMCCAIN-20021009123336151PDT"></A10po> <A10po pono="SMCCAIN-20021009123336341PDT"></A10po> <A10po pono="SMCCAIN-20021009123336341PDT"></A10po> <A10po pono="SMCCAIN-20021009123335681PDT"></A10po> <A10po pono="SMCCAIN-20021009123335681PDT"></A10po> <A10po pono="SMCCAIN-20021009123335470PDT"></A10po> <A10po pono="SMCCAIN-20021009123336972PDT"></A10po> <A10po pono="SMCCAIN-20021009123336842PDT"></A10po> <A10po pono="SMCCAIN-20021009123336842PDT"></A10po> <A10po pono="SMCCAIN-20021009123336512PDT"></A10po> <A10po pono="SMCCAIN-20021009123336512PDT"></A10po> <A10po pono="SMCCAIN-2002100912333694PDT"></A10po> <A10po pono="SMCCAIN-2002100912333694PDT"></A10po> <A10po pono="SMCCAIN-2002100912333694PDT"></A10po> <A10po pono="SMCCAIN-2002100912333694PDT"></A10po> <A10po pono="SMCCAIN-20021009123337403PDT"></A10po>

132 rows selected.

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.

Example 5-12 Using XMLTABLE with RETURNING SEQUENCE BY REF

SELECT t.* FROM purchaseorder, XMLTable('/PurchaseOrder RETURNING SEQU		eltems/LineItem' PASSING OBJECT_VALUE 3Y REF
		VARCHAR2(30) PATH '//Reference',
item		VARCHAR2(4) PATH '@ItemNumber',
descrip	ption	VARCHAR2(45) PATH 'Description') t
WHERE item = $5;$		
REFERENCE	ITEM	DESCRIPTION
AMCEWEN-20021009123336171PDT	5	Coup De Torchon (Clean Slate)
AMCEWEN-20021009123336271PDT	5	The Unbearable Lightness Of Being
PTUCKER-20021009123336191PDT	5	The Scarlet Empress
PTUCKER-20021009123336291PDT	5	The Unbearable Lightness Of Being
SBELL-20021009123336231PDT	5	Black Narcissus
SBELL-20021009123336331PDT	5	Fishing With John 1 -3
SKING-20021009123336321PDT	5	The Red Shoes
SMCCAIN-20021009123336151PDT	5	Wages of Fear
SMCCAIN-20021009123336341PDT	5	The Most Dangerous Game
VJONES-20021009123336301PDT	5	Le Trou



Example 5-14 Using XMLTABLE to Decompose XML Collection Elements into Relational Data

l: FROM pi	<pre>ines.lineitem, lines.description ines.unitprice, lines.quantity urchaseorder, 4LTable('for \$i in /PurchaseOrder where \$i/@ItemNumber >=</pre>	er/LineItems/LineIt = 8 e > 50	em	
	lineitem NUMBER	PATH '@ItemNumb	er',	
	description VARCHAR2(3			
	partid NUMBER			
	unitprice NUMBER	PATH 'Part/@Uni	tPrice',	
	quantity NUMBER	PATH 'Part/@Qua	nully) 1	lines;
LINEITEM	DESCRIPTION	PARTID UNIT	PRICE QUA	ANTITY
11	Orphic Trilogy	37429148327	80	3
		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		80	3
	Orphic Trilogy	37429148327	80	4
13	Orphic Trilogy	37429148327	80	4
10	Brazil	37429138526	60	4
	Eisenstein: The Sound Years	37429149126	80	3
12	Dreyer Box Set	37429158425	80	4

13 Dreyer Box Set

37429158425 80 4

17 rows selected.

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.

Note:

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 5-15 illustrates use of a namespace declaration in an XQuery expression.

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 5-16, 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 5-15 Using XMLQUERY with a Namespace Declaration



```
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="{$d}"/>'
             RETURNING CONTENT) FROM DUAL
1
XMLQUERY('DECLARENAMESPACEE="HTTP://EXAMPLE.COM"; FOR$IINDOC("/PUBLIC/EMPSNS.XML"
_____
<emp ename="Jack" dept=""></emp><emp ename="Jill" dept=""></emp>
```

Example 5-16 Using XMLTABLE with the XMLNAMESPACES Clause

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.

Querying XML Data Using SQL and PL/SQL

You can query XML data from XMLType columns and tables in various ways.

- Select XMLType data using SQL, PL/SQL, or Java.
- Query XMLType data using SQL/XML functions such as XMLQuery. See Querying XMLType Data Using XQuery.
- Perform full-text search using XQuery Full Text. See Support for XQuery Full Text and Indexes for XMLType Data.

The examples in this section illustrate different ways you can use SQL and PL/SQL to query XML data. Example 5-17 inserts two rows into table <code>purchaseorder</code>, then queries data in those rows using SQL/XML functions <code>XMLCast</code>, <code>XMLQuery</code>, and <code>XMLExists</code>.

Example 5-18 uses a PL/SQL cursor to query XML data. It uses a local XMLType instance to store transient data.

Example 5-19 and Example 5-20 both use SQL/XML function XMLTable to extract data from an XML purchase-order document. They then insert that data into a relational table. Example 5-19 uses SQL; Example 5-20 uses PL/SQL.

Example 5-20 defines and uses a PL/SQL procedure to extract data from an XML purchaseorder document and insert it into a relational table.

Example 5-21 tabulates the purchase orders whose shipping address contains the string "Shores" and which were requested by customers whose names contain the string "11" (double L). These purchase orders are grouped by customer and counted. The example uses XQuery Full Text to perform full-text search.

Example 5-22 extracts the fragments of a document that are 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 5-22 uses XMLType method isFragment() to determine whether the result is a fragment.

Note:

You cannot insert fragments into XMLType columns. You can use SQL/XML function XMLQuery to convert a fragment into a well-formed document.

Example 5-17 Querying XMLTYPE Data

ORACLE

```
INSERT INTO purchaseorder
 VALUES (XMLType(bfilename('XMLDIR', 'VJONES-2002091614000000PDT.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 SMCCAIN-2002091213000000PDT	SVOLLMAN SKING	-	2002-10-11 2002-10-12

2 rows selected.

Example 5-18 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",
                                                AS "r");
                                     reference
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 5-19 Extracting XML Data and Inserting It into a Relational Table Using SQL

```
CREATE TABLE purchaseorder table (reference
                                                             VARCHAR2(28) PRIMARY KEY,
                                                              VARCHAR2(48),
                                       requestor
                                                             XMLType,
                                       actions
                                                            VARCHAR2(32),
                                       userid
                                                       VARCHAR2(3),
VARCHAR2(48),
                                       costcenter
                                       shiptoname
                                                             VARCHAR2(512),
                                       address
                                       phone
                                                              VARCHAR2(32),
                                       rejectedby
                                                             VARCHAR2(32),
                                       daterejected DATE,
                                                             VARCHAR2(2048),
                                       comments
                                       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"),
                                                        VARCHAR2(14),
                                          upc
                                          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',
                                             VARCHAR2(20) FATH 'Reference',

VARCHAR2(48) PATH 'Requestor',

XMLType PATH 'Actions',

VARCHAR2(32) PATH 'User',

VARCHAR2(3) PATH 'CostCenter',

VARCHAR2(48) PATH 'ShippingInstructions/name',

VARCHAR2(512) PATH 'ShippingInstructions/name',
                             requestor
                              actions
                              userid
                              costcenter
                              shiptoname
                              address
                                                  VARCHAR2(512) PATH 'ShippingInstructions/address',
                              phoneVARCHAR2(32)PATH 'ShippingInstructions/telephone',rejectedbyVARCHAR2(32)PATH 'Reject/User',daterejectedDATEPATH 'Reject/Date',
```



VARCHAR2(2048) PATH 'Reject/Comments', 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 5-20 Extracting XML Data and Inserting It into a Table Using PL/SQL

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('Sn/PurchaseOrder', Proceeder, and the second se

SELECT * FROM XMLTable('\$p/PurchaseOrder' PAS	SING purchaseorder A	s "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',
specialinstruc	tions VARCHAR2(2048)	PATH	'SpecialInstructions');
<pre>INSERT INTO purchaseorder_lineitem (reference, SELECT t.reference, li.lineno, li.upc, li.des FROM XMLTable('\$p/PurchaseOrder' PASSING purc COLUMNS reference VARCHAR2(28) lineitem XMLType PATH ' XMLTable('LineItem' PASSING t.lineitem COLUMNS lineno NUMBER(10) PA upc VARCHAR2(14) PATH description VARCHAR2(12) quantity NUMBER(10)</pre>	cription, li.quantit haseorder AS "p" PATH 'Reference', LineItems/LineItem') TH '@ItemNumber', 'Part/@Id', 8) PATH 'Description PATH 'Part/@Quantity	y, li. t, ',	unitprice
unitprice NUMBER(12,2)	PATH 'Part/@UnitPri	ce') 1	11;

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 7155150	209058 A Night to Remember	2
SBELL-2002100912333601PDT	2 3742914	40222 The Unbearable Lightness Of E	eing 2
SBELL-2002100912333601PDT	3 7155150	011020 Sisters	4

Example 5-21 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"; (: :)
      $p/PurchaseOrder/ShippingInstructions[address/text() contains text "Shores"]'
     PASSING po.OBJECT VALUE AS "p")
    AND XMLCast (XMLQuery ('$p/PurchaseOrder/Requestor/text()'
                        PASSING po.OBJECT VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(128))
       LIKE '%11%'
  GROUP BY XMLCast (XMLQuery ('$p/PurchaseOrder/Requestor'
                           PASSING po.OBJECT VALUE AS "p" RETURNING CONTENT)
                  AS VARCHAR2(128));
                     COUNT(*)
NAME
----- -----
Allan D. McEwen
                           9
Ellen S. Abel
                            4
Sarah J. Bell
                           13
William M. Smith
                             7
```

Example 5-22 Extracting Fragments 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(*)

REFERENCE	COONT (**)
TFOX-20021009123337784PDT	3



Using the SQL*Plus XQUERY Command

You can evaluate an XQuery expression using the SQL*Plus XQUERY command.

Example 5-23 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.

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.



Example 5-23 Using the SQL*Plus XQUERY Command

<ROW><DEPARTMENT_ID>30</DEPARTMENT_ID><DEPARTMENT_NAME>Purchasing</DEPARTMENT_NAME><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>

Using XQuery with PL/SQL, JDBC, and ODP.NET to Access Database Data

You can use XQuery with the Oracle APIs for PL/SQL, JDBC, and Oracle Data Provider for .NET (ODP.NET).

Example 5-24 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 shows how to use XQuery with JDBC, binding variables by position with the PASSING clause of SQL/XML function XMLTable.

Example 5-26 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 5-24 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 5-25 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())
    {
       SQLXML sqlXml = rs.getSQLXML(1);
       System.out.println("LineItem Description: " + sqlXml.getString());
      sqlXml.free();
    }
    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 5-26 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
 /// </summarv>
 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=orallgr2"; // 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>

Related Topics

- PL/SQL APIs for XMLType There are several PL/SQL packages that provide APIs for XMLType.
- Java DOM API for XMLType

The Java DOM API for XMLType lets you operate on XMLType instances using a DOM. You can use it to manipulate XML data in Java, including fetching it through Java Database Connectivity (JDBC).

 Oracle XML DB and Oracle Data Provider for .NET
 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.

Updating XML Data

There are several ways you can use Oracle XML DB features to update XML data, whether it is transient or stored in database tables.

- Updating an Entire XML Document To update an entire XML document, use a SQL UPDATE statement.
- Replacing XML Nodes You can use XQuery Update with a SQL UPDATE statement to update an existing XML document instead of creating a new document. The entire document is updated, not just the part of it that is selected.

Inserting Child XML Nodes

You can use XQuery Update to insert new children (either a single attribute or one or more elements of the same type) under parent XML elements. The XML document that is the target of the insertion can be schema-based or non-schema-based.

- Deleting XML Nodes An example uses XQuery Update to delete XML nodes.
- Creating XML Views of Modified XML Data You can use XQuery Update to create new views of XML data.

Updating an Entire XML 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 documents stored as binary XML can be made in a piecewise manner.

Example 5-27 updates an XMLType instance using a SQL UPDATE statement.



Example 5-27 Updating XMLType Data Using SQL UPDATE

```
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('$1/LineItem' PASSING t.lineitem AS "1"
              COLUMNS lineno NUMBER(10) PATH '@ItemNumber',
                     description VARCHAR2(128) PATH 'Description') li
 WHERE t.reference = 'DAUSTIN-20021009123335811PDT' AND ROWNUM < 6;
REFERENCE
                              LINENO DESCRIPTION
_____ ____
DAUSTIN-20021009123335811PDT
                                  1 Nights of Cabiria
                               2 For All Mankind
DAUSTIN-20021009123335811PDT
DAUSTIN-20021009123335811PDT
                                 3 Dead Ringers
                                 4 Hearts and Minds
DAUSTIN-20021009123335811PDT
DAUSTIN-20021009123335811PDT 5 Rushmore
UPDATE purchaseorder po
 SET po.OBJECT VALUE = XMLType(bfilename('XMLDIR','NEW-DAUSTIN-20021009123335811PDT.xml'),
                            nls charset id('AL32UTF8'))
 WHERE XMLExists('$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('$1/LineItem' PASSING t.lineitem AS "1"
              COLUMNS lineno NUMBER(10) PATH '@ItemNumber',
                     description VARCHAR2(128) PATH 'Description') li
 WHERE t.reference = 'DAUSTIN-20021009123335811PDT';
REFERENCE
                             LINENO DESCRIPTION
_____
                            1 Dead Ringers
DAUSTIN-20021009123335811PDT
                                 2 Getrud
DAUSTIN-20021009123335811PDT
                          3 Branded to Kill
DAUSTIN-20021009123335811PDT
```

Replacing XML Nodes

You can use XQuery Update with a SQL UPDATE statement to update an existing XML document instead of creating a new document. The entire document is updated, not just the part of it that is selected.

In Example 5-28 we pass the SQL string literal 'SKING' to the XQuery expression as a variable (\$p2). In this simple example, since the value is a string literal, we could have simply used replace value of node \$j with "SKING". That is, you can just use a literal XQuery string here, instead of passing a literal string from SQL to XQuery. In real-world examples you will typically pass a value that is available only at runtime; Example 5-28 shows how to do that. This is also true of other examples.

Example 5-29 updates multiple text nodes and attribute nodes.

Example 5-30 updates selected nodes within a collection.

Example 5-31 illustrates the common mistake of using an XQuery Update replace-value operation 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.

Instead of updating only the intended node, Example 5-31 updates the values of *all* text nodes that belong to the Description element. This 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 *XQuery expression* passed to XQuery Update determines which *nodes* within the document must be updated. In this case, the XQuery expression identifies all three Description nodes, so all three of the associated text nodes were updated.

To correctly update a node that occurs multiple times within a collection, use the XQuery expression passed XQuery Update to identify which nodes in the XML document to update. By introducing the appropriate predicate into the XQuery expression, you can limit which nodes in the document are updated. Example 5-32 illustrates the correct way to update one node within a collection.

Example 5-28 Updating XMLTYPE Data Using SQL UPDATE and XQuery Update

```
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 =
   XMLQuery('copy $i := $p1 modify
              (for $j in $i/PurchaseOrder/Actions/Action[1]/User
              return replace value of node $j with $p2)
             return $i' PASSING po.OBJECT VALUE AS "p1",
                               'SKING' AS "p2" RETURNING CONTENT)
 WHERE XMLExists('$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 5-29 Updating Multiple Text Nodes and Attribute Nodes

```
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 =
    XMLQuery('copy $i := $p1 modify
                ((for $j in $i/PurchaseOrder/Requestor
                 return replace value of node $j with $p2),
                 (for $j in $i/PurchaseOrder/LineItems/LineItem[1]/Part/@Id
                 return replace value of node $j with $p3),
                 (for $j in $i/PurchaseOrder/LineItems/LineItem[1]/Description
                 return replace value of node $j with $p4),
                 (for $j in $i/PurchaseOrder/LineItems/LineItem[3]
                 return replace node $j with $p5))
               return $i'
            PASSING OBJECT VALUE AS "p1",
                    'Stephen G. King' AS "p2",
                     '786936150421' AS "p3",
                    'The Rock' AS "p4",
                    XMLType('<LineItem ItemNumber="99">
                               <Description>Dead Ringers</Description>
                               <Part Id="715515009249" UnitPrice="39.95" Quantity="2"/>
                             </LineItem>') AS "p5"
            RETURNING CONTENT)
  WHERE XMLExists('$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
                LINETTEMS
```

Example 5-30 Updating Selected Nodes within a Collection

```
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 =
     XMLQuery(
        'copy $i := $p1 modify
           ((for $j in $i/PurchaseOrder/Requestor
            return replace value of node $j with $p2),
            (for $j in $i/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]/@Quantity
            return replace value of node $j with $p3),
            (for $j in $i/PurchaseOrder/LineItems/LineItem
                         [Description/text()="The Unbearable Lightness Of Being"]
            return replace node $j with $p4))
           return $i'
        PASSING OBJECT VALUE AS "p1",
                'Stephen G. King' AS "p2",
                25 AS "p3",
```

```
XMLType('<LineItem ItemNumber="99">
                          <Part Id="786936150421" Quantity="5" UnitPrice="29.95"/>
                          <Description>The Rock</Description>
                        </LineItem>') AS "p4"
       RETURNING CONTENT)
     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>
```

Example 5-31 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
  SET OBJECT VALUE =
       XMLQuery('copy $i := $p1 modify
                   (for $j in $i/PurchaseOrder/LineItems/LineItem/Description
```

```
return replace value of node $j with $p2)

return $i'

PASSING OBJECT_VALUE AS "p1", 'The Wizard of Oz' AS "p2"

RETURNING CONTENT)

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.

Example 5-32 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
 SET OBJECT VALUE =
      XMLQuery('copy $i := $p1 modify
                  (for $j in $i/PurchaseOrder/LineItems/LineItem/Description
                               [text()="Sisters"]
                   return replace value of node $j with $p2)
                return $i'
               PASSING OBJECT VALUE
                                        AS "p1",
                       'The Wizard of Oz' AS "p2" RETURNING CONTENT)
      WHERE XMLExists ('$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.

 Updating XML Data to NULL Values Certain considerations apply to updating XML data to NULL values.

Updating XML Data to NULL Values

Certain considerations apply to updating XML data to 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 5-33.
- If you update an *attribute* value to NULL, the value appears as the empty string. See Example 5-33.
- 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 5-34.

Example 5-33 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 5-33 shows two different but equivalent ways to remove the value of a node. For element Description and attribute Quantity, a literal XQuery empty sequence, (), replaces the existing value directly. For element LineItem, SQL NULL is passed into the XQuery expression to provide the empty node value. Since the value used is literal, it is simpler not to pass it from SQL to XQuery. But in real-world examples you will often pass a value that is available only at runtime. Example 5-33 shows how to do this for an empty XQuery sequence: pass a SQL NULL value.

Example 5-34 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. In that XML schema, Description is an attribute of the Part element, not a sibling element.





Example 5-33 NULL Updates – 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
                LINETTEMS
-----
                           _____
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 =
     XMLQuery(
        'copy $i := $p1 modify
           ((for $j in $i/PurchaseOrder/LineItems/LineItem[Part/@Id="715515009058"]/Description
            return replace value of node $j with ()) ,
            (for $j in $i/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]/@Quantity
            return replace value of node $j with ()) ,
            (for $j in $i/PurchaseOrder/LineItems/LineItem
                        [Description/text() = "The Unbearable Lightness Of Being"]
            return replace node $j with $p2))
        return $i'
        PASSING OBJECT VALUE AS "p1", NULL AS "p2"
       RETURNING CONTENT)
      WHERE XMLExists('$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>

</LineItem>
```

Example 5-34 NULL Updates – 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 =
     XMLQuery(
        'copy $i := $p1 modify
           (for $j in $i/PurchaseOrder/LineItems/LineItem/Part[@Description="A Night to Remember"]
            return replace value of node $j with $p2)
         return $i
        PASSING OBJECT VALUE AS "p1", NULL AS "p2" RETURNING CONTENT)
  WHERE XMLExists('$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
```

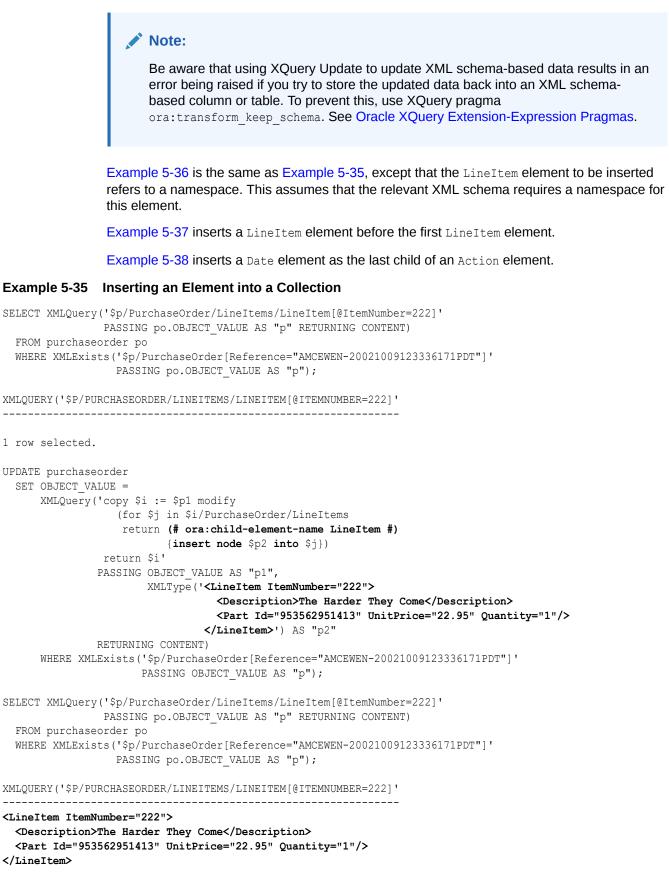
---- <Part Description="A Night to Remember" UnitCost="39.95"/>

Inserting Child XML Nodes

You can use XQuery Update to insert new children (either a single attribute or one or more elements of the same type) under parent XML elements. The XML document that is the target of the insertion can be schema-based or non-schema-based.

Example 5-35 inserts a new LineItem element as a child of element LineItems. It uses the Oracle XQuery pragma ora:child-element-name to specify the name of the inserted child element as LineItem.

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.



1 row selected.

Example 5-36 Inserting an Element that Uses a Namespace

```
UPDATE purchaseorder
 SET OBJECT VALUE =
     XMLQuery('declare namespace e = "films.xsd"; (: :)
               copy $i := $p1 modify
                 (for $j in $i/PurchaseOrder/LineItems
                  return (# ora:child-element-name e:LineItem #)
                         {insert node $p2 into $j})
               return $i'
              PASSING OBJECT VALUE AS "p1",
                     XMLType('<e:LineItem ItemNumber="222">
                                <Description>The Harder They Come</Description>
                                <Part Id="953562951413" UnitPrice="22.95" Quantity="1"/>
                              </e:LineItem>') AS "p2"
              RETURNING CONTENT)
     WHERE XMLExists ('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                     PASSING OBJECT VALUE AS "p");
               Example 5-37 Inserting an Element Before an 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>Salesperson</Description>
                 <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
               </LineItem>
               UPDATE purchaseorder
                 SET OBJECT VALUE =
                     XMLQuery('copy $i := $p1 modify
                                 (for $j in $i/PurchaseOrder/LineItems/LineItem[1]
                                 return insert node $p2 before $j)
                               return $i'
                              PASSING OBJECT VALUE AS "p1",
                                      XMLType('<LineItem ItemNumber="314">
                                                <Description>Brazil</Description>
                                                <Part Id="314159265359" UnitPrice="69.95"
                                                      Quantity="2"/>
                                              </LineItem>') AS "p2"
                              RETURNING CONTENT)
                     WHERE XMLExists ('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                                     PASSING OBJECT VALUE AS "p");
               SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[position() <= 2]'</pre>
                               PASSING po.OBJECT VALUE AS "p" RETURNING CONTENT)
                 FROM purchaseorder po
                 WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                                 PASSING po.OBJECT VALUE AS "p");
               XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[POSITION()<=2]'PASSINGPO.OBJECT</pre>
                _____
               <LineItem ItemNumber="314">
                 <Description>Brazil</Description>
                 <Part Id="314159265359" UnitPrice="69.95" Quantity="2"/>
               </LineItem>
               <LineItem ItemNumber="1">
```

```
<Description>Salesperson</Description>
 <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
</LineItem>
Example 5-38 Inserting an Element as the Last Child 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 =
     XMLQuery('copy $i := $p1 modify
                (for $j in $i/PurchaseOrder/Actions/Action[1]
                 return insert nodes $p2 as last into $j)
              return $i'
              PASSING OBJECT VALUE AS "p1",
                    XMLType('<Date>2002-11-04</Date>') AS "p2"
             RETURNING CONTENT)
     WHERE XMLExists ('$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>
```

Deleting XML Nodes

An example uses XQuery Update to delete XML nodes.

Example 5-39 deletes the LineItem element whose ItemNumber attribute has value 222.

Example 5-39 Deleting an Element

```
_____
```

```
<LineItem ItemNumber="222">
```



```
<Description>The Harder They Come</Description>
  <Part Id="953562951413" UnitPrice="22.95" Quantity="1"/>
</LineItem>
UPDATE purchaseorder
  SET OBJECT VALUE =
      XMLQuery('copy $i := $p modify
                  delete nodes $i/PurchaseOrder/LineItems/LineItem[@ItemNumber="222"]
                return $i'
               PASSING OBJECT VALUE AS "p" RETURNING CONTENT)
      WHERE XMLExists('$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.

Creating XML Views of Modified XML Data

You can use XQuery Update to create new views of XML data.

Example 5-40 creates a view of table purchaseorder.

Example 5-40 Creating a View Using Updated XML Data

```
CREATE OR REPLACE VIEW purchaseorder summary OF XMLType AS
  SELECT XMLQuery('copy $i := $p1 modify
                    ((for $j in $i/PurchaseOrder/Actions
                     return replace value of node $j with ()),
                     (for $j in $i/PurchaseOrder/ShippingInstructions
                     return replace value of node $j with ()),
                     (for $j in $i/PurchaseOrder/LineItems
                     return replace value of node $j with ()))
                  return $i'
                 PASSING OBJECT VALUE AS "p1" RETURNING CONTENT)
   FROM purchaseorder p;
SELECT OBJECT VALUE FROM purchaseorder summary
  WHERE XMLExists('$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>
```

Performance Tuning for XQuery

A SQL query that involves XQuery expressions can often be automatically rewritten (optimized) in one or more ways. This optimization is referred to as **XML query rewrite** or optimization. When this happens, the XQuery expression is, in effect, evaluated directly against the XML document without constructing a DOM in memory.

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 all of the following:

- Single-pass streaming of XMLType data stored as binary XML A set of XPath expressions is evaluated in a single scan of the data.
- XMLIndex optimizations A SQL statement that uses an XPath expression is rewritten to an equivalent SQL statement that does not use it but which instead references the relational XMLIndex tables. The rewritten SQL statement can also make use of any B-tree indexes on the underlying XMLIndex tables.
- Optimizations for XMLType data stored object-relationally and for XMLType views A SQL statement that uses an XPath expression is rewritten to an equivalent SQL statement that does not use it but which instead references the object-relational or relational data structures that underly the XMLType data. The rewritten SQL statement can also make use of any B-tree indexes on the underlying data structures. This can take place for both queries and update operations.

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 object-relational storage: XPath Rewrite for Object-Relational Storage
- For compact schema-aware binary XML and transportable binary XML storage: Indexes for 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:

- 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 >>)

Topics in this section present execution plans for some of the examples shown in XQuery and Oracle XML DB, to indicate how they are executed.

- 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.
- XQuery Optimization over Relational Data

Use of SQL/XML functions XMLQuery and XMLTable over relational data can be optimized. Examples are included that use 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 Use of SQL/XML functions XMLQuery and XMLTable XML Schema-based data can be optimized. Examples are included that use XQuery expressions that target an XML schema-based XMLType table stored object-relationally.
- Diagnosis of 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.
- Performance Improvement for fn:doc and fn:collection on Repository Data You can improve the performance of fn:doc and fn:collection queries over the Oracle XML DB Repository, by linking them to the actual database tables that hold the repository data being queried.

Related Topics

 Oracle XML DB Support for XQuery Oracle XML DB support for the XQuery language includes SQL support and support for XQuery functions and operators.

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

Use of SQL/XML functions XMLQuery and XMLTable over relational data can be optimized. Examples are included that use XQuery expressions that target XML data created on the fly using fn:collection together with URI scheme oradb.

Example 5-41 shows the optimization of XMLQuery over relational data accessed as XML. Example 5-42 shows the optimization of XMLTable in the same context.

Example 5-41 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.

PLAN_TABLE_OUTPUT

Plan hash value: 3341889589

								 					-
	Id	I	Operation		Name	I	Rows	Bytes	I	Cost	(%CPU)	Time	
	0		SELECT STATEMENT				1	 		2	2 (0)	00:00:01	
	1		SORT AGGREGATE	L			1	41	I		1		
	2		TABLE ACCESS BY INDEX ROWID		LOCATIONS		1	41		1	(0)	00:00:01	
*	3		INDEX UNIQUE SCAN	L	LOC ID PK		1		T	C) (0)	00:00:01	
	4		SORT AGGREGATE	L			1	6	I		1		
1	5		TABLE ACCESS FULL	L	WAREHOUSES		9	54	T	2	2 (0)	00:00:01	
	6		FAST DUAL	I		I	1		I	2	2 (0)	00:00:01	

Predicate Information (identified by operation id):

3 - access("LOCATION ID"=:B1)

18 rows selected.

Example 5-42 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.

PLAN TABLE OUTPUT

```
------
```

Plan hash value: 1021775546

:	Id		Operation	Name		Rows		Bytes		Cost	(%CPU)	Time	
	() L	SELECT STATEMENT SORT AGGREGATE			9		54 41		2	(0)	00:00:01	
İ	2	2	TABLE ACCESS BY INDEX ROWID	LOCATIONS	İ	1	i	41	i	1	(0)	00:00:01	i
*	3	3	INDEX UNIQUE SCAN	LOC_ID_PK	I	1	Ι			0	(0)	00:00:01	I
	4	1	TABLE ACCESS FULL	WAREHOUSES		9	I	54	I	2	(0)	00:00:01	Ι

Predicate Information (identified by operation id):

```
3 - access("LOCATION_ID"=:B1)
```

16 rows selected.

XQuery Optimization over XML Schema-Based XMLType Data

Use of SQL/XML functions XMLQuery and XMLTable XML Schema-based data can be optimized. Examples are included that use XQuery expressions that target an XML schema-based XMLType table stored object-relationally.

Example 5-43 shows the optimization of XMLQuery over an XML schema-based XMLType table. Example 5-44 shows the optimization of XMLTable in the same context.

Example 5-43 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	
Ι		0		SELECT STATEMENT	I			1	I	530	I	5	(0)	00:00:01	I
		1		SORT AGGREGATE				1							
*		2		FILTER											
		3		FAST DUAL	1		Ι	1				2	(0)	00:00:01	
*		4		TABLE ACCESS	FULL	PURCHASEORDER	Ι	1		530	I	5	(0)	00:00:01	

Predicate Information (identified by operation id):



22 rows selected.

Example 5-44 Optimization of XMLTable with Schema-Based XMLType Data

Here again is the query of Example 5-14, 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 (%C	PU)	Time	
0 1 2 * 3 * 4 * 5	SELECT STATEMENT NESTED LOOPS NESTED LOOPS TABLE ACCESS FULL INDEX RANGE SCAN TABLE ACCESS BY INDEX ROWID	PURCHASEORDER SYS_C005478 LINEITEM_TABLE	 	4 4 1 17 3	 	384 384 37 177	 	7 7 5 1 2	 (0) (0) (0)	00:00:01 00:00:01 00:00:01 00:00:01 00:00:01	

Predicate Information (identified by operation id):

"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 purchaseorder 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.

Assuming that this has been done, the following statement creates the appropriate index:

CREATE INDEX unitprice index ON lineitem table("PART"."UNITPRICE");

With this index defined, the query of Example 5-14 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
      |

      1*
      2
      INDEX UNIQUE SCAN
      SYS_IOT_TOP_49323
      3
      564
      5
      (0)
      00:00:01
      |

      1*
      3
      INDEX RANGE SCAN
      UNITPRICE_INDEX
      20
      |
      2
      (0)
      00:00:01
      |

      1*
      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.
```

Diagnosis of 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:

- 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.
- OCI users can use OCIStmtExecute or event 19201. Only the event is available to Java users.

See Also:

Turning Off Use of XMLIndex for information about optimizer hint NO XML QUERY REWRITE

Performance Improvement for fn:doc and fn:collection on Repository Data

You can improve the performance of fn:doc and fn:collection queries over the Oracle XML DB Repository, by linking them to the actual database tables that hold the repository data being queried.

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.

- Use EQUALS_PATH and UNDER_PATH Instead of fn:doc and fn:collection Using Oracle SQL functions equals_path and under_path instead of XQuery functions fn:doc and fn:collection can improve performance.
- Using Oracle XQuery Pragma ora:defaultTable You can use Oracle XQuery extension-expression pragma ora:defaultTable to improve the performance of querying repository data.

Use EQUALS_PATH and UNDER_PATH Instead of fn:doc and fn:collection

Using Oracle SQL functions equals_path and under_path instead of XQuery functions fn:doc and fn:collection can improve performance.

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 5-45 and Example 5-46 illustrate this for functions fn:doc and equals_path; functions fn:collection and under_path are treated similarly.

Example 5-45 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 5-46 Optimized Repository Query Using EQUALS_PATH

¹ 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

You can use Oracle XQuery extension-expression pragma ora:defaultTable to improve the performance of querying repository data.

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 guery manually to use an explicit join and equals path or under path. Example 5-47 illustrates this; the query is rewritten automatically to what is shown in Example 5-46.

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.

Repository Query Using Oracle XQuery Pragma ora:defaultTable Example 5-47

```
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 Sval}'
                RETURNING CONTENT)
```

FROM DUAL;



6 Indexes for 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. There are various ways that you can index XMLType data, whether it is XML schema-based or non-schema-based, and regardless of the XMLType storage model you use.

Note:

Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes. Oracle recommends that you recreate unstructured XML indexes as XML search indexes and use it alongside Transportable Binary XML.

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.

- Oracle XML DB Tasks Involving Indexes Common tasks involving indexes for XML data are described.
- Overview of Indexing XMLType Data
- XMLIndex
- XML Search Index: Indexing for Full Text Search and Ad-hoc Queries
 When you need full-text search, or range-search capabilities over large, unstructured
 documents stored as Transportable Binary XML, Oracle recommends that you use XML
 Search Index to index the XML data.
- Indexing XMLType Data Stored Object-Relationally You can effectively index XMLType data that is stored object-relationally by creating B-tree indexes on the underlying database columns that correspond to XML nodes.

See Also:

- Oracle Database Concepts for an overview of indexing
- Oracle Database Development Guide for information about using indexes in application development

Oracle XML DB Tasks Involving Indexes

Common tasks involving indexes for XML data are described.

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
Choose an indexing approach	Overview of Indexing XMLType Data
Create, drop, or rename an XMLIndex index	Example 6-6, Example 6-8
Obtain the name of an XMLIndex index for a given table or column	Example 6-7
Determine whether a given XMLIndex index is used in evaluating a query	How to Tell Whether XMLIndex is Used
Turn off use of an XMLIndex index	Turning Off Use of XMLIndex
Creating and Using an XML Search Index	Creating and Using an XML Search Index
Queries using an XML Search Index	Queries using an XML Search Index
Index XMLType data stored object-relationally	Indexing XMLType Data Stored Object- Relationally, Guideline: Create indexes on ordered collection tables

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 6-11, Example 6-9				
Drop the structured component of an XMLIndex index (drop all structure groups)	Example 6-13				
Ensure data type correspondence between a query and an XMLIndex index with a structured component	Data Type Considerations for XMLIndex Structured Component				
Create a B-tree index on a content table of an XMLIndex structured component	Example 6-14				
Create an Oracle Text CONTEXT index on a content table of an XMLIndex structured component	Example D-6				

Table 6-3 identifies the documentation for some other user tasks involving XMLIndex indexes.

Table 6-3 Tasks involving XML Search Index

For information about	See
How to create and use an XML Search Index	Creating and Using an XML Search Index
How to maintain XML Search Indexes	Maintenance of XML Search Indexes
Default Preferences and Recommendations	Preference Defaults and Recommendations
How to query using an XML Search Index	Queries using an XML Search Index
Index Migration	Migrating to Use XML Search Index

Overview of Indexing XMLType Data

If your XML data contains islands of structured, predictable data, and your application only projects values from these islands of structured content:



- Use XMLIndex with a structured component to index the structured islands.
- A structured index component reflects the queries you use. You can change this set of known queries over time, provided you update the index definition accordingly. See XMLIndex Structured Component.

In addition, if you need to query content using full-text search within your XML data:

Use an XML search index. See Queries using an XML Search Index.

Does your XML data contain islands of data that is highly structured and predictable ?

- If Yes: Use XMLIndex with a structured component to index the islands. See XMLIndex Structured Component.
- If No: Do you need to support ad-hoc XML queries that involve predicates?
 - If Yes: Use XML Search Index which can be customized to support both Full Text Search and Range-Search Predicates.
 - If No: Do not index your XML data.

Note:

XMLIndex and XML Search Index can both be created over the same XMLType column. See section Queries using an XML Search Index for more information about index usage during queries.

XMLIndex

Note:

Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes. Oracle recommends that you recreate unstructured XML indexes as XML search indexes and use it alongside Transportable Binary XML.

Advantages of XMLIndex

B-tree indexes can be used advantageously with object-relational XMLType 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 either transportable binary XML or binary XML. That is the special domain of XMLIndex.

XMLIndex Components

XMLIndex is used to index XML data that may have some structure. It applies to XMLType data that is stored as both transportable binary XML and binary XML.

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.



- Creating, Dropping, Altering, and Examining an XMLIndex Index Basic operations on an XMLIndex index include creating it, dropping it, altering it, and examining it. Examples are presented.
- Use of XMLIndex with a Structured Component An XMLIndex structured component indexes specific islands of structure in your XML data.
- How to Tell Whether XMLIndex is Used To know whether a particular XMLIndex index has been used in resolving a query, you can examine an execution plan for the query.
- Turning Off Use of XMLIndex You can turn off the use of XMLIndex by using optimizer hint: /*+ NO_XML_QUERY_REWRITE */ or optimizer hint /*+ NO_XMLINDEX_REWRITE */.
- Guidelines for Using XMLIndex with a Structured Component There are several guidelines that can help you use XMLIndex with a structured component.
- XMLIndex Partitioning and Parallelism If you partition an XMLType table, or a table with an XMLType column, using range, list, or hash partitioning, you can also create an XMLIndex index on the table. You can optionally ensure that index creation and maintenance are carried out in parallel.
- 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.
- 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.
- PARAMETERS Clause for CREATE INDEX and ALTER INDEX Creation or modification of an XMLIndex index often involves the use of a PARAMETERS clause with SQL statement CREATE INDEX or ALTER INDEX. You can use it to specify index characteristics in detail.

Advantages of XMLIndex

B-tree indexes can be used advantageously with object-relational XMLType 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 either transportable binary XML or binary XML. That is the special domain 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.
- You can use an XMLIndex index with either XML schema-based or non-schema-based XMLType data stored as binary XML. B-tree indexing is appropriate only for XML schema-based data that is stored object-relationally.



- 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.
- If you have prior knowledge of the XPath expressions to be used in queries, then you can improve performance 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 indexes, which are deprecated.

XMLIndex Components

XMLIndex is used to index XML data that may have some structure. It applies to XMLType data that is stored as both transportable binary XML and binary XML.

Note:

Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes. Oracle recommends that you recreate unstructured XML indexes as XML search indexes and use it alongside Transportable Binary XML.

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. A typical example is a free-form specification with fixed fields author, date, and title. 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. It is appropriate for general indexing of document-centric XML data. A typical example is an XML web document or a book chapter.

You can create an XMLIndex index with both structured and unstructured components. A typical use case is supporting queries that extract an XML fragment from a document whenever some structured data is also present. The unstructured component is used for the fragment extraction. The structured component is used for a query predicate that checks for the structured data (for example, in the SQL WHERE clause).

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. 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.



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 two columns). And an XMLIndex index with a structured component is useful for document-centric data that contains structured islands.

	Data Centric	Document Centric
Use Case	XML schema-based data, with little variation and little structural change over time	 Variable, free-form data, with some fixed embedded structures Variable, free-form data
Typical Data	Employee Record	 Technical article, with author, date, and title fields Web document or book chapter
Storage Model	Object-Relational(Structured)	Transportable Binary XML
Indexing	B-Tree Index	 XMLIndex index with structured components XML search index

Table 6-4 XML Use Cases and XML Indexing

Related Topics

 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.

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.

See Also:

Advantages of XMLIndex 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 an XMLIndex index with a structured component 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 automatically rewritten to use the relational indexing tables of an XMLIndex structured component. In particular, SQL ORDER BY, GROUP BY, and window constructs operating on columns of an XMLTable virtual table are rewritten to the same constructs operating on the real columns of the relational indexing tables of the structured XMLIndex component.

The relational tables used for XMLIndex structured indexing 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 11g 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.

- 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).
 - Exchange Partitioning and XMLIndex In exchange partitioning, you exchange a table with a partition of another table. The first table must have the same structure as the partition of the second table, with which it is to be exchanged. The two tables must also be similar with respect to indexing with an XMLIndex index.

Related Topics

•

- Use of XMLIndex with a Structured Component An XMLIndex structured component indexes specific islands of structure in your XML data.
- SQL/XML Functions XMLQUERY, XMLTABLE, XMLExists, and XMLCast 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.

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.

Related Topics

 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.

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.



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						

Table 6-5 XML and SQL Data Type Correspondence for XMLIndex

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.

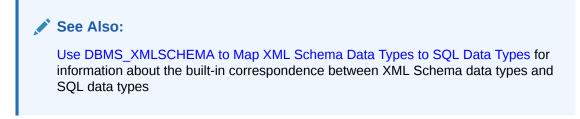
If the XML and SQL data types involved do not have a built-in one-to-one correspondence, then you must make them correspond (according to Table 6-5), in order for the index to be picked up for your query. There are two ways you can do this:

- 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 6-1 and Example 6-2 show how you can cast an XQuery expression in your query to match the SQL data type used in the index content table.

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 6-1, 25 is a SQL number of data type INTEGER; in Example 6-2, 25 is an XQuery number of data type xs:decimal.

In Example 6-1, the XMLQuery result is cast to SQL type INTEGER, which is compared with the SQL value 25. In Example 6-2, 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.



```
Example 6-1 Making Query Data Compatible with Index Data – SQL Cast
```

Example 6-2 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");
```

Exchange Partitioning and XMLIndex

In exchange partitioning, you exchange a table with a partition of another table. The first table must have the same structure as the partition of the second table, with which it is to be exchanged. The two tables must also be similar with respect to indexing with an XMLIndex index.

One of the following must be true:

- Neither table has an XMLIndex index.
- Both have an XMLIndex index, and one of the following is true:
 - Neither index has a structured component.
 - Both indexes have a structured component.

If none of those conditions holds then you cannot perform exchange partitioning.

If both tables have an XMLIndex index with a structured component then in the general case you must perform some preprocessing before invoking ALTER TABLE EXCHANGE PARTITION, and you must perform some postprocessing after invoking it. Otherwise, the exchange-partition operation raises an error.

You use PL/SQL procedures exchangePreProc and exchangePostProc in package DBMS_XMLSTORAGE_MANAGE to perform this preprocessing and postprocessing, as illustrated in Example 6-3. Each of the XMLType tables there, table and exchange_table, has an XMLIndex index that has a structured component.

In the special case of *reference*-partitioned tables there are foreign-key constraints involved, so things are a bit more complex. In this case, you use PL/SQL procedure **refPartitionExchangeIn** or **refPartitionExchangeOut**, to load data into (exchange-in) or out of (exchange-out) the partitioned tables, respectively.

Example 6-4 illustrates this, loading data from exchange tables parent_ex and child_ex into base tables parent and child. Example 6-5 shows the table and index definitions.

See Also:

- Oracle Database SQL Language Reference
- Oracle Database Data Cartridge Developer's Guide for general information about using ALTER TABLE EXCHANGE PARTITION with tables that have domain indexes (XMLIndex is a domain index)
- Oracle Database PL/SQL Packages and Types Reference for information about procedures exchangePreProc, exchangePostProc, refPartitionExchangeIn, and refPartitionExchangeIOut in package DBMS XMLSTORAGE MANAGE.

Example 6-3 Exchange-Partitioning Tables That Have an XMLIndex Structured Component

```
EXEC DBMS_XMLSTORAGE_MANAGE.exchangePreProc(USER, 'table');
EXEC DBMS_XMLSTORAGE_MANAGE.exchangePreProc(USER, 'exchange_table');
```

ALTER TABLE table EXCHANGE PARTITION partition WITH TABLE exchange_table WITH VALIDATION UPDATE INDEXES;

EXEC DBMS_XMLSTORAGE_MANAGE.exchangePostProc(USER, 'table'); EXEC DBMS_XMLSTORAGE_MANAGE.exchangePostProc(USER, 'exchange table');

Example 6-4 Exchange-Partitioning Reference-Partitioned Tables That Use XMLIndex

In this example:

- parent is the partitioned base table.
- **child** is a *reference*-partitioned child table with XMLType column xcol.
- **child_xidx** is an XMLIndex index with a structured component, defined on column xcol of table child. This is a local index, which is partitioned.
- parent ex is the exchange table for base table parent.
- child ex is the exchange table for child table child.
- **child_xidx_ex** is an XMLIndex index with a structured component, defined on column xcol of table child ex. This is *not* a local index (unlike the case for index child xidx).
- USER is the owner (database schema) of the tables.

This example performs an exchange-in operation, loading data from the exchange tables into the partitioned tables. An exchange-out operations, which loads data out of the partitioned tables into the exchange tables, would look the same, except that it would use procedure refPartitionExchangeOut instead. The procedure is passed the relevant tables and the necessary ALTER TABLE ... EXCHANGE statements.

```
EXEC DBMS_XMLSTORAGE_MANAGE.refPartitionExchangeIn(
    USER, 'parent', 'child', 'parent_ex', 'child_ex',
    'ALTER TABLE parent EXCHANGE PARTITION part_all WITH TABLE parent_ex
    INCLUDING INDEXES WITH VALIDATION UPDATE INDEXES',
    'ALTER TABLE child EXCHANGE PARTITION part_all WITH TABLE child_ex
    INCLUDING INDEXES WITH VALIDATION UPDATE INDEXES');
```



Example 6-5 Data Used in Example of Exchange-Partitioning for Reference-Partitioned Tables

This example shows the creation operations for the tables and indexes used in Example 6-4.

NUMBER PRIMARY KEY, CREATE TABLE parent (id created DATE) PARTITION BY RANGE (created) (PARTITION part 2014 VALUES LESS THAN (to date('01-jan-2015', 'dd-mon-yyyy')), PARTITION part all VALUES LESS THAN (maxvalue)); CREATE TABLE child (parent id NUMBER NOT NULL, xcol XMLType, CONSTRAINT child tab fk FOREIGN KEY (parent id) **REFERENCES** parent (id) ENABLE VALIDATE) XMLType COLUMN xcol STORE AS BINARY XML PARTITION BY REFERENCE (child tab fk); CREATE INDEX child xidx ON child p (xcol) INDEXTYPE IS XDB.XMLIndex PARAMETERS ('XMLTable po index tab ''purchaseorder'' COLUMNS pid NUMBER(4) PATH ''@id''') LOCAL ; CREATE TABLE parent ex (id NUMBER PRIMARY KEY, created DATE); CREATE TABLE child_ex (parent id NUMBER NOT NULL, xcol XMLType, CONSTRAINT child_tab_fk1 FOREIGN KEY (parent_id) **REFERENCES** parent ex(id) ENABLE VALIDATE) XMLType COLUMN xcol STORE AS BINARY XML; CREATE INDEX child_ex_xidx ON child_ex p (xcol) INDEXTYPE IS XDB.XMLIndex PARAMETERS ('XMLTable po index tab ex ''purchaseorder'' COLUMNS pid NUMBER(4) PATH ''@id''');

Creating, Dropping, Altering, and Examining an XMLIndex Index

Basic operations on an XMLIndex index include creating it, dropping it, altering it, and examining it. Examples are presented.

Note:

Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes. Oracle recommends that you recreate unstructured XML indexes as XML search indexes.

You create an XMLIndex index by declaring the index type to be XDB.XMLIndex, as illustrated in Example 6-6.



This creates an XMLIndex index named po_xmlindex_ix on XMLType table po_binxml. 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.

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.

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.

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 C-6.

See Also:

- structured_clause ::=
- path_table_clause ::=
- drop_path_table_clause ::=
- alter_index_group_clause ::=

Example 6-6 Creating an XMLIndex Index

```
CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex;
```

Example 6-7 Obtaining the Name of an XMLIndex Index on a Particular Table

```
SELECT INDEX_NAME FROM USER_INDEXES
  WHERE TABLE_NAME = 'PO_BINXML' AND ITYP_NAME = 'XMLINDEX';
INDEX_NAME
PO XMLINDEX IX
```



1 row selected.

Example 6-8 Renaming and Dropping an XMLIndex Index

ALTER INDEX po xmlindex ix RENAME TO new name ix;

DROP INDEX new name ix;

Related Topics

• PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX The syntax for the PARAMETERS clause for CREATE INDEX and ALTER INDEX is defined.

Use of XMLIndex with a Structured Component

An XMLIndex structured component indexes specific islands of structure in your XML data.

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* ::=.

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. To simplify the creation of such an index, you can use PL/SQL function DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView to provide exactly the XMLTable expression needed for creating the index.

- Using Namespaces and Storage Clauses with an XMLIndex Structured Component When you create an XMLIndex index that has a structured component you can specify XML namespaces and storage options to use.
- Adding a Structured Component to an XMLIndex Index You can use ALTER INDEX to add a structured component to an existing XMLIndex index.
- Using Non-Blocking ALTER INDEX with an XMLIndex Structured Component You can prevent ALTER INDEX from blocking when you add a group or column for the structured component of an XMLIndex index, so that queries that use the index do not need to wait.
- Modifying the Data Type of a Structured XMLIndex Component
 If an error is raised because some of your data does not match the data type used for the
 corresponding column of the structured XMLIndex component, you can in some cases
 simply modify the index by passing keyword MODIFY COLUMN TYPE to ALTER INDEX.

- Dropping an XMLIndex Structured Component 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.
 - Indexing the Relational Tables of a Structured XMLIndex Component 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.

Related Topics

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.

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).

See Also:

- Indexing Binary XML Data Exposed Using a Relational View for information about using DBMS XMLSTORAGE MANAGE.getSIDXDefFromView
- Indexing XML Data for Full-Text Queries (pre-23ai)
- structured_clause ::=
- Usage of XMLIndex_xmltable_clause for information about an XMLType column in an XMLTable clause
- Usage of column_clause for information about keywords COLUMNS and VIRTUAL

Using Namespaces and Storage Clauses with an XMLIndex Structured Component

When you create an XMLIndex index that has a structured component you can specify XML namespaces and storage options to use.

Example 6-9 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 USERTBS1. The example assumes a binary XML table po_binxml with non XML schema-based data.

Each of the (identical) TABLESPACE clauses in Example 6-9 applies at the table level (tables po ptab and li tab).

In general you can specify storage options at both the table level and the partition level. A specification at the partition level overrides one at the table level. A TABLESPACE clause can also be specified at the *index* level, that is, so that it applies to all of the partitions and tables used for the index. If TABLESPACE is specified at more than one level, the partition level overrides the table level, which overrides the index level.

Example 6-10 specifies the same TABLESPACE for each of the tables used in the index. This commonality can be factored out by specifying the TABLESPACE at the index level, as shown in Example 6-10.



Chapter 6 XMLIndex

Example 6-9 XMLIndex with a Structured Component, Using Namespaces and Storage Options

```
CREATE INDEX po struct ON po binxml (OBJECT VALUE) INDEXTYPE IS XDB.XMLIndex
 PARAMETERS ('XMLTable po ptab
               (TABLESPACE "USERTBS1" 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 "USERTBS1" 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''');
```

Example 6-10 XMLIndex with a Structured Component, Specifying TABLESPACE at the Index Level

```
CREATE INDEX po_struct ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex

PARAMETERS ('XMLTable po_ptab,

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,

XMLTable li_tab,

XMLNAMESPACES (DEFAULT ''http://www.example.com/po''),

''/item'' PASSING items

COLUMNS partnum VARCHAR2(15) PATH ''@partNum'',

description CLOB PATH ''uSPrice',

shipdat DATE PATH ''uSPrice'',

shipdat DATE PATH ''shipDate''

TABLESPACE "USERTBS1" COMPRESS FOR OLTP)');
```

Adding a Structured Component to an XMLIndex Index

You can use ALTER INDEX to add a structured component to an existing XMLIndex index.

Example 6-11 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 6-11 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.

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 6-11 also illustrates the use of a registered parameter string in the PARAMETERS clause. It uses PL/SQL procedure DBMS_XMLINDEX.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.

Example 6-11 XMLIndex Index: Adding a Structured Component

```
CREATE INDEX po xmlindex ix ON po binxml (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');

Using Non-Blocking ALTER INDEX with an XMLIndex Structured Component

You can prevent ALTER INDEX from blocking when you add a group or column for the structured component of an XMLIndex index, so that queries that use the index do not need to wait.

When you use ALTER INDEX to add a group or a column for the structured component of an XMLIndex index, this index-maintenance operation obtains an exclusive DDL lock on the base table and the index.

The base table is locked to DML operations, and the index cannot be used for queries until the ALTER INDEX operation is finished. This means that during this index maintenance the index cannot be used by other sessions that query or perform DML operations on the base table. The duration of the ALTER INDEX operation and the attendant locking depends on the volume of data in the base XMLType column.

You can avoid or work around this problem as follows:

1. Use keyword **NONBLOCKING** before ADD_GROUP or ADD_COLUMN in the PARAMETERS clause of the ALTER INDEX statement that creates the structured-component group or column.



This updates the index as needed, but it does not index any base-table data. Because it does not depend on the base-table data it is quick regardless of the base-table size.

Invoke PL/SQL procedure DBMS_XMLINDEX.process_pending.

This procedure indexes rows of the base table and populates tables of the index, just as if keyword NONBLOCKING were absent. However, in this case only a few rows are locked at a time while they are processed and the changes committed. Rows that have already been locked for some other purpose are skipped. This can significantly reduce lock contention and allow indexing of some rows to proceed at the same time as querying or DML on other rows.

When procedure process pending finishes it returns, as OUT parameters:

• The number of rows that it could not index. This is either because they were *locked* for another purpose or because an error was raised (this number includes the number returned as the other OUT parameter).

After you think those locks have been removed, invoke procedure process_pending again to try to process those pending rows.

• The number of rows that it could not index because an *error* was raised. (This should be rare.)

Check table SYS_AIXSXI_index_number_ERRORTAB for information about those errors, then take action to fix the underlying problems. *index_number* is the object number of the index.

 Repeat step 2 as many times as necessary until procedure process_pending indicates that all rows have been successfully indexed or you encounter an insurmountable problem and decide to cancel the indexing operation altogether.

You can cancel the indexing at any time (before step 2) by using keywords **NONBLOCKING ABORT** in the PARAMETERS clause of a separate ALTER INDEX statement for the same XMLIndex index.

4. If all rows have been successfully indexed then use keywords **NONBLOCKING COMPLETE** in the PARAMETERS clause of a separate ALTER INDEX statement for the same XMLIndex index.

Example 6-12 illustrates this.

Just as table SYS_AIXSXI_index_number_ERRORTAB reports errors, so table SYS_AIXSXI_index_number_PENDINGTAB records the current status of each base-table row: whether or not it has been indexed. A row might not yet be indexed because it is locked by for some other purpose or because trying to index it raised an error. In the latter case, consult SYS_AIXSXI_index_number_ERRORTAB for specific information about the error.

See Also: alter_index_group_clause ::=

Example 6-12 Using DBMS_XMLINDEX.PROCESS_PENDING To Index XML Data

```
CREATE INDEX po_struct ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex

PARAMETERS ('XMLTable po_idx_tab

''/PurchaseOrder''

COLUMNS reference VARCHAR2(30) PATH ''Reference'',

requestor VARCHAR2(30) PATH ''Requestor'',

username VARCHAR2(30) PATH ''User'',
```



```
PATH ''LineItems/LineItem'' VIRTUAL
                                       XMLType
                          lineitem
                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''');
ALTER INDEX po struct
  PARAMETERS ('NONBLOCKING ADD GROUP GROUP po action group
              XMLTABLE po idx tab
                 ''/PurchaseOrder''
                 COLUMNS actions
                                       XMLType PATH ''Actions/Action'' VIRTUAL
               XMLTABLE po idx action
                 ''/Action'' PASSING actions
                 COLUMNS actioned by VARCHAR2(10) PATH ''User'',
                         date actioned TIMESTAMP PATH ''Date''');
DECLARE
  num pending NUMBER := 0;
  num errored NUMBER := 0;
BEGIN
 DBMS XMLINDEX.process pending ('oe', 'po struct', num pending, num errored);
 DBMS OUTPUT.put line('Number of rows still pending = ' || num pending);
 DBMS OUTPUT.put line ('Number of rows with errors = ' || num errored);
END;
/
Number of rows still pending = 0
Number of rows with errors
                              = 0
PL/SQL procedure successfully completed.
ALTER INDEX po struct PARAMETERS('NONBLOCKING COMPLETE');
```

Modifying the Data Type of a Structured XMLIndex Component

If an error is raised because some of your data does not match the data type used for the corresponding column of the structured XMLIndex component, you can in some cases simply modify the index by passing keyword MODIFY COLUMN TYPE to ALTER INDEX.

You can, for example, expand a VARCHAR2 (30) column to, say, VARCHAR2 (40) if it needs to accommodate data that is up to 40 characters. This is simpler and more efficient than dropping the column and then adding a new column. The new data type must be compatible with the old one: the same restrictions apply as apply for ALTER TABLE MODIFY COLUMN.

See Also:

- Oracle Database SQL Language Reference for information about ALTER TABLE
 MODIFY COLUMN
- modify_column_type_clause :==

Dropping an XMLIndex Structured Component

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 6-13 shows how to drop the structured component that was added in Example 6-11, by dropping its only structure group, po item.

Example 6-13 Dropping an XMLIndex Structured Component

ALTER INDEX po xmlindex ix PARAMETERS('DROP GROUP GROUP po item');

Indexing the Relational Tables of a Structured XMLIndex Component

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.

This is explained in section XMLIndex Structured Component. It is illustrated by Example 6-14, which creates a B-tree index on the reference column of the index content table (structured fragment) for the XMLIndex index of Example 6-11.

Example 6-14 Creating a B-tree Index on an XMLIndex Index Content Table

CREATE INDEX idx_tab_ref_ix ON po_idx_tab (reference);

How to Tell Whether XMLIndex is Used

To know whether a particular XMLIndex index has been used in resolving a query, you can examine an execution plan for the query.

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 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.

You can examine the execution plan for a query to see whether a particular XMLIndex index has been used in resolving the query.

• 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 6-15.

See Also:

Oracle Database SQL Tuning Guide

CREATE INDEX po_struct ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex PARAMETERS ('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''');
```

Example 6-15 shows an execution plan that indicates that the same XMLIndex index is also picked up for a query that uses multilevel XMLTable chaining.

The execution plan shows direct access to the relational index content tables, po_idx_tab and po index lineitem.

Example 6-15 Using a Structured XMLIndex Component for a Query with Multilevel Chaining

```
EXPLAIN PLAN FOR

SELECT po.reference, li.*

FROM po_binxml 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';
```

			Operation (%CPU) Time 	Name	R	ows]	Bytes
- 			SELECT STATEMENT	I	I	17 3	2036
		8	(0) 00:00:01				
	1		NESTED LOOPS			1	
	2		NESTED LOOPS		1	17 3	2036
		8	(0) 00:00:01				
	3		NESTED LOOPS		I	1	53
		3	(0) 00:00:01				
*	4		TABLE ACCESS FULL	PO_IDX_TAB	1	1	52
		3	(0) 00:00:01				
*	5		INDEX UNIQUE SCAN	SYS C007442		1	1



```
0
       (0) | 00:00:01 |
|* 6 | INDEX RANGE SCAN | SYS86751_86759_PKY_IDX | 17 |
1
   1 (0) | 00:00:01 |
| 7 | TABLE ACCESS BY INDEX ROWID| PO INDEX LINEITEM | 17 | 11203
    5 (0)| 00:00:01 |
 _____
Predicate Information (identified by operation id):
  4 - filter("SYS SXI 2"."REFERENCE"='SBELL-20021009123335280PDT')
  5 - access("P"."SYS_NC_OID$"="SYS_SXI_2"."OID")
  6 - access("SYS SXI 2"."KEY"="SYS SXI 3"."PKEY")
Note
  - dynamic sampling used for this statement
25 rows selected.
```

Related Topics

 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.

Turning Off Use of XMLIndex

You can turn off the use of XMLIndex by using optimizer hint: /*+ NO_XML_QUERY_REWRITE */ or optimizer hint /*+ NO_XMLINDEX REWRITE */.

Each of these hints turns off the use of *all* XMLIndex indexes. In addition to turning off the use of XMLIndex, NO_XML_QUERY_REWRITE turns off all XQuery optimization (XMLIndex is part of XPath rewrite).

Example 6-16 shows the use of these optimizer hints.

Note:

The NO INDEX optimizer hint does not apply to XMLIndex.

See Also:

XQuery Optional Features for information about XQuery pragmas ora:no_xmlquery_rewrite and ora:xmlquery_rewrite, which you can use for finegrained control of XQuery optimization

Example 6-16 Turning Off XMLIndex Using Optimizer Hints

```
SELECT /*+ NO_XMLINDEX_REWRITE */
count(*) FROM po_binxml WHERE XMLExists('$p/*' PASSING OBJECT_VALUE AS "p");
SELECT /*+ NO_XML_QUERY_REWRITE */
```

```
count(*) FROM po binxml WHERE XMLExists('$p/*' PASSING OBJECT VALUE AS "p");
```

Guidelines for Using XMLIndex with a Structured Component

There are several guidelines that can help you use 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.
- Ensure data type correspondence between a query and an XMLIndex index that has a structured component. See Data Type Considerations for XMLIndex Structured Component.
- If you create a relational view over XMLType data (for example, using SQL function XMLTable), then consider also creating an XMLIndex index with a structured component that targets the same relational columns. See Relational Views over XML Data.
- 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.

XMLIndex Partitioning and Parallelism

If you partition an XMLType table, or a table with an XMLType column, using range, list, or hash partitioning, you can also create an XMLIndex index on the table. You can optionally ensure that index creation and maintenance are carried out in parallel.

To ensure parallel index creation and maintenance, you use a PARALLEL clause (with optional degree) when creating or altering an XMLIndex index.

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 composite-partition a table, then you cannot create an XMLIndex index on it.

If you use a PARALLEL clause and 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 6-17 creates an XMLIndex index with a parallelism degree of 10. If the base table is partitioned, then this index is equipartitioned.

In Example 6-17, 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 6-18 demonstrates this. Again, because of keyword LOCAL, if the base table is partitioned, then this index is equipartitioned.

In Example 6-18, 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.

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

Example 6-17 Creating an XMLIndex Index in Parallel

```
CREATE INDEX po_xmlindex_ix ON sale_info (sale_po_clob)
INDEXTYPE IS XDB.XMLIndex
LOCAL PARALLEL 10;
```

Example 6-18 Using Different PARALLEL Degrees for XMLIndex Internal Objects

CREATE INDEX po_xmlindex_ix ON sale_info (sale_po_clob) INDEXTYPE IS XDB.XMLIndex LOCAL NOPARALLEL PARAMETERS ('PATH TABLE po_path_table (PARALLEL 10) PIKEY INDEX po_pikey_ix VALUE INDEX po_value_ix (PARALLEL 5)');

Related Topics

- PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX The syntax for the PARAMETERS clause for CREATE INDEX and ALTER INDEX is defined.
- XMLIndex Components XMLIndex is used to index XML data that may have some structure. It applies to XMLType data that is stored as both transportable binary XML and binary XML.



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.

Note:

The following applies only to procedures in package DBMS_STATS; it does not apply to ANALYZE INDEX.

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 6-19 collects statistics on the base table po_binxml. Statistics are automatically collected on the XMLIndex index, its path table, and the secondary path-table indexes.

See Also:

Data Dictionary Static Public Views Related to XMLIndex for information about database views that record statistics information for an XMLIndex index

Example 6-19 Automatic Collection of Statistics on XMLIndex Objects

CALL DBMS_STATS.gather_table_stats(USER, 'PO_BINXML', ESTIMATE_PERCENT => NULL);

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 6-6 describes the columns in each of these views.

Table 6-6 XMLIndex Static Public Views

Column Name	Туре	Description
ASYNC	VARCHAR2	Asynchronous index updating specification. See Asynchronous (Deferred) Maintenance of XMLIndex Indexes.



Column Name	Туре	Description		
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 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.		
		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.		
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.		
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.		

Table 6-6 (Cont.) XMLIndex Static Public Views

These views provide information about an XMLIndex index, but there is no single static data dictionary 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 can be used to identify the tables conforming an XMLIndex index. To retrieve statistics information on these tables, query using the name of the XMLTable table as TABLE NAME.
- 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

Creation or modification of an XMLIndex index often involves the use of a PARAMETERS clause with SQL statement CREATE INDEX or ALTER INDEX. You can use it to specify index characteristics in detail.

You can use PL/SQL procedures registerParameter and modifyParameter in package DBMS XMLINDEX to bypass the 1000-character PARAMETERS clause limit.

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.

- PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX The syntax for the PARAMETERS clause for CREATE INDEX and ALTER INDEX is defined.
- 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
 Certain considerations apply to using XMLIndex_parameters.
- Usage of groups_clause and alter_index_group_clause Clause groups_clause is used only with CREATE INDEX (or following ADD GROUP in clause alter_index_group_clause). Clause alter_index_group_clause is used only with ALTER INDEX.
- Usage of XMLIndex_xmltable_clause Certain considerations apply to using XMLIndex xmltable clause.
- Usage of column_clause Certain considerations apply to using *column* clause.



🖋 See Also:

- PARAMETERS Clause for CREATE INDEX and ALTER INDEX in Unstructured Index
- 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.

See Also: Example 6-11

PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX

The syntax for the PARAMETERS clause for CREATE INDEX and ALTER INDEX is defined.

XMLIndex_parameters_clause ::=





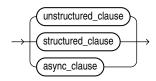
See Also: Usage of XMLIndex_parameters_clause

XMLIndex_parameters ::=

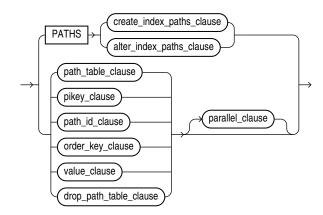




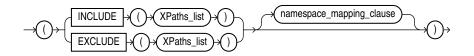
XMLIndex_parameter_clause ::=



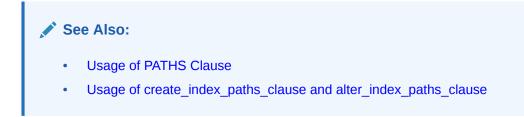
unstructured_clause ::=



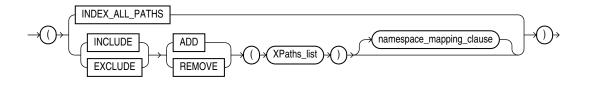
create_index_paths_clause ::=





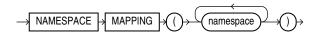


alter_index_paths_clause ::=

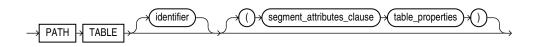


See Also:
Usage of PATHS Clause
Usage of create index paths clause and alter index paths clause

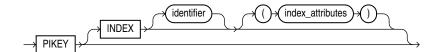
namespace_mapping_clause ::=



path_table_clause ::=



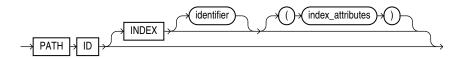
pikey_clause ::=

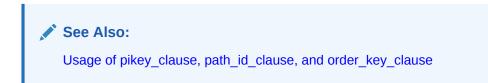


See Also: Usage of pikey_clause, path_id_clause, and order_key_clause



path_id_clause ::=



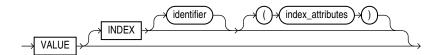


order_key_clause ::=



See Also: Usage of pikey_clause, path_id_clause, and order_key_clause

value_clause ::=

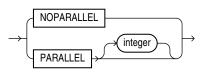




drop_path_table_clause ::=

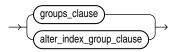


parallel_clause ::=





structured_clause ::=



See Also: Usage of groups_clause and alter_index_group_clause

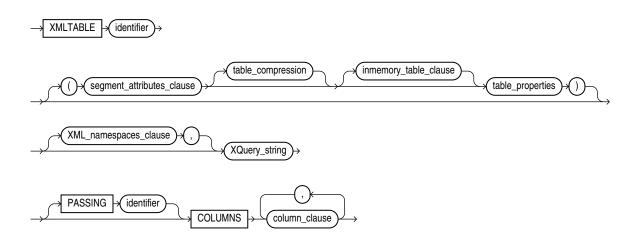






See Also: Usage of groups_clause and alter_index_group_clause

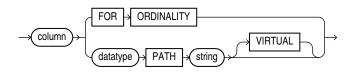
XMLIndex_xmltable_clause ::=



Syntax elements XML_namespaces_clause and XQuery_string are the same as for SQL/XML function XMLTable.



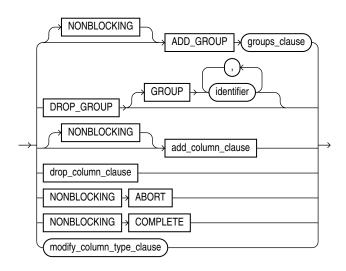
column_clause ::=



Syntax element column_clause is similar, but not identical, to XML_table_column in SQL/XML function XMLTable.

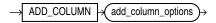


alter_index_group_clause ::=



See Also: Usage of groups_clause and alter_index_group_clause

add_column_clause :==



add_column_options :==

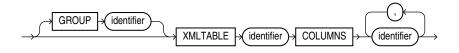


Syntax element XML_namespaces_clause is the same as for SQL/XML function XMLTable. See XMLTABLE SQL/XML Function in Oracle XML DB.

drop_column_clause :==



drop_column_options :==





modify_column_type_clause :==

modify_column_type_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.

Note:

Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes. Oracle recommends that you recreate unstructured XML indexes as XML search indexes and use it alongside Transportable Binary XML.

See Also:

- Oracle Database SQL Language Reference for information about the use context for XMLIndex_parameters_clause in CREATE INDEX
- Oracle Database SQL Language Reference for information about the use context for XMLIndex_parameters_clause in ALTER INDEX

Usage of XMLIndex_parameters

Certain considerations apply to using XMLIndex_parameters.

Note:

Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes. Oracle recommends that you recreate unstructured XML indexes as XML search indexes and use it alongside Transportable Binary XML.



- 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 groups_clause and alter_index_group_clause

Clause groups_clause is used only with CREATE INDEX (or following ADD GROUP in clause alter_index_group_clause). Clause alter_index_group_clause is used only with ALTER INDEX.

Usage of XMLIndex_xmltable_clause

Certain considerations apply to using XMLIndex_xmltable_clause.

- The XQuery_string expression in XMLIndex_xmltable_clause must not use the XQuery functions ora:view (desupported), 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

Certain considerations apply to using *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.

XML Search Index: Indexing for Full Text Search and Ad-hoc Queries

When you need full-text search, or range-search capabilities over large, unstructured documents stored as Transportable Binary XML, Oracle recommends that you use XML Search Index to index the XML data.

The XMLExists expression supports querying XMLType data that is stored as Transportable Binary XML with XQuery Full Text (XQFT) predicates. If you use an XQFT full-text predicate in



an XMLExists expression within a SQL WHERE clause, then you must create an XML Search Index. This section describes the creation and use of such an index.

Oracle recommends that you store your data in the Transportable Binary XML format. When your data is not using TBX storage, you can still index XML data by creating an XQuery Full Text CONTEXT index or an Oracle Text Index. For more information on how to create an XQuery Full Text CONTEXT index, see Indexing XML Data for Full-Text Queries (pre-23ai).

Creating and Using an XML Search Index

To create an XML search index, specify the FOR XML clause in the CREATE SEARCH INDEX statement. You can create an XML search index only on a column with the SYS.XMLType data type that stores documents using the TRANSPORTABLE BINARY XML (TBX) storage option, which is the 23ai default.

- Maintenance of XML Search Indexes XML Search Indexes are asynchronous indexes. New documents are not immediately reflected by the index until the new documents are synchronized.
- Preference Defaults and Recommendations
- Queries using an XML Search Index
- Migrating to Use XML Search Index XML Search Indexes can only be used when the XMLType is stored as Transportable Binary XML.

See Also:

- Oracle Text Reference for more information about CONTEXT indexes and Oracle
 Text Search indexes.
- Indexing XML Data for Full-Text Queries (pre-23ai) for information on how to index XML Data using XQuery Full Text Indexes.

Creating and Using an XML Search Index

To create an XML search index, specify the FOR XML clause in the CREATE SEARCH INDEX statement. You can create an XML search index only on a column with the SYS.XMLType data type that stores documents using the TRANSPORTABLE BINARY XML (TBX) storage option, which is the 23ai default.

An XQuery Full Text query can use an XML search index to improve performance. To create an XML Search Index you must be granted database role CTXAPP. More generally, this role is needed to create and maintain Oracle Text indexes, to set Oracle Text index preferences, or to use Oracle Text PL/SQL packages.

To create an XML Search Index, specify the FOR XML clause in the CREATE SEARCH INDEX statement. You can create an XML search index only on a column with the SYS.XMLType data type that stores documents using the TRANSPORTABLE BINARY XML (TBX) storage option.

If you run the CREATE SEARCH INDEX command without the FOR XML clause on a column that is of type SYS.XMLTYPE the type of the created index will depend on the storage option of the column. When the storage option of the column is TBX, and XML Search Index is created. Otherwise, an Oracle Text index is created.

CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (document);



You may use the FOR XML clause explicitly if you have ensured that the storage option of the column is TBX.

CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML;

When creating an XML Search Index, you must decide the types of queries that you expect to take advantage of the index.

Type of Query	SEARCH_ON argument				
XQuery Full Text	TEXT				
Range-search on Numbers	VALUE (BINARY_FLOAT) or VALUE (NUMBER)				
Range-search on Timestamp	VALUE (TIMESTAMP)				
Range-search on Keywords	VALUE (VARCHAR2)				

If you will only use XQuery Full Text predicates in an XMLExists operator, then specify SEARCH_ON TEXT on the index's parameter string. This is the default if no SEARCH_ON clause is specified:

CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML PARAMETERS ('SEARCH ON TEXT');

Multiple SEARCH ON clauses can be specified as part of the PARAMETERS clause:

CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML PARAMETERS ('SEARCH_ON VALUE(BINARY_DOUBLE) SEARCH_ON VALUE(VARCHAR2)');

Equivalently, it can be combined into a single clause as below:

CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML PARAMETERS ('SEARCH ON VALUE(BINARY DOUBLE, VARCHAR2)');

Note:

You cannot specify the same data type multiple times.

If you plan on enabling the index for both range-search and full-text search queries, then specify SEARCH ON TEXT VALUE.

CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML PARAMETERS ('SEARCH ON TEXT VALUE(BINARY DOUBLE, VARCHAR2, TIMESTAMP)');

You can index XML data that is not stored using the TBX option by creating an XQuery Full Text CONTEXT index.

Syntax for XML Search Index:

```
CREATE SEARCH INDEX [schema.]index ON [schema.]table(xml_column)
FOR XML
[LOCAL]
PARAMETERS(
[SEARCH_ON (TEXT | TEXT_VALUE(data_types) | VALUE(data_types))]
[STORAGE storage_pref]
[MEMORY memsize]
[SYNC (MANUAL | EVERY "interval-string" | ON COMMIT)]
```



```
[MAINTENANCE AUTO | MAINTENANCE MANUAL]
  [OPTIMIZE (MANUAL | EVERY "interval-string" | AUTO DAILY)]
[PARALLEL N]
[UNUSABLE];
```

where:

)

- [schema.] index specifies the name of the XML search index to create.
- [schema.]table(xml column) specifies the names of table and column to index. xml column is the name of the column on which the index is created.

See Also:

Oracle Text Reference for more information about these parameters.

Maintenance of XML Search Indexes

XML Search Indexes are asynchronous indexes. New documents are not immediately reflected by the index until the new documents are synchronized.

When maintaining an XML Search Index, the index will synchronize unindexed documents in the background in a system-controlled manner by default. The index synchronization can also be triggered on demand by invoking procedure CTX DDL.SYNC INDEX.

Index synchronization can be tuned to happen on transaction commit or periodically after at least an interval of time has passed. To change the method of index synchronization, specify the SYNC option to:

- SYNC (ON COMMIT) The index will be synchronized during transaction commit.
- SYNC (EVERY "interval information") The index will be synchronized periodically in the background by verifying if there are documents pending indexing.

To create an XML Search Index that synchronizes during commit time:

CREATE SEARCH INDEX xml ft idx ON xmldoctab (tbx document) FOR XML PARAMETERS ('SYNC (ON COMMIT) SEARCH ON TEXT VALUE(BINARY DOUBLE, VARCHAR2)');

To create an XML Search Index that synchronizes every two hours:

```
CREATE SEARCH INDEX xml ft idx ON xmldoctab (tbx document)
FOR XML PARAMETERS ('SYNC (EVERY "FREQ=HOURLY; INTERVAL=2") SEARCH ON
TEXT VALUE (BINARY DOUBLE, VARCHAR2)');
```

See Also:

Oracle Text Reference for details on index synchronization and maintenance

Preference Defaults and Recommendations



When you create an XML Search Index with no Oracle Text preferences, the following preferences are enabled by default:

Preference	Attribute	Value
N/A	MAINTENANCE	AUTO
BASIC_STORAGE	STAGE_ITAB	TRUE
BASIC_STORAGE	STAGE_ITAB_MAX_ROWS	10000
BASIC_STORAGE	STAGE_ITAB_AUTO_OPT	TRUE
BASIC_STORAGE	XML_SAVE_COPY	TRUE

You can specify your own Text index preferences when creating an XML Search Index. Oracle recommends that the following preferences are used:

- D_TABLE_CLAUSE Specify SECUREFILE storage for column DOC of index data table \$D, which contains information about the structure of your XML documents. Specify caching and medium compression.
- I_TABLE_CLAUSE Specify SECUREFILE storage for column TOKEN_INFO of index data table \$I, which contains information about full-text tokens and their occurrences in the indexed documents. Specify caching (but not compression).
- STAGE_ITAB Specify TRUE to enable the staging table feature. The staging table decreases the amount of fragmentation in the index after maintenance operations by merging the contents of the staging table with the index only after a specific row threshold is reached. Data in the staging table is still part of the index.
- STAGE_ITAB_MAX_ROWS Specify a value of 10000 or more. This value should be tuned according to how often the index will be maintained due to DML. Larger values will merge the contents of the staging table less often while keeping the staging table fragmented.
- STAGE_ITAB_AUTO_OPT Specify TRUE to allow the merging of the staging table to happen in the background.
- STAGE_ITAB_PARALLEL Specify a parallel degree when merging the staging table. By default, the merge operation will use parallel degree 4.

You can use the interfaces defined in the CTX_DDL package and the PARAMETERS clause to specify non-default values for these options:

END;



```
/
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML PARAMETERS ('STORAGE
xml_sto SEARCH_ON TEXT_VALUE(NUMBER, VARCHAR2)');

Note:
   Oracle Text Reference for information about procedure
   CTX_DDL.set_sec_grp_attr
   Oracle Text Reference for information about procedure
   CTX_DDL.create_preference
   Oracle Text Reference for information about procedure
   CTX_DDL.set_set_preference
   Oracle Text Reference for information about procedure
   CTX_DDL.set_attribute
```

• Oracle Text Reference for information about preference BASIC_STORAGE, D_TABLE_CLAUSE, and I_TABLE_CLAUSE

Queries using an XML Search Index

An XML Search Index is considered for usage when there are XMLExists predicates in your query with the following operators:

- XQuery Full Text Predicate Operators
 - CONTAINS TEXT
 - CONTAINS TEXT ... FTAND
 - CONTAINS TEXT ... FTOR
 - CONTAINS TEXT ... FTAND FTNOT
 - CONTAINS TEXT ... WINDOW n WORDS
- Range Search Predicates over scalar values of numeric, datatime, or string data type using relational operators.

Note:

Which predicates inside an XMLExists operator pick up the index are determined by the index's setting for SEARCH_ON. If you use XQuery Full Text Predicate Operators for an index that is not Full-Text enabled, then compile time error ORA-18177 will be raised.

The following restrictions need to be considered when writing queries to pick up an XML Search Index:

- The expression that computes the XML nodes for the search context must be an XPath expression whose steps are only along forward and descendent axes.
- You can pass only one XMLType instance as a SQL expression in the PASSING clause of SQL/XML function XMLExists, and each of the other, non-XMLType SQL expressions in



that clause must be either a compile-time constant of a SQL built-in data type or a bind variable that is bound to an instance of such a data type.

Example 6-20 XQuery Full Text Query

Example 6-21 Execution Plan for XQuery Full Text Query

- Id 		Operation		Nai	me		Rows		Bytes		Cost	(%CPU)	Time
- 0	Ι	SELECT STATEMENT	I			Ι	1		2014	I	4	(0)	00:00:01
1	I	TABLE ACCESS BY	INDEX ROWID	PO	BINXML		1	Ι	2014	I	4	(0)	00:00:01
* 2 	I	DOMAIN INDEX	I	PO	_CTX_IDX	Ι					4	(0)	00:00:01

Predicate Information (identified by operation id):

```
2 - access("CTXSYS"."CONTAINS"(SYS_MAKEXML(0,"XMLDATA"),'<query><textquery
grammar="CONTEXT" lang="english"> ( ( {Big} ) and ( {Street} ) ) INPATH
(/PurchaseOrder/LineItems/LineItem/Description)</textquery></query>')>0)
```

Here are some more full text query examples:

Retrieving records where description contains text "Julius"

```
select id FROM po_binxml
where xmlexists('/PurchaseOrder/LineItems/LineItem/Description[. contains text
"Julius"]' PASSING doc);
```

Retrieving records where address contains text "Oracle" or "Austin"

```
select id FROM po_binxml where
xmlexists('/PurchaseOrder/ShippingInstructions/address[. contains text "Oracle" ftor
"Austin"]' PASSING doc);
```

In addition to the restrictions for any query that will use an XML Search Index, when using range-search predicates in your query, the following restrictions are applied:



- The comparison value must be a compile time constant or a bind variable that was bound prior to executing the cursor.
- The data type of the compile time constant or bind variable must be enabled in the index.
- The relational predicate must not be !=.
- The predicate must not be negated by using the NOT().

To ensure that the correct data type is picked up for the index, data type constructors like xs:double, xs:decimal, or xs:dateTime can be used to allow the correct data type to be inferred.

Example 6-22 A Range Search Query in an XMLExists predicate answered by an XML Search Index

This query returns User and Reference of Purchase orders that contain at least one item that is cheaper than \$15.00.

Example 6-23 Execution Plan for a Range Search Query in an XMLExists predicate answered by an XML Search Index

Id		Operation	Name	
0 1 * 2 3 4 5 * 6 7		SELECT STATEMENT SORT AGGREGATE FILTER FAST DUAL NESTED LOOPS TABLE ACCESS BY INDEX ROWID DOMAIN INDEX XPATH EVALUATION	XTAB XSIDX	

Predicate Information (identified by operation id):

Retrieving orders after Feb 5

```
SELECT id from xtab where
    xmlexists('/PurchaseOrder[@orderDate > xs:dateTime("2024-02-05")]' PASSING
    doc);
```

Retrieving orders with Description > "Light"



```
SELECT id from xtab where
    xmlexists('/PurchaseOrder/LineItems/LineItem[Description > "Lava"]' PASSING
    doc);
```

Which index is used on a query?

When the only index on an XMLType column is an XML Search Index, queries will attempt to use the XML Search Index by default.

However, when there is an Structured XMLIndex on the same column as the XML Search Index, usage of the Structured XMLIndex will take priority over the XML Search index.

XQuery Full Text queries will continue to pick up the XML Search Index, meanwhile, rangesearch queries will use the Structured XMLIndex instead.

This behavior can be overridden by using the ora:use_xmltext_idx pragma to force the XML Search Index to be picked up over the Structured XML Index.

```
SELECT XMLQuery('for $i in /PurchaseOrder
        return <Details>{$i/User}{$i/Reference}</Details>'
        PASSING doc RETURNING CONTENT)
FROM xtab
WHERE XMLExists('(#ora:use_xmltext_idx #)){/PurchaseOrder/LineItems/LineItem/
Part[UnitPrice < 15.0]}'
        PASSING doc);
```

Furthermore, the NO INDEX hint can be used to avoid considering the XML Search Index at all.

Migrating to Use XML Search Index

XML Search Indexes can only be used when the XMLType is stored as Transportable Binary XML.

If you have legacy queries for XMLType data stored as Non Transportable Binary XML that use SQL function CONTAINS and an Oracle Text index that is not XML-enabled the data can be migrated to use Transportable Binary XML Storage and use an XML Search Index.

Furthermore, if you have legacy queries for XMLType data stored as Non Transportable Binary XML alongside an XQuery Full Text Context index, you can also migrate this data to use Transportable Binary XML and use XML Search Index.

Oracle recommends that you use XML Search Index for indexing of unstructured sections of XML documents. XML Search Indexes allow for much greater flexibility in data partitioning. CONTEXT indexes and XQuery Full Text indexes can only be locally partitioned over range-partitioned tables.

When migrating from a CONTEXT index to an XML Search Index, the usage of the CONTAINS function should be replaced with XMLEXISTS predicates.

When migrating from an XQFT index to an XML Search Index, the ora:use_xmltext_idx pragma used within some of the XMLExists predicates is only required to force the XML Search Index to be picked up when there is an Structured XML Index on the same column.

Migrating from using Oracle Text Index to XML Search Index

The table below provides examples of typical queries using Oracle Text specific operators and their replacement when using XML Search Index:



Original Examp	le	Replacement Example				
<pre>CONTAINS(t.x, > 0</pre>	'HASPATH (/P/LIs/LI/Description)')	XMLExists('\$d/P/LIs/LI/Description}' PASSING t.x AS "d")				
CONTAINS(t.x,	'Big INPATH (/P/LIs/LI/Description)') >	XMLExists('\$d/P/LIs/LI/Description [. contains text "Big"]' PASSING t.x AS "d")				
CONTAINS(t.x,	'(Big) AND (Street) INPATH (/P/LIs/LI/Description)') >	<pre>XMLExists('\$d/P/LIs/LI/Description [. contains text "Big" ftand "Street"]' PASSING t.x AS "d")</pre>				
CONTAINS(t.x,	'(Big) OR (Street) INPATH (/P/LIs/LI/Description)') >	<pre>XMLExists('\$d/P/LIs/LI/Description [. contains text "Big" ftor "Street"]' PASSING t.x AS "d")</pre>				
CONTAINS(t.x, INPATH 0	<pre>'({Big}) NOT ({Street}) (/P/LIs/LI/Description)') ></pre>	XMLExists('\$d/P/LIs/LI/Description [. contains text "Big" ftand ftnot "Street"]' PASSING t.x AS "d")				
CONTAINS(t.x, Street}) INPAT 0	'({Street}) MNOT ({Big H (/P/LIs/LI/Description)') >	<pre>XMLExists('\$d/P/LIs/LI/Description [. contains text "Street" not in "Big Street"]' PASSING t.x AS "d")</pre>				
CONTAINS(t.x, 3) INPATH 0	<pre>'(NEAR (({Big}, {Street}), (/P/LIs/LI/Description)') ></pre>	<pre>XMLExists('\$d/P/LIs/LI/Description [. contains text "Big" ftand "Street" window 3 words]' PASSING t.x AS "d")</pre>				
(Not applicable – namespace awa	- Oracle Text queries are not XML re.)	<pre>XMLExists('declare namespace</pre>				

The path test can contain a predicate expression, which is the same for both the original query (with HASPATH) and its replacement. For example:

/PurchaseOrder/LineItems/LineItem/Part[@Id < "31415927"]</pre>

Migrating from using XQuery Full Text Index to XML Search Index

The main purpose of an XQuery Full Text Index is to efficiently search data in XML documents using XQuery Full Text predicates.

Creating an XQuery Full Text Index requires then the creation of a Section Group and Storage preferences to configure the underlying CONTEXT index.

The equivalent index creation statement when using XML Search Index is:

CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML PARAMETESRS ('SEARCH_ON TEXT');

Indexing XMLType Data Stored Object-Relationally

You can effectively index XMLType data that is stored object-relationally 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.

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 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.

Indexing Repeating (Collection) Elements

In XMLType data stored object-relationally, a collection is stored as an ordered collection table (OCT) of an XMLType instance, which means that you can directly access its members. Because object-relational storage directly reflects the fine-grained structure of the XML data, you can create indexes that target individual collection members.

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 6-24 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 XPath node test text().)

Example 6-25 ostensibly tries to create a function-based index using (deprecated) Oracle SQL function extractValue, targeting the same data.

In reality, in both Example 6-24 and Example 6-25 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 19-7 and Example 19-8 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 11g 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.

Example 6-24 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 6-25 CREATE INDEX Using EXTRACTVALUE on a Singleton Element

CREATE INDEX po_reference_ix ON purchaseorder
 (extractValue(OBJECT VALUE, '/PurchaseOrder/Reference'));

Indexing Repeating (Collection) Elements

In XMLType data stored object-relationally, a collection is stored as an ordered collection table (OCT) of an XMLType instance, which means that you can directly access its members. Because object-relational storage 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 <code>extractValue</code> does not apply to collections (the XPath expression passed to <code>extractValue</code> 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 for an example of how to do this.



7

Transformation and Validation of XMLType Data

There are several Oracle SQL functions and XMLType APIs for transforming XMLType data using XSLT stylesheets and for validating XMLType instances against an XML schema.

- XSL Transformation and Oracle XML DB You can apply XSL transformations to XML Schema-based documents using the built-in Oracle XML DB XSLT processor. In-database XML-specific optimizations can significantly reduce the memory required, eliminate the overhead associated with parsing, and reduce network traffic.
- Validation of XMLType Instances
 Besides needing to know whether a particular XML document is well-formed, you often
 need to know whether it conforms to a given XML schema, that is, whether it is valid with
 respect to that XML schema.

XSL Transformation and Oracle XML DB

You can apply XSL transformations to XML Schema-based documents using the built-in Oracle XML DB XSLT processor. In-database XML-specific optimizations can significantly reduce the memory required, eliminate the overhead associated with parsing, and reduce network traffic.

The W3C XSLT Recommendation defines an XML language for specifying how to transform XML documents from one form to another. See XSL Transformations (XSLT) Version 1.0 for information about the XSLT standard.

Transformation can include mapping from one XML schema to another or mapping from XML to some other format such as HTML or WML.

XSL transformation can be costly in terms of the amount of memory and processing required. In typical XSL processors, the entire source document and stylesheet must be parsed and loaded into memory, before processing can begin. Typically, XSL processors use DOM to provide dynamic memory representations of document and stylesheet, to allow random access to their different parts. The XSL processor then applies the stylesheet to the source document, generating a third document.

Parsing and loading the document and stylesheet into memory before beginning transformation requires significant memory and processor resources. It is especially inefficient when only a small part of the document needs to be transformed.

Oracle XML DB includes an XSLT processor that performs XSL transformations *inside the database*. In this way, it can provide XML-specific optimizations that can significantly reduce the memory required to perform the transformation, eliminate overhead associated with parsing, and reduce network traffic.

These optimizations are available, however, *only* when the source for the transformation is a *schema-based* XML document. In that case, there is no need to parse before processing can begin. The Oracle XML DB lazily loaded virtual DOM loads content only on demand, as the nodes are accessed. This also reduces the memory required, because only parts of the document that need to be processed are loaded.



You can transform XML data in the following ways:

- In Oracle Database Using Oracle SQL function XMLtransform, XMLType method transform(), or PL/SQL package DBMS XSLPROCESSOR
- In the middle tier Using Oracle XML Developer's Kit transformation options, such as XSLT Processor for Java.

See Also:

- Oracle Database SQL Language Reference for information about SQL function XMLTransform
- PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) and Oracle Database PL/SQL Packages and Types Reference for information about PL/SQL package DBMS_XSLPROCESSOR
- Oracle XML Developer's Kit Programmer's Guide for information about XSLT Processor for Java

Each of these XML transformation methods takes as input a source XML document and an XSL stylesheet 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 data generated by a transformation using package DBMS_XSLPROCESSOR is XHTML data, which is both valid XML data and valid HTML data.

Example 7-1 shows part of an XSLT stylesheet, PurchaseOrder.xsl. The complete stylesheet is given in XSLT Stylesheet Example, PurchaseOrder.xsl.

These is nothing Oracle XML DB-specific about the stylesheet of Example 7-1. A stylesheet can be stored in an XMLType table or column or stored as non-schema-based XML data inside Oracle XML DB Repository.

Example 7-1 XSLT Stylesheet Example: PurchaseOrder.xsl

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<xsl:stylesheet version="1.0"</pre>
 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">
   <hr/>
 </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">
 <P>
          \langle B \rangle
           <FONT SIZE="+1" COLOR="#FF0000" FACE="Arial, Helvetica, sans-serif">Internal</FONT>
          </B>
        </P>
        . . .
      <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>
```

- SQL Function XMLTRANSFORM and XMLType Method TRANSFORM() SQL function XMLtransform transforms an XML document by using an XSLT stylesheet. It returns the processed output as XML, HTML, and so on, as specified by the stylesheet.
- XSL Transformation Using DBUri Servlet You can apply an XSL transformation to XML content that is generated by the DBUri servlet.

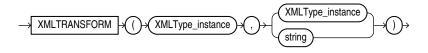
SQL Function XMLTRANSFORM and XMLType Method TRANSFORM()

SQL function XMLtransform transforms an XML document by using an XSLT stylesheet. It returns the processed output as XML, HTML, and so on, as specified by the stylesheet.

Figure 7-1 shows the syntax of Oracle SQL function XMLtransform. This function takes as arguments an XMLType instance and an XSLT stylesheet. The stylesheet can be an XMLType instance or a VARCHAR2 string literal. It applies the stylesheet to the instance and returns an XMLType instance.



Figure 7-1 XMLTRANSFORM Syntax



You can alternatively use XMLType method transform() as an alternative to Oracle SQL function XMLtransform. It has the same functionality.

Figure 7-2 shows how XMLtransform transforms an XML document by using an XSLT stylesheet. It returns the processed output as XML, HTML, and so on, as specified by the XSLT stylesheet. You typically use XMLtransform when retrieving or generating XML documents stored as XMLType in the database.



Figure 7-2 Using XMLTRANSFORM



• XMLTRANSFORM and XMLType.transform(): Examples

Examples illustrate how to use Oracle SQL function XMLtransform and XMLType method transform() to transform XML data stored as XMLType to various formats.

XMLTRANSFORM and XMLType.transform(): Examples

Examples illustrate how to use Oracle SQL function XMLtransform and XMLType method transform() to transform XML data stored as XMLType to various formats.

Example 7-2 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 raises an error.

Example 7-3 stores an XSLT stylesheet, then retrieves it and uses it with Oracle SQL function XMLTransform to transform the XML data stored in Example 7-2.

Example 7-4 uses XMLType method transform() with an XSL stylesheet created on the fly.

Example 7-5 uses XMLTransform to apply an XSL stylesheet to produce HTML code. PL/SQL constructor XDBURIType reads the XSL stylesheet from Oracle XML DB Repository.

Only part of the HTML result is shown in Example 7-5. Omitted parts are indicated with an ellipsis (. . .). Figure 7-3 shows what the transformed result looks like in a Web browser.

Example 7-2 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"</pre>
                           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"</pre>
                                                    type="ipo:Address"/>
                         <element name="billTo"
                                                   type="ipo:Address"/>
                         <element ref="ipo:comment" minOccurs="0"/>
                                                    type="ipo:Items"/>
                         <element name="items"</pre>
                       </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"</pre>
                                                           type="decimal"/>
                               <element ref="ipo:comment" minOccurs="0"/>
                               <element name="shipDate"</pre>
                                                          type="date" minOccurs="0"/>
                             </sequence>
                             <attribute name="partNum" type="ipo:SKU" use="required"/>
                           </complexType>
                         </element>
                       </sequence>
                     </complexType>
                     <complexType name="Address">
                       <sequence>
                         <element name="name"</pre>
                                                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 purchase-order documents, and insert the 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 7-3 Using SQL Function XMLTRANSFORM to Apply an XSL Stylesheet



```
<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</pre>
                                                           select="text()"/>
                     </xsl:otherwise>
                   </xsl:choose>
                 </xsl:template>
               <xsl:template match="*" name="nested" priority="-1" mode="nested2">
                 <h>>
                   <!-- xsl:value-of select="count(child::*)"/ -->
                   <xsl:choose>
                     <xsl:when test="count(child::*) > 1">
                       <xsl:value-of select="name(.)"/>:<xsl:apply-templates</pre>
                                                           mode="nested2"/>
                     </xsl:when>
                     <xsl:otherwise>
                       <xsl:value-of select="name(.)"/>:<xsl:value-of</pre>
                                                           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
_____
<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>
```

Example 7-4 Using XMLType Method TRANSFORM() with a Transient XSL Stylesheet

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="*">
                    <t.d>
                      <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>
                    </xsl:template>
                  <xsl:template match="*" name="nested" priority="-1"</pre>
                                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;
```

Example 7-5 Using XMLTRANSFORM to Apply an XSL Stylesheet Retrieved Using XDBURIType

```
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>
   <t.r>
       <P>
         <B>
          <FONT SIZE="+1" COLOR="#FF0000" FACE="Arial, Helvetica,
              sans-serif">Internal</FONT>
         </B>
        </P>
        . . .
        <B>
         <FONT COLOR="#FF0000">
          <FONT SIZE="+1">Ship To</FONT>
         </FONT>
        </B>
        . . .
        <br/>
   <B>
    <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
   </B>
   <br/>
   <br/>
   . . .
   </FONT>
 </body>
</html>
```

1 row selected.

XSL Transformation Using DBUri Servlet

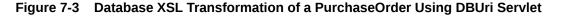
You can apply an XSL transformation to XML content that is generated by the DBUri servlet.

Figure 7-3 shows the result of such a transformation. The URL is the following (it is split and truncated here):

http://localhost:8080/oradb/SCOTT/PURCHASEORDER/ROW/PurchaseOrder[Reference="SBELL-2003030912333601PDT"]
contenttype=text/html&transform=/home/SCOTT/xsl/purchaseOrder.xsl...



The presence of parameter transform causes the DBUri servlet to use SQL function XMLTransform to apply the XSL stylesheet at /home/SCOTT/xsl/purchaseOrder.xsl to the PurchaseOrder document that is identified by the main URL. The result of the transformation, which is HTML code, is returned to the browser for display. The URL also uses parameter contentType to specify that the MIME-type of the final document is text/html.



http://localhost:	:8080/oradb/SCOTT/PURCHASEORDER/ROW,	/PurchaseOrder[Ref	erence="SB	ELL-20030309	11233 - Microsoft Internet Explorer	_ 8
ile Edit View	Favorites Tools Help					
🔁 Back 🔹 💮 🦂	- 💌 🛃 🏠 🔎 Search 🤸 Favorite	es 😢 Media 🧭	🍰 • 💐	2 🔜 🥃	😣 🗉 🚯 🚳	
idress 🙋 http://lo	calhost:8080/oradb/SCOTT/PURCHASEORDER/ROV	V/PurchaseOrder[Refer	ence="SBELL-2	20030309123336	601PDT"]?contenttype=text/html&transform=/home/SCOT💌 🛃	Go Links
oogle -	💌 💏 Search Web 👻 🚿	2 1330 blocked	🗧 AutoFill 🧧	🛐 🛛 🛃 Option	15 🌶	
		-				
		SBELL-200	30309123	33601PDT	r	
Internal				Ship Te		
				Name	Sarah J. Bell	
Actions	User Date				400 Oracle Parkway	
Actions	SVOLLMAN			Address	Redwood Shores CA 94065	
Requestor	Sarah J. Bell				USA	
User	KCHUNG			Telephone	e 650 506 7400	
Cost Center	S30					
ltemNumber	Description	PartId	Quantit∨	Unit Price	Total Price	
1	A Night to Remember	715515009058	2	39.95	79.90000000000006	
	The Unbearable Lightness Of Being	37429140222	2	29.95	59.899999999999999	
	The Wizard of Oz	715515011020	4	29.95	119.799999999999997	

Figure 7-4 shows table departments displayed as an HTML document. You need no code to achieve this. You need only an XMLType view based on SQL/XML functions, an industry-standard XSL stylesheet, and DBUri servlet.

Edit View Favorites				
Back 🝷 💮 🕤 🗾	🗿 🏠 🔎 Search 🤺 Favo	orites Media 🔾	8 8- 2 3	 🔇
				rm=/home/SCOTT/poSource/xsl/empdept 💟 툳
🙆 Aria 🛛 🙆 Customize Links	🗧 🙆 Free Hotmail 🛛 🙆 My Oracle	Network Request	🧉 Oracle CRM 🥌 Orac	:le Email 💰 Software 💰 Windows
	וח	EPARTME	NTS	
			N 1 5	
DEPARTME	INT LOCATION		EMPLOYEE	S
IT	2014 Jabberwocky Rd Southlake	Alexander Hunold		9000 03-JA17-90
	Texas 26192	Diana Lorentz		4200 07-FEB-99
	United States of	Valli Pataballa		4800 05-FEB-98
	America	David Austin		4800 25-JUN-97
		Bruce Ernst		6000 ^{21-MAY-} 91
Shipping	2011 Interiors Blvd South San Francisco	Kevin Mourgos		5800 ¹⁶⁻ NOV- 99
	California 99236 United States of	Shanta Vollman		6500 10-OCT- 97
	America	Payam Kaufling		7900 ^{01-MAY-} 95
				8200 10-APR-97
		D. Courts and NUT allow		8000 18-JUL-96

Figure 7-4 Database XSL Transformation of Departments Table Using DBUri Servlet

Validation of XMLType Instances

Besides needing to know whether a particular XML document is well-formed, you often 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.

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.
- Partial and Full XML Schema Validation When you insert XML Schema-based documents into the database they can be validated partially or fully.
- Validating XML Data Stored as XMLType: Examples
 Examples here 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.

🖍 See Also:

- 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()

Partial and Full XML Schema Validation

When you insert XML Schema-based documents into the database they can be validated partially or fully.

Partial Validation

For binary XML storage, Oracle XML DB validates XML Schema-based documents fully when they are inserted into an XMLType table or column. For object-relational XML storage, only a partial validation is performed. This is because complete XML schema validation is quite costly for object-relational storage, in terms of performance.



Full Validation

Loading XML Schema-based data into binary XML storage fully validates it against the target XML schemas. For object-relational XMLType storage, you can force full validation at any time.

Partial Validation

For binary XML storage, Oracle XML DB validates XML Schema-based documents fully when they are inserted into an XMLType table or column. For object-relational XML storage, only a partial validation is performed. This is because complete XML schema validation is quite costly for object-relational storage, 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 7-6 provides an example of failing partial validation while inserting an XML document into table PurchaseOrder, which is stored object-relationally.

Example 7-6 Error When Inserting Incorrect XML Document (Partial Validation)

Full Validation

Loading XML Schema-based data into binary XML storage fully validates it against the target XML schemas. For object-relational XMLType storage, you can force full validation at any time.

To force full validation, use 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 about *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.

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 then 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 7-7 shows how to force a full XML schema validation by adding a CHECK constraint to an XMLType table. In Example 7-7, 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 7-7 Forcing Full XML Schema Validation Using a CHECK Constraint

Pseudocolumn <code>OBJECT_VALUE</code> can be used to access the content of an <code>XMLType</code> table from within a trigger. Example 7-8 illustrates this, showing how to use a <code>BEFORE INSERT</code> trigger to validate that the data being inserted into the <code>XMLType</code> table conforms to the specified XML schema.

Example 7-8 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;
```



Validating XML Data Stored as XMLType: Examples

Examples here 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 7-9 and Example 7-10 show how to validate an XML instance against an XML schema using PL/SQL method isSchemaValid().

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 7-11 illustrates this.

Example 7-12 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

Note:

/

The validation functions and procedures described in Validation of XMLType Instances facilitate validation checking. Of these, schemaValidate is the only one that raises errors that indicate why validation has failed.

Example 7-9 Validating XML Using Method ISSCHEMAVALID() in SQL

FROM po_tab x;

Example 7-10 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.

Example 7-11 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 7-12 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";
```

Part III

Relational Data To and From XML Data

There are various ways that you can view existing relational data as XML data.

- Generation of XML Data from Relational Data Oracle XML DB provides features for generating (constructing) XML data from relational data in the database. There are both SQL/XML standard functions and Oracle-specific functions and packages for generating XML data from relational content.
- Relational Views over XML Data Relational database views over XML data provide conventional, relational access to XML content.
- XMLType Views

You can create XMLType views over relational and object-relational data.



Generation of XML Data from Relational Data

Oracle XML DB provides features for generating (constructing) XML data from relational data in the database. There are both SQL/XML standard functions and Oracle-specific functions and packages for generating XML data from relational content.

Overview of Generating XML Data

You can generate XML data using Oracle XML DB using standard SQL/XML functions, Oracle-specific SQL functions, PL/SQL subprograms from package DBMS_XMLGEN, or DBURIType.

- Generation of XML Data Using SQL Functions Oracle XML DB provides SQL functions that you can use to construct XML data. Most of these belong to the SQL/XML standard.
- Generation of XML Data 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.
- SYS_XMLAGG Oracle SQL Function

Oracle SQL function sys_XMLAgg aggregates all XML documents or fragments represented by an expression, producing a single XML document from them. It wraps the results of the expression in a new element named ROWSET (by default).

- 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.
- 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.

See Also:

XQuery and Oracle XML DB for information about constructing XML data using SQL/XML functions XMLQuery and XMLTable

Overview of Generating XML Data

You can generate XML data using Oracle XML DB using standard SQL/XML functions, Oraclespecific SQL functions, PL/SQL subprograms from package DBMS XMLGEN, or DBURITYPE.

Note:

The package DBMS XMLGEN is deprecated in Oracle Database 23ai.



- Use standard SQL/XML functions. See Generation of XML Data Using SQL Functions.
- Use Oracle SQL functions . See the following sections:
 - XMLCOLATTVAL Oracle SQL Function
 - XMLCDATA Oracle SQL Function
 - SYS_XMLAGG Oracle SQL Function. This operates on groups of rows, aggregating several XML documents into one.
- Use PL/SQL package DBMS XMLGEN. See Generation of XML Data Using DBMS_XMLGEN.
- Use a DBURIType instance to construct XML documents from database data. See Data Access Using URIs.

See Also:

•

- Overview of How To Use Oracle XML DB
- Transformation and Validation of XMLType Data
- PL/SQL APIs for XMLType
- Java DOM API for XMLType

Generation of XML Data Using SQL Functions

Oracle XML DB provides SQL functions that you can use to construct XML data. Most of these belong to the SQL/XML standard.

The standard XML-generation functions are also known as SQL/XML **publishing** or **generation** functions.

The use of SQL/XML function XMLQuery is not limited to generating (publishing) XML data. Function XMLQuery is very general and is referred to in this book as a SQL/XML **query and update** function.

The following XML-generating SQL functions are Oracle-specific (not part of the SQL/XML standard):

- XMLCOLATTVAL Oracle SQL Function.
- XMLCDATA Oracle SQL Function.
- SYS_XMLAGG Oracle SQL Function. 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.

- XMLELEMENT and XMLATTRIBUTES SQL/XML Functions SQL/XML standard function XMLElement constructs XML elements from relational data. SQL/XML standard function XMLAttributes can be used together with XMLElement, to specify attributes for the generated elements.
- XMLFOREST SQL/XML Function You use SQL/XML standard function XMLForest to construct a forest of XML elements.



•	XMLCONCAT SQL/XML Function You use SQL/XML standard function XMLConcat to construct an XML fragment by concatenating multiple XMLType instances.
•	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.
•	XMLPI SQL/XML Function You use SQL/XML standard function XMLPI to construct an XML processing instruction (PI).
•	XMLCOMMENT SQL/XML Function You use SQL/XML standard function XMLComment to construct an XML comment.
•	XMLSERIALIZE SQL/XML Function You use SQL/XML standard function XMLSerialize to obtain a string or LOB representation of XML data.
•	XMLPARSE SQL/XML Function You use SQL/XML standard function XMLParse to parse a string containing XML data and construct a corresponding XMLType instance.
•	XMLCOLATTVAL Oracle 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.
•	XMLCDATA Oracle SQL Function You use Oracle SQL function XMLCDATA to generate an XML CDATA section.
	See Also:
	 XQuery and 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

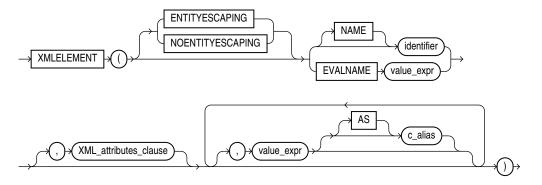
XMLELEMENT and XMLATTRIBUTES SQL/XML Functions

SQL/XML standard function XMLElement constructs XML elements from relational data. SQL/XML standard function XMLAttributes can be used together with XMLElement, to specify attributes for the generated elements.

SQL/XML standard function XMLElement takes as arguments an XML 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.







For an explanation of keywords ENTITYESCAPING and NOENTITYESCAPING, see Escape of Characters in Generated XML Data. These keywords are Oracle extensions to standard SQL/XML functions XMLElement and XMLAttributes.

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 8-1) 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 possibility of using EVALNAME is an Oracle extension to standard SQL/XML function XMLElement.

The optional *XML-attributes-clause* argument of function *XMLElement* specifies the attributes of the root element to be generated. Figure 8-2 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 then 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.

Note:

The AS preceding an alias (c_{alias}) is required by the SQL/XML standard, but is optional for Oracle.

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.



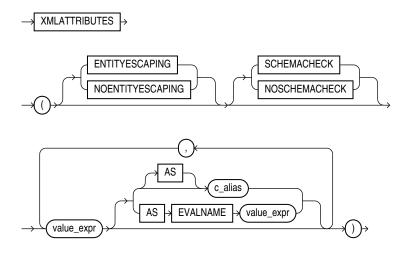


Figure 8-2 XMLAttributes Clause Syntax (XMLATTRIBUTES)

For an explanation of keywords ENTITYESCAPING and NOENTITYESCAPING, see Escape of Characters in Generated XML Data. These keywords are Oracle extensions to standard SQL/XML functions XMLElement and XMLAttributes.

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 12c Release 1 (12.1), the default behavior is to perform the check. Keyword SCHEMACHECK can be used to obtain backward compatibility.

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.

Keywords SCHEMACHECK and NOSCHEMACHECK are Oracle extensions to standard SQL/XML function XMLAttributes.

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 that is generated is not XML schema-based. If the XML schema is subsequently registered, then XML data that is 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.

Note:

The following are Oracle extensions to the standard SQL/XML syntax:

- The possibility of using EVALNAME.
- The fact that AS preceding an alias (*c* alias) is optional.

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.

- Escape of 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.
- 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.

• XMLElement Examples Examples here illustrate the use SQL/XML function XMLElement.

Escape of 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 8-1) and to attribute identifier aliases named with AS clauses of XMLAttributes (see Figure 8-2).

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. Keywords NOENTITYESCAPING and ENTITYESCAPING are Oracle extensions to standard SQL/XML functions XMLElement and XMLAttributes.



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 non-standard XML date or timestamp, use SQL function to char – see Example 8-1.

See Also:

XML Schema Part 2: Datatypes, D. ISO 8601 Date and Time Formats for the XML Schema specification of XML date and timestamp formats

XMLElement Examples

Examples here illustrate the use SQL/XML function XMLElement.

Example 8-1 uses XMLElement to generate an XML date with a format that is different from the XML Schema standard date format.

Example 8-2 uses XMLElement to generate an Emp element for each employee, with the employee name as the content.

Example 8-3 uses XMLElement to generate an Emp element for each employee, with child elements that provide the employee name and hire date.

Example 8-4 uses XMLElement to generate an Emp element for each employee, with attributes id **and** name.

As mentioned in Escape of Characters in Generated XML Data, 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 8-5 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.

A full description of character escaping is included in the SQL/XML standard.

Example 8-6 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 8-6 creates an XMLType instance conforming to that schema:



Example 8-7 uses XMLElement to generate an XML document with employee and department information, using data from sample database schema table hr.departments.

Example 8-1 XMLELEMENT: Formatting a Date

1 row selected.

Example 8-2 XMLELEMENT: Generating an Element for Each Employee

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 8-3 XMLELEMENT: Generating Nested XML

This query produces the following typical XML result:

RESULT <Emp><name>Michael Hartstein</name><hiredate>2004-02-17</hiredate></Emp>



```
<mme>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 8-4 XMLELEMENT: Generating Employee Elements with Attributes ID and Name

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.
```

Example 8-5 XMLELEMENT: Characters in Generated XML Data Are Not Escaped

This query produces the following result, which is not well-formed XML:

1 row selected.

Example 8-6 Creating a Schema-Based XML Document Using XMLELEMENT with Namespaces

This creates the following XML document that conforms to XML schema Employee.xsd. (The result is shown here pretty-printed, for clarity.)



```
1 row selected.
```

```
Example 8-7 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.
```

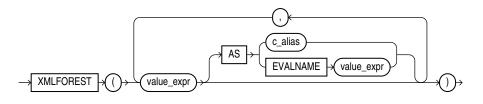


XMLFOREST SQL/XML Function

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 8-3 describes the XMLForest syntax.

Figure 8-3 XMLFOREST Syntax



Each of the value expressions (*value_expr* in Figure 8-3) 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). The possibility of using EVALNAME is an Oracle extension to standard SQL/XML function XMLForest.

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 Escape of Characters in Generated XML Data. If the value expression evaluates to NULL, then no element is created for that expression.

Example 8-8 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 8-8 XMLFOREST: Generating Elements with Attribute and Child Elements

(The WHERE clause is used here to keep the example brief.) This query produces the following XML result:



See Also: Example 8-19

Example 8-9 uses XMLForest to generate hierarchical XML data from user-defined data-type instances.

Example 8-9 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 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 might want to compare this example with Example 8-7 and Example 8-24.

XMLCONCAT SQL/XML Function

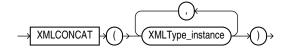
You use SQL/XML standard function XMLConcat to construct an XML fragment by concatenating multiple XMLType instances.

Figure 8-4 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 8-4 XMLCONCAT Syntax



Example 8-10 uses SQL/XML function XMLConcat to return a concatenation of XMLType instances from an XMLSequenceType value (a varray of XMLType instances).

Example 8-10 XMLCONCAT: Concatenating XMLType Instances from a Sequence

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 8-11 uses XMLConcat to create and concatenate XML elements for employee first and the last names.

Example 8-11 XMLCONCAT: Concatenating XML Elements

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>
```

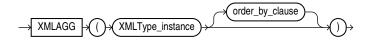
6 rows selected.

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 8-5 describes the XMLAgg syntax.

Figure 8-5 XMLAGG Syntax



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. SQL/XML function XMLAgg is similar to Oracle SQL function sys_XMLAgg, but XMLAgg returns a forest of nodes and it does not accept an XMLFormat parameter.

SQL/XML 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 8-12 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 8-12 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 8-13 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 8-13 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 8-14 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 8-14 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>
```

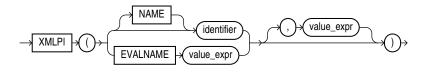
1 row selected.

XMLPI SQL/XML Function

You use SQL/XML standard function XMLPI to construct an XML processing instruction (PI).

Figure 8-6 shows the syntax:

Figure 8-6 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. The possibility of using EVALNAME is an Oracle extension to standard SQL/XML function XMLPI.

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 8-15 uses XMLPI to generate a simple processing instruction.

Example 8-15 Using SQL/XML Function XMLPI

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

This results in the following output:

XMLCOMMENT SQL/XML Function

You use SQL/XML standard function XMLComment to construct an XML comment.

Figure 8-7 shows the syntax:

Figure 8-7 XMLComment Syntax

XMLCOMMENT value_expr



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 8-16 uses XMLComment to generate a simple XML comment.

Example 8-16 Using SQL/XML Function XMLCOMMENT

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

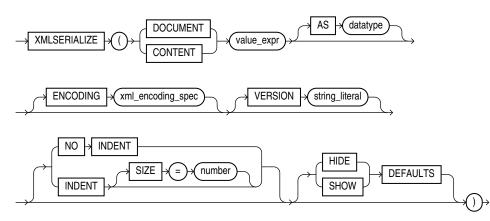
This query results in the following output:

XMLSERIALIZE SQL/XML Function

You use SQL/XML standard function XMLSerialize to obtain a string or LOB representation of XML data.

Figure 8-8 shows the syntax of XMLSerialize:





Argument *value_expr* is evaluated, and the resulting XMLType instance is serialized to produce the content of the created string or LOB. If present¹, the specified *datatype* must be one of the following (the default data type is CLOB):

VARCHAR2 (N), where N is the size in bytes²

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.



- 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. For UTF-16 characters, *xml_encoding_spec* must be one of the following:

- encoding=UTF-16BE Big-endian UTF-16 encoding
- encoding=UTF-16LE Little-endian UTF-16 encoding

If you specify VERSION then the specified 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 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 8-17 uses XMLSerialize to produce a CLOB instance containing serialized XML data.

Example 8-17 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:

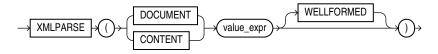
² The limit is 32767 or 4000 bytes, depending on the value of initialization parameter MAX_STRING_SIZE. See Oracle Database PL/SQL Packages and Types Reference.

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 8-9 shows the 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 8-18 uses XMLParse to parse a string of XML code and produce an XMLType instance.

Example 8-18 Using SQL/XML Function XMLPARSE

This results in the following output:

```
PO

124 <purchaseOrder poNo="12435">

<customerName>Acme Enterprises</customerName>

<itemNo>32987457</itemNo>

</purchaseOrder>
```

See Also:

Extensible Markup Language (XML) 1.0 for the definition of well-formed XML documents and fragments

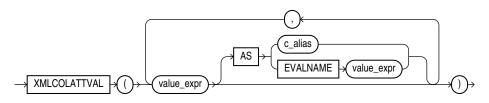


XMLCOLATTVAL Oracle SQL Function

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 8-10 shows the 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 8-19 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 8-19 XMLCOLATTVAL: Generating Elements with Attribute and Child Elements

This query produces the following XML result. (The result is shown here pretty-printed, for clarity.)



```
<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 8-8
```

XMLCDATA Oracle SQL Function

You use Oracle SQL function XMLCDATA to generate an XML CDATA section.

Figure 8-11 shows the syntax:



```
→ XMLCDATA → ()→ value_expr → ()→
```

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 8-20 uses XMLCDATA to generate an XML CDATA section.

Example 8-20 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>
```

Generation of XML Data 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.

Note:

The package DBMS XMLGEN is deprecated in Oracle Database 23ai.

This package is deprecated, and can be desupported in a future release. Oracle recommends that you use SQL/XML operators to generate XML from relational columns instead. Using ANSI SQL/XML operators for any generation and modification of XML documents provides a standardized and future-proof way to work with XML documents.

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.

- Using PL/SQL Package DBMS_XMLGEN
 You can use package DBMS_XMLGEN to generate XML data from relational data.
- Functions and Procedures of Package DBMS_XMLGEN
 PL/SQL package DBMS_XMLGEN provides functions and procedures for generating XML data from relational data.
- DBMS_XMLGEN Examples Examples here illustrate the use of PL/SQL package DBMS_XMLGEN.



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

You can use package DBMS XMLGEN to generate XML data from relational data.

Figure 8-12 illustrates how to use package DBMS XMLGEN. The steps are as follows:

- 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.



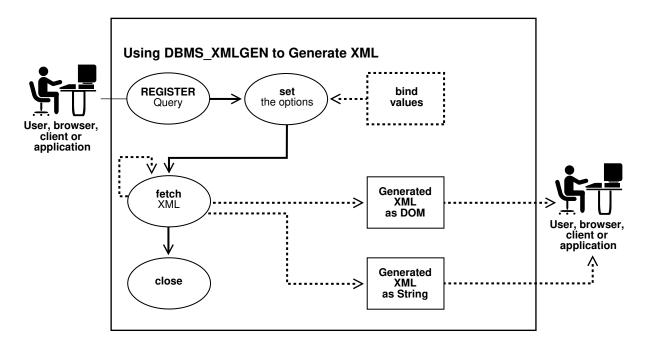


Figure 8-12 Using PL/SQL Package DBMS_XMLGEN

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

PL/SQL package DBMS_XMLGEN provides functions and procedures for generating XML data from relational data.

 Table 8-1 describes the functions and procedures of package DBMS XMLGEN.

Table 8-1	DBMS_XMLGEN Functions and Procedures
-----------	--------------------------------------

Function or Procedure	Description
	The context handle used by all functions.
SUBTYPE ctxHandle IS NUMBER	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.
	Returns a new context
<pre>newContext(queryString IN VARCHAR2)</pre>	<i>Parameter:</i> <pre>queryString (IN) - the query string, the result of which must be converted to XML</pre>
	<i>Returns:</i> 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.
<pre>newContext(queryString IN SYS_REFCURSOR) RETURN ctxHandle;</pre>	Creates a new context handle from a PL/SQL cursor variable. The context handle can be used for the rest of the functions.

Table 8-1	(Cont.) DBMS_XMLGEN Functions and Procedures
-----------	--

Function or Procedure	Description		
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 sam property as the result set generated by a CONNECT BY query. The resu 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.		
setRowTag(ctx IN ctxHandle,	Parameters:		
rowTag IN VARCHAR2);	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 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 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		
	Parameters:		
setRowSetTag(ctx(IN) - the context handle obtained from the newContext call.		
<pre>ctx IN ctxHandle, rowSetTag IN VARCHAR2);</pre>	rowSetTag(IN) - the name of the document root element to be used in the output. A NULL value for rowSetTag indicates that you do <i>not</i> 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.		

Function or Procedure	Description		
	Parameters:		
getXML(ctx(IN) - The context handle obtained from calling newContext.		
ctx IN ctxHandle, clobval IN OUT NCOPY clob,	clobval(IN/OUT) - the <code>CLOB</code> to which the XML document is to be		
dtdOrSchema IN number:= NONE);	appended,		
	dtd0rSchema (IN) - whether you should generate the DTD or Schema. This parameter is NOT supported.		
	Use this version of function getXML, to avoid any extra CLOB copies and if you want to reuse the same CLOB for subsequent calls. This getXML call is more efficient than the next flavor, though this involves that you create the LOB locator. When generating the XML, the number of rows indicated by the setSkipRows call are skipped, then the maximum number of rows as specified by the setMaxRows call (or the entire result if not specified) is fetched and converted to XML. Use the getNumRowsProcessed function to check if any rows were retrieved or not.		
getXML()	Generates the XML document and returns it as a CLOB.		
	Parameters:		
getXML(ctx(IN) - The context handle obtained from calling newContext.		
ctx IN ctxHandle, dtdOrSchema IN number:= NONE)	dtdOrSchema (IN) - whether to generate a DTD or XML schema. This parameter is <i>not</i> supported.		
RETURN clob;	<i>Returns:</i> A temporary CLOB containing the document. Free the temporary CLOB obtained from this function using the DBMS_LOB.freeTemporary call.		
	Parameters:		
getXMLType(ctx(IN) - The context handle obtained from calling newContext.		
ctx IN ctxHandle, dtdOrSchema IN number:= NONE)	dtdOrSchema(IN) - whether to generate a DTD or XML schema. This parameter is <i>not</i> supported.		
RETURN XMLType;	<i>Returns:</i> An XMLType instance containing the document.		
getXML (Converts the query results from the SQL query string sqlQuery to XML format.		
sqlQuery IN VARCHAR2, dtdOrSchema IN NUMBER := NONE) RETURN CLOB;	Returns: A CLOB instance.		
getXMLType(Converts the query results from the SQL query string <code>sqlQuery</code> to XML format.		
<pre>sqlQuery IN VARCHAR2, dtdOrSchema IN NUMBER := NONE) RETURN XMLType;</pre>	Returns: An XMLType instance.		
getNumRowsProcessed()	Gets the number of SQL rows processed when generating XML data using function getXML. This count does not include the number of rows <i>skipped</i> before generating XML data.		

Table 8-1 (Cont.) DBMS_XMLGEN Functions and Procedures



Function or Procedure	Description			
getNumRowsProcessed(<i>Parameter:</i> queryString(IN) - the query string, the result of which must be converted to XML			
ctx IN ctxHandle) RETURN number;	<i>Returns:</i> 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 <i>skipped</i> before generating XML data. Use this function to determine the terminating condition if you are calling getXML in a loop. getXML always generates 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.			
	Parameters:			
<pre>setMaxRows(ctx IN ctxHandle,</pre>	ctx(IN) - the context handle corresponding to the query executed,			
<pre>maxRows IN NUMBER);</pre>	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.			
	Parameters:			
setSkipRows(ctx IN ctxHandle,	$\mathtt{ctx}\left(\mathtt{IN}\right)$ - the context handle corresponding to the query executed,			
skipRows IN NUMBER);	${\tt skipRows}({\tt IN})$ - the number of rows to skip for each call to ${\tt getXML}.$			
	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.			
<pre>setConvertSpecialChars()</pre>	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.			

Table 8-1 (Cont.) DBMS_XMLGEN Functions and Procedures

Table 8-1	(Cont.) DBMS_XMLGEN Functions and Procedures
-----------	--

Function or Procedure	Description			
	Parameters:			
setConvertSpecialChars(ctx(IN) - the context handle to use,			
ctx IN ctxHandle,	conv (IN) - true indicates that conversion is needed.			
conv IN BOOLEAN);	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 is the type name itself. You can override that to use the name of the column with the _ITEM tag appended to it using this function.			
	Parameter: ctx(IN) - the context handle.			
<pre>useItemTagsForColl(ctx IN ctxHandle);</pre>	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.			
	Restarts the query and generate the XML from the first row again.			
restartQuery()				
<pre>restartQuery(ctx IN ctxHandle);</pre>	<i>Parameter:</i> 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.			
<pre>closeContext(ctx IN ctxHandle);</pre>	<i>Parameter:</i> 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	Encodes or decodes the XML data string argument.			
convert(Encoding refers to replacing entity references such as < to their escaped equivalent, such as <. 			
xmlData IN varchar2,	 Decoding refers to the reverse conversion. 			
<pre>flag IN NUMBER := ENTITY_ENCODE) RETURN VARCHAR2;</pre>				
	Encodes or decodes the passed in XML CLOB data.			
convert(• Encoding refers to replacing entity references such as < to their			
xmlData IN CLOB, flag IN NUMBER := ENTITY_ENCODE) RETURN CLOB;	 escaped equivalent, such as <. Decoding refers to the reverse conversion. 			

Function or Procedure	Description			
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". 			
<pre>useNullAttributeIndicator(ctx IN ctxHandle, attrind IN BOOLEAN := TRUE);</pre>	 EMPTY_TAG CONSTANT NUMBER := 2; This sets, for example, <foo></foo>. useNullAttributeIndicator is a shortcut for setNullHandling(ctx, NULL_ATTR). 			
<pre>setBindValue(ctx IN ctxHandle, bindVariableName IN VARCHAR2, bindValue IN VARCHAR2);</pre>	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.			
<pre>clearBindValue(ctx IN ctxHandle);</pre>	Clears all the bind values for all the bind variables appearing in the query string associated with the context handle. Afterward, all of the bind variables must rebind new values using setBindValue.			

Table 8-1 (Cont.) DBMS_XMLGEN Functions and Procedures

DBMS_XMLGEN Examples

Examples here illustrate the use of PL/SQL package DBMS XMLGEN.

Example 8-21 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.

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 8-22 uses DBMS XMLGEN to retrieve results from table HR.employees:

Example 8-23 uses DBMS XMLGEN with object types to represent nested structures.

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 to XML elements see 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 userdefined data-type instance is mapped to an XML document using a 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 8-24. To generate a hierarchical view of the data, that is, departments with their employees, Example 8-24 defines suitable object types to create the structure inside the database.

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 8-25 uses DBMS_XMLGEN.getXMLType to generate a purchase order document in XML format using object views.

Example 8-26 shows how to open a cursor variable for a query and use that cursor variable to create a new context handle for DBMS XMLGEN.

See Also:

Oracle Database PL/SQL Language Reference for more information about cursor variables (REF CURSOR)

Example 8-27 shows how to specify NULL handling when using DBMS XMLGEN.

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 instance. 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 8-28 uses DBMS XMLGEN.newContextFromHierarchy to generate a manageremployee hierarchy.

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 8-29 illustrates this.



Example 8-21 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.
```

Example 8-22 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 8-23 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 n
                                               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(gryCtx, 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.
```

Example 8-24 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 (0) 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,
                                      VARCHAR2(20),
                             dname
                             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(gryCtx) = 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.

```
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>
```

SELECT * FROM temp_clob_tab;

1 row selected.



Example 8-25 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
                                            NUMBER,
CREATE TYPE customer typ AS OBJECT(CustNo
                                   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
                    NUMBER,
  AS OBJECT (PONO
                         REF customer_typ,
            Cust ref
            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),
                                     CHAR(2),
                           State
                                     VARCHAR2(20),
                           Zip
                           Phonel
                                     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 */
                     (PONO
Custno
                                         /* purchase ofder hand
/* foreign KEY referencing customer */
                                NUMBER
                                CONSTRAINT po cust fk REFERENCES customer tab,
```

```
/* date of order */
                    OrderDate DATE,
                    ShipDate TIMESTAMP, /* date to be shipped */
                    ToStreet VARCHAR2(200), /* shipto address */
                              VARCHAR2(200),
                    ToCity
                    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),
                                     1.quantity, 1.discount)
                   FROM lineitems tab 1, stock tab s
                   WHERE 1.pono = p.pono AND s.stockno=1.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;
  gryCtx := 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">
   <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 8-26 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.

Example 8-27 DBMS_XMLGEN: Specifying NULL Handling

```
CREATE TABLE emp_tab (emp_id
                                  NUMBER PRIMARY KEY,
                                  VARCHAR2(20),
                     name
                                  NUMBER);
                      dept id
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.

Example 8-28 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(gryctx)));
 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;
Х
_____
<?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.
```

Example 8-29 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>
```

 $\ensuremath{\texttt{PL}}\xspace/\ensuremath{\texttt{SQL}}\xspace$ 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.

SYS_XMLAGG Oracle SQL Function

Oracle SQL function sys_XMLAgg aggregates all XML documents or fragments represented by an expression, producing a single XML document from them. It wraps the results of the expression in a new element named ROWSET (by default).

Oracle function sys_XMLAgg is similar to standard SQL/XML function XMLAgg, but sys_XMLAgg returns a single node and it accepts an XMLFormat parameter. You can use that parameter to format the resulting XML document in various ways.

```
Figure 8-13 SYS_XMLAGG Syntax
```





See Also:

- Oracle Database SQL Language Reference for information about sys XMLAgg
- Oracle Database SQL Language Reference for information about an XMLFormat parameter

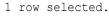
Ordering Query Results Before Aggregating, Using XMLAGG ORDER BY Clause

To use the XMLAgg ORDER BY clause before aggregation, specify the ORDER BY clause following the first XMLAGG argument.

This is illustrated in Example 8-30.

Example 8-30 Using XMLAGG ORDER BY Clause

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.)





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.

This is shown in Example 8-31.

Example 8-31 Returning a Rowset Using XMLTABLE

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.



Relational Views over XML Data

Relational database views over XML data provide conventional, relational access to XML content.

- Introduction to Creating and Using Relational Views over XML Data You can use the XML-specific functions and methods provided by Oracle XML DB to create conventional database views that provide relational access to XML content. This lets programmers, tools, and applications that understand Oracle Database, but not necessarily XML, work with XML content stored in the database.
- Creating a Relational View over XML: One Row for Each XML Document To expose each document in an XMLType table as a row in a relational view, use CREATE OR REPLACE VIEW AS SELECT, selecting from a join of the XMLType table and a relational table that you create from the XML data using SQL/XML function XMLTable.
- Creating a Relational View over XML: Mapping XML Nodes to Columns To expose data from multiple levels of an XMLType table as individual rows in a relational view, apply SQL/XML function XMLTable to each level. Use this technique whenever there is a one-to-*many* (1:N) relationship between documents in the XMLType table and rows in the view.
- Indexing Binary XML Data Exposed Using a Relational View
 If the relational columns of the structured component of an XMLIndex index over binary
 XML data match the columns of a relational view over that data, then the view too is
 effectively indexed.
- Querying XML Content As Relational Data Examples here show relational queries of XML data. They illustrate some of the benefits provided by creating relational views over XMLType tables and columns.

Introduction to Creating and Using Relational Views over XML Data

You can use the XML-specific functions and methods provided by Oracle XML DB to create conventional database views that provide relational access to XML content. This lets programmers, tools, and applications that understand Oracle Database, but not necessarily XML, work with XML content stored in the database.

The relational views can use XQuery expressions and SQL/XML functions such as XMLTable to define a mapping between columns in the view and nodes in an XML document.

Related Topics

• XQuery and Oracle XML DB

The XQuery language is one of the main ways that you interact with XML data in Oracle XML DB. Support for the language includes SQL*Plus commandXQUERY and SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast.



Indexes for 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. There are various ways that you can index XMLType data, whether it is XML schema-based or non-schema-based, and regardless of the XMLType storage model you use.

Creating a Relational View over XML: One Row for Each XML Document

To expose each document in an XMLType table as a row in a relational view, use CREATE OR REPLACE VIEW AS SELECT, selecting from a join of the XMLType table and a relational table that you create from the XML data using SQL/XML function XMLTable.

You use standard SQL/XML function XMLTable to map nodes in the XML document to columns in the view. Use this technique whenever there is a one-to-one (1:1) relationship between documents in the XMLType table and the rows in the view.

Example 9-1 creates relational view purchaseorder_master_view, which has one row for each row in XMLType table po binaryxml.

Example 9-1 Creating a Relational View of XML Content

```
CREATE TABLE po_binaryxml OF XMLType XMLTYPE STORE AS BINARY XML;
```

INSERT INTO po binaryxml SELECT OBJECT VALUE FROM OE.purchaseorder;

```
CREATE OR REPLACE VIEW purchaseorder_master_view AS

SELECT po.*

FROM po_binaryxml 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(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(20) PATH 'ShippingInstructions/telephone',

instructions VARCHAR2(2048) PATH 'SpecialInstructions') po;
```

View created.

DESCRIBE purchaseorder master view

Name	Null?	Туре		
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 INSTRUCTIONS VARCHAR2(24) VARCHAR2(2048)

Creating a Relational View over XML: Mapping XML Nodes to Columns

To expose data from multiple levels of an XMLType table as individual rows in a relational view, apply SQL/XML function XMLTable to each level. Use this technique whenever there is a one-to-many (1:N) relationship between documents in the XMLType table and rows in the view.

That is, you use the same general approach as for breaking up a single level (see Creating a Relational View over XML: One Row for Each XML Document): Define the columns making up the view, and map the XML nodes to those columns. But in this case you apply XMLTable to each document level that is to be broken up and stored in relational columns.

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, use XMLTable to project both the PurchaseOrder element and the LineItem collection.

When element PurchaseOrder is broken up, its descendant LineItem element is mapped to a column of type XMLType, which contains an XML fragment. That column is then passed to a second call to XMLTable to be broken into its various parts as multiple columns of relational values.

Example 9-2 illustrates this. It uses XMLTable to effect a one-to-*many* (1:N) relationship between the documents in XMLType table po_binaryxml and the rows in relational view purchaseorder_detail_view. The view provides access to the individual members of a collection and exposes the collection members as a set of rows.

In Example 9-2, there is one row in view purchaseorder_detail_view for each LineItem element in the XML documents stored in XMLType table po binaryxml.

The CREATE OR REPLACE VIEW statement of Example 9-2 defines the set of relational columns that make up the view. The SELECT statement passes table po_binaryxml as context to function XMLTable to create 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 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 child element Description.

Element Reference is projected in view purchaseorder_detail_view as column reference. It provides a foreign key that can be used to join rows in view purchaseorder_detail_view to corresponding rows in view purchaseorder_master_view. The correlated join in the CREATE OR REPLACE VIEW statement ensures that the one-to-many (1:N) relationship between element Reference and the associated LineItem elements is maintained whenever the view is accessed.



Example 9-2 Accessing Individual Members of a Collection Using a View

```
CREATE OR REPLACE VIEW purchaseorder detail view AS
  SELECT po.reference, li.*
    FROM po binaryxml 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
                                  NUMBER(38) PATH '@ItemNumber',
                      itemno
                       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') 1;
View created.
DESCRIBE purchaseorder_detail_view
Name Null? Type
-----
                VARCHAR2 (28)
REFERENCE
                          VARCHAR2(30)
ITEMNO
ITEMNONOMBER (38)DESCRIPTIONVARCHAR2 (256)PARTNOVARCHAR2 (14)QUANTITYNUMBER (12, 2)UNITPRICENUMBER (8, 4)
```

Indexing Binary XML Data Exposed Using a Relational View

If the relational columns of the structured component of an XMLIndex index over binary XML data match the columns of a relational view over that data, then the view too is effectively indexed.

When the XMLType data that is exposed in a relational view is stored as binary XML, you can typically improve performance by creating an XMLIndex index that has a structured component that matches the view columns. Such an index projects parts of the XML data onto relational columns, just as the view does. When the columns of the index match the columns of the view, the view is itself indexed.

To simplify the creation of such an XMLIndex index, you can PL/SQL function DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView to provide exactly the XMLTable expression needed for creating the index. That is the sole purpose of this function: to return an XMLTable expression that you can use to create an XMLIndex index for a relational view. It takes the view as argument and returns a CLOB instance. Example 9-3 illustrates this.

Example 9-4 shows the XMLTable expression used in Example 9-3.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about PL/SQL function DBMS XMLSTORAGE MANAGE.getSIDXDefFromView



Example 9-3 XMLIndex Index that Matches Relational View Columns

```
CALL DBMS_XMLINDEX.registerParameter(
    'my_param',
    DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView('PURCHASEORDER_MASTER_VIEW'));<sup>1</sup>
CREATE INDEX my_idx on po_binaryxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
    PARAMETERS ('PARAM my_param');
```

Example 9-4 XMLTable Expression Returned by PL/SQL Function getSIDXDefFromView

```
SELECT DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView('PURCHASEORDER_MASTER_VIEW')
FROM DUAL;
```

```
XMLTABLE po_binaryxml_XTAB_1 '/PurchaseOrder' PASSING OBJECT_VALUE
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'
```

Related Topics

 Use of XMLIndex with a Structured Component An XMLIndex structured component indexes specific islands of structure in your XML data.

Querying XML Content As Relational Data

Examples here show relational queries of XML data. They illustrate some of the benefits provided by creating relational views over XMLType tables and columns.

Example 9-5 and Example 9-6 show how to query master and detail relational views of XML data. Example 9-5 queries the master view to select the rows where column userid starts with S.

Example 9-6 joins the master view and the detail view. It selects 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.

Example 9-7 shows how to use relational views over XML content to perform businessintelligence queries on XML documents. The example query selects PurchaseOrder documents that contain orders for titles identified by UPC codes 715515009058 and 715515009126.

The query in Example 9-7 determines the number of copies of each film title that are ordered in each PurchaseOrder document. For example, 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.

¹ The view-name argument to getSIDXDefFromView must be uppercase, because that is how the name is recorded.

Example 9-5 Querying Master Relational View of XML Data

```
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.
```

Example 9-6 Querying Master and Detail Relational Views of XML Data

```
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 ITE		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.

Example 9-7 Business-Intelligence Query of XML Data Using a View

FROM purchas WHERE partno	<pre>count(*) "No of eorder_detail_vid IN (71551500912 lup(partno, quan)</pre>	ew 6, 715515("No of	Copies"
PARTNO	No of Orders No	of Copies	3		
			-		
715515009058	7	1	_		
715515009058	9		2		
715515009058	5	3	}		
715515009058	2	4	l		
715515009058	23				
715515009126	4	1	_		



715515009126	7	3
715515009126	11	
	34	
9 rows selected.		

10 XMLType Views

You can create XMLType views over relational and object-relational data.

• What Are XMLType Views?

XMLType views wrap existing relational and object-relational data in XML formats. This lets you use existing data in contexts that expect XML data and exploit XML features, including XML Schema.

- CREATE VIEW for XMLType Views: Syntax The syntax for the CREATE VIEW clause for creating XMLType views is presented.
- Creating Non-Schema-Based XMLType Views
 The XML data in a non XML Schema-based XMLType view is not constrained to conform to
 a registered XML schema. You can create a non-schema-based XMLType view using
 SQL/XML publishing functions.
- Creating XML Schema-Based XMLType Views The XML data in an XML Schema-based XMLType view is constrained to conform to an XML schema. You can create a schema-based XMLType view using SQL/XML publishing functions or using object types or views.
- 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.
- Referencing XMLType View Objects Using SQL Function REF You can reference an XMLType view object using SQL function ref.
- Using 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 DML. To determine whether an XMLType view is implicitly updatable, query it to see whether it is based on an object view or constructor that is itself inherently updatable.

What Are XMLType Views?

XMLType views wrap existing relational and object-relational data in XML formats. This lets you use existing data in contexts that expect XML data and exploit XML features, including XML Schema.

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 forms of storage for your data. You need not decide immediately whether to store it as XMLType or which XMLType storage model to use.

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.



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 nonschema-based XMLType views. XPath rewrite is described in XPath Rewrite for Object-Relational 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.

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 or 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 an XML-schema-based XMLType view requires that you annotate the XML schema with a mapping to existing object types or that you generate the XML schema from the existing object types.
- Directly from an XMLType table.

Related Topics

XML Schema Storage and Query: Basic

XML Schema is a standard for describing the content and structure of XML documents. You can register, update, and delete an XML schema used with Oracle XML DB. You can define storage structures to use for your XML schema-based data and map XML Schema data types to SQL data types.

- Relational Views over XML Data Relational database views over XML data provide conventional, relational access to XML content.
- Choice of XMLType Storage and Indexing Important design choices for your application include what XMLType storage model to use and which indexing approaches to use.

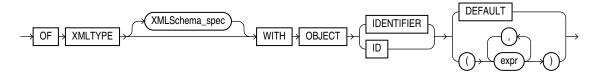
CREATE VIEW for XMLType Views: Syntax

The syntax for the CREATE VIEW clause for creating XMLType views is presented.

Figure 10-1 shows this syntax. See Oracle Database SQL Language Reference for details on the CREATE VIEW syntax.



Figure 10-1 Creating XMLType Views Clause: Syntax



Creating Non-Schema-Based XMLType Views

The XML data in a non XML Schema-based XMLType view is not constrained to conform to a registered XML schema. You can create a non-schema-based XMLType view using SQL/XML publishing functions.

Example 10-1 shows how to create an XMLType view using SQL/XML function XMLELement.

Existing data in relational tables or views can be exposed as XML data this way. If a view is generated using a SQL/XML publishing function then queries that access that view using XQuery expressions can often be rewritten. These optimized queries can then directly access the underlying relational columns. See XPath Rewrite for Object-Relational Storage for details.

You can perform a DML operation on an XMLType view, but, in general, you must write insteadof triggers to handle the DML operation.

See Also:

Generation of XML Data from Relational Data, for details on SQL/XML publishing functions

Example 10-1 Creating an XMLType View Using XMLELEMENT

<Emp empno="100"><name>steven king</name>chiredate>2003-00-17</niredate></fmp>
<Emp empno="101"><name>Neena Kochhar</name>chiredate>2005-09-21</hiredate></Emp>
<Emp empno="102"><name>Lex De Haan</name>chiredate>2001-01-13</hiredate></fmp>

Creating XML Schema-Based XMLType Views

The XML data in an XML Schema-based XMLType view is constrained to conform to an XML schema. You can create a schema-based XMLType view using SQL/XML publishing functions or using object types or views.



Create a schema-based view in either of these ways:

- Using SQL/XML publishing functions.
- 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.
- Creating XML Schema-Based XMLType Views Using SQL/XML Publishing Functions You can use SQL/XML publishing functions to create an XML Schema-based XMLType view.
- Creating XML Schema-Based XMLType Views Using Object Types or Object Views You can create an XML Schema-based XMLType view from object types or views by annotating the XML schema to define a mapping between XML types and SQL object types and object attributes.

Creating XML Schema-Based XMLType Views Using SQL/XML Publishing Functions

You can use SQL/XML publishing functions to create an XML Schema-based XMLType view.

- 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.
- Use SQL/XML publishing functions to create an XMLType view that conforms to the XML schema.

These two steps are illustrated in Example 10-2 and Example 10-3, respectively.

Example 10-4 illustrates querying an XMLType view.

Example 10-2 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 10-3 illustrates this.

Note:

Whenever you use SQL/XML function XMLAttributes with an XML schema reference to create an XMLType view, register the XML schema before creating the view, if possible. Otherwise, you must recompile the view after registering the XML schema, in order for the generated documents to be based on the XML schema.

In Example 10-3, 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 10-4 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 10-2 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"</pre>
                           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"</pre>
                                    type = "positiveInteger" minOccurs = "0"/>
                           <element name = "Name"</pre>
                                    type = "string" minOccurs = "0"/>
                           <element name = "Job"</pre>
                                    type = "string" minOccurs = "0"/>
                           <element name = "Manager"</pre>
                                    type = "positiveInteger" minOccurs = "0"/>
                           <element name = "HireDate"</pre>
                                    type = "date" minOccurs = "0"/>
                           <element name = "Salary"</pre>
                                    type = "positiveInteger" minOccurs = "0"/>
                           <element name = "Commission"</pre>
                                    type = "positiveInteger" minOccurs = "0"/>
                           <element name = "Dept">
                             <complexType>
                               <sequence>
                                 <element name = "DeptNo"</pre>
                                          type = "positiveInteger" minOccurs = "0"/>
                                 <element name = "DeptName"</pre>
                                          type = "string" minOccurs = "0"/>
                                 <element name = "Location"</pre>
                                           type = "positiveInteger" minOccurs = "0"/>
                               </sequence>
                             </complexType>
                           </element>
                        </sequence>
                      </complexType>
                    </element>
                  </schema>',
   LOCAL
            => TRUE,
```



```
GENTYPES => TRUE);
END;
```

```
Example 10-3 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;
```

Example 10-4 Querying an XMLType View

SELECT OBJECT_VALUE AS RESULT FROM emp_simple_xml WHERE ROWNUM < 2;

```
RESULT
```



```
</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.

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 10-5 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.

If the XML schema had no target namespace, then you could use attribute xsi:noNamespaceSchemaLocation to indicate that. Example 10-6 shows such an XML schema.

Example 10-7 creates a view that conforms to the XML schema in Example 10-6. The XMLAttributes clause creates an XML element that contains the noNamespace schema location attribute.

Example 10-8 creates view dept xml, which conforms to XML schema dept.xsd.

Example 10-5 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"),
                                                         AS "ipo:EmployeeId",
            XMLForest(e.employee id
                                                         AS "ipo:Name",
                      e.last 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",
                                                         AS "ipo:Commission",
                      e.commission pct
                 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/
               <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:Name>Fay</ipo:Name><ipo:Job>MK_REP</ipo:EmployeeId>202</ipo:EmployeeId> <ipo:Manager> <ipo:HireDate>2005-08-17</ipo:HireDate><ipo:Salary>6000</ipo:Salary> <ipo:Dept><ipo:DeptNo>20</ipo:DeptName><ipo:Location>1800</ipo:Location> </ipo:Dept> </ipo:DeptName>Marketing</ipo:DeptName><ipo:Location>1800</ipo:Location> </ipo:Dept> </ipo:Employee>

Example 10-6 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>',
            => TRUE,
    LOCAL
    GENTYPES => TRUE);
```

```
END;
```

Example 10-7 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 10-8 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="gualified">
                    <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>',
          => TRUE,
   LOCAL
   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 AS "DeptNo",
              d.department name AS "DeptName",
              d.location id AS "Location"),
    (SELECT XMLagg(
              XMLElement("Employee",
                         XMLForest(
                           e.employee id AS "EmployeeId",
                           e.last name AS "Name",
                           e.job id AS "Job",
                           e.manager id AS "Manager",
                           to char(e.hire date, 'YYYY-MM-DD') AS "Hiredate",
                           e.salary AS "Salary",
                           e.commission pct AS "Commission")))
       FROM employees e
       WHERE e.department_id = d.department_id))
    FROM departments d;
```

This is the XMLType instance that results:

```
<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

You can create an XML Schema-based XMLType view from object types or views by annotating the XML schema to define a mapping between XML types and SQL object types and object attributes.

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.

Annotate the XML schema *before* registering it. You typically do this when you wrap existing data to create an XMLType view.

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.
- 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 topics in this section 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.

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 Examples here create XMLType view emp_xml based on object views. Creating XMLType Department View, with Nested Employee Information
 XMLType view dept_xml is created so that each department in the view contains nested employee information.

Related Topics

XML Schema Storage and Query: Basic
 XML Schema is a standard for describing the content and structure of XML documents.
 You can register, update, and delete an XML schema used with Oracle XML DB. You can define storage structures to use for your XML schema-based data and map XML Schema data types to SQL data types.

Creating XMLType Employee View, with Nested Department Information

Examples here create XMLType view emp xml based on object views.

For the *last* step of the view creation, 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 for XMLType Employee View Create an object type for an XML Schema-based view.
- Step 2. Create and Register XML Schema emp_complex.xsd Create and register an XML schema, emp_complex.xsd. The schema maps XML elements and attributes to corresponding object-relational object attributes.
- Step 3a. Create XMLType View emp_xml Using Object Type emp_t Create an XMLType view using an object type.
- Step 3b. Create XMLType View emp_xml Using Object View emp_v Create an XMLType view using an object view.

Step 1. Create Object Types for XMLType Employee View

Create an object type for an XML Schema-based view.

Example 10-9 creates the object types used in the other steps.

Example 10-9 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),
              VARCHAR2(10),
      job
      mgr NUMBER(6),
      hiredate DATE,
      sal
               NUMBER(8, 2),
              NUMBER(2,2),
      comm
```



```
dept dept_t);
```

Step 2. Create and Register XML Schema emp_complex.xsd

Create and register an XML schema, emp_complex.xsd. The schema maps XML elements and attributes to corresponding object-relational object attributes.

Create 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), then register it. Example 10-10 registers it.

Example 10-10 creates and registers the XML schema using the target location http://www.oracle.com/emp_complex.xsd.

Example 10-10 Creating and 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"</pre>
                               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">
                     <rpre>xsd:element name="Employee" type="EMP_TType" xdb:SQLType="EMP_T"/>
                     <xsd:complexType name="EMP TType" xdb:SQLType="EMP T" xdb:maintainDOM="false">
                       <xsd:sequence>
                         <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="EMPNO"</pre>
                                      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"</pre>
                                      xdb:SQLType="NUMBER"/>
                         <xsd:element name="HIREDATE" type="xsd:date" xdb:SQLName="HIREDATE"</pre>
                                      xdb:SQLType="DATE"/>
                         <xsd:element name="SAL" type="xsd:double" xdb:SQLName="SAL"</pre>
                                      xdb:SQLType="NUMBER"/>
                         <xsd:element name="COMM" type="xsd:double" xdb:SQLName="COMM"</pre>
                                      xdb:SQLType="NUMBER"/>
                         <rpre><xsd:element name="DEPT" type="DEPT_TType" xdb:SQLName="DEPT"</pre>
                                      xdb:SQLType="DEPT T"/>
                       </xsd:sequence>
                     </xsd:complexType>
                     <xsd:complexType name="DEPT_TType" xdb:SQLType="DEPT_T"</pre>
```



```
xdb:maintainDOM="false">
                       <xsd:sequence>
                         <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"</pre>
                                     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"</pre>
                                    xdb:SQLType="NUMBER"/>
                      </xsd:sequence>
                    </xsd:complexType>
                  </xsd:schema>',
    LOCAL
           => TRUE,
    GENTYPES => FALSE);
END:
```

Step 3a. Create XMLType View emp_xml Using Object Type emp_t

Create an XMLType view using an object type.

Example 10-11 creates an XMLType view using object type emp t.

Example 10-11 uses SQL/XML function XMLCast in the OBJECT ID clause to convert the XML employee number to SQL data type BINARY DOUBLE.

See Also:

Step 3b. Create XMLType View emp_xml Using Object View emp_v for an alternative way to create view emp_xml, which uses object view emp_v

Example 10-11 Creating XMLType View emp_xml Using Object Type emp_t

Step 3b. Create XMLType View emp_xml Using Object View emp_v

Create an XMLType view using an object view.

Example 10-12 creates object view emp_v and then creates XMLType view emp_xml based on that object view.



Example 10-12 Creating an Object View and an XMLType View Based on the Object View

Creating XMLType Department View, with Nested Employee Information

XMLType view dept_xml is created so that each department in the view contains nested employee information.

For the *last* step of the view creation, 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 for XMLType Department View Create an object type for an XML Schema-based view.
- Step 2. Register XML Schema dept_complex.xsd Register XML schema dept complex.xsd.
- Step 3a. Create XMLType View dept_xml Using Object Type dept_t Create XMLType view dept xml using object type dept t.
- Step 3b. Create XMLType View dept_xml Using Relational Data Directly You can use SQL/XML publishing functions to create XMLType view dept_xml from the relational tables without using object type dept t.

Step 1. Create Object Types for XMLType Department View

Create an object type for an XML Schema-based view.

Example 10-13 creates the object types used in the other steps.

Example 10-13 Creating Object Types

CREATE T	YPE 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

Register XML schema dept_complex.xsd.

Example 10-14 illustrates this.

```
Example 10-14 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"</pre>
                              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"/>
                    <xsd:complexType name="DEPT TType" xdb:SQLType="DEPT T"</pre>
                                     xdb:maintainDOM="false">
                      <xsd:sequence>
                        <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"</pre>
                                     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"</pre>
                                     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:SQLCollSchema="HR"/>
                      </xsd:sequence>
                    </xsd:complexType>
                    <xsd:complexType name="EMP TType" xdb:SQLType="EMP T" xdb:maintainDOM="false">
                      <xsd:sequence>
                        <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="EMPNO"</pre>
                                     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"</pre>
                                      xdb:SQLType="NUMBER"/>
                         <xsd:element name="HIREDATE" type="xsd:date" xdb:SQLName="HIREDATE"</pre>
                                     xdb:SQLType="DATE"/>
                         <xsd:element name="SAL" type="xsd:double" xdb:SQLName="SAL"</pre>
                                      xdb:SQLType="NUMBER"/>
                         <xsd:element name="COMM" type="xsd:double" xdb:SQLName="COMM"</pre>
                                     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

Create XMLType view dept_xml using object type dept_t.

Example 10-15 illustrates this.

```
Example 10-15 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

You can use SQL/XML publishing functions to create XMLType view dept_xml from the relational tables without using object type dept t.

Example 10-16 illustrates this.



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.

```
Example 10-16 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;

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 10-17 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 10-18 shows how you can create an XMLType view by transforming XML data using an XSL stylesheet.

Example 10-17 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';
```

Here, dallas_dept_view restricts the XMLType table rows to those departments whose location is Dallas.

Example 10-18 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.coll)
FROM purchaseorder p, xsl tab x;
```

Related Topics

 SQL Function XMLTRANSFORM and XMLType Method TRANSFORM() SQL function XMLtransform transforms an XML document by using an XSLT stylesheet. It returns the processed output as XML, HTML, and so on, as specified by the stylesheet.

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.

Using 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 DML. To determine whether an XMLType view is implicitly updatable, query it to see whether it is based on an object view or constructor that is itself inherently updatable.

Example 10-19 illustrates this.

Example 10-19 Determining Whether an XMLType View Is Implicitly Updatable, and Updating It

```
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_t.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"</pre>
                              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"/>
                    <xsd:complexType name="DEPT TType" xdb:SQLType="DEPT T"</pre>
                                     xdb:maintainDOM="false">
                       <xsd:sequence>
                         <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"</pre>
                                      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"</pre>
                                      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</pre>
         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 =
    XMLQuery('copy $i := $p1 modify
                (for $j in $i/Department/DNAME
                 return replace value of node $j with $p2)
              return $i'
             PASSING d.OBJECT VALUE AS "p1", 'Shipping' AS "p2" RETURNING CONTENT)
    WHERE XMLExists('/Department[DEPTNO=300]' PASSING OBJECT VALUE);
```

Part IV XMLType APIs

•

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.

- PL/SQL APIs for XMLType There are several PL/SQL packages that provide APIs for XMLType.
 - PL/SQL Package DBMS_XMLSTORE You can use the PL/SQL package DBMS_XMLSTORE to insert, update, or delete data from XML documents stored object-relationally. It uses a canonical XML mapping similar to the one produced by package DBMS_XMLGEN. It converts the mapping to object-relational constructs and then inserts, updates or deletes the corresponding values in relational tables.
- Java DOM API for XMLType

The Java DOM API for XMLType lets you operate on XMLType instances using a DOM. You can use it to manipulate XML data in Java, including fetching it through Java Database Connectivity (JDBC).

- C DOM API for XMLType The C DOM API for XMLType lets you operate on XMLType instances using a DOM in C.
- Oracle XML DB and Oracle Data Provider for .NET
 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.



11 PL/SQL APIs for XMLType

There are several PL/SQL packages that provide APIs for XMLType.

- Overview of PL/SQL APIs for XMLType The PL/SQL Application Program Interfaces (APIs) for XMLType include a DOM API, a parser API, and a processor API.
- PL/SQL DOM API for XMLType (DBMS_XMLDOM) The PL/SQL DOM API for XMLType, DBMS_XMLDOM lets you operate on XMLType instances using a DOM.
- PL/SQL Parser API for XMLType (DBMS_XMLPARSER) The PL/SQL Parser API for XMLType (DBMS_XMLPARSER) builds a parsing result tree that can be accessed by PL/SQL APIs. If parsing fails, it raises an error.
- PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) You can use PL/SQL package DBMS_XSLPROCESSOR to transform one XML document to another or to convert XML data into HTML, PDF, or other formats. This package traverses the DOM tree for the document and applies Extensible Stylesheet Language Transformation (XSLT) statements to it to produce the transformed document.

Overview of PL/SQL APIs for XMLType

The PL/SQL Application Program Interfaces (APIs) for XMLType include a DOM API, a parser API, and a processor API.

 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-schemabased documents.

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 creating a DOM and 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.
- PL/SQL APIs for XMLType: Features
 You can use the PL/SQL APIs for XMLType to create XMLType tables, columns, and views;
 construct XMLType instances from data encoded in different character sets; and access and manipulate XMLType in various ways.
- PL/SQL APIs for XMLType: References The PL/SQL Application Programming Interfaces (APIs) for XMLType are described.



PL/SQL APIs for XMLType: Features

You can use the PL/SQL APIs for XMLType to create XMLType tables, columns, and views; construct XMLType instances from data encoded in different character sets; and access and manipulate XMLType in various ways.

Lazy Load of XML Data (Lazy Manifestation)
 Lazy XML loading loads rows of data only when they are requested, enhancing scalability of your applications that involve large XML documents and many concurrent users.

- XMLType Data Type Supports XML Schema SQL data type XMLType supports XML Schema.
- XMLType Supports Data in Different Character Sets
 You can use PL/SQL to create XMLType instances from data that is encoded in any Oracle supported character set. To do this, you use the PL/SQL XMLType constructor or XMLType
 method createXML().

Related Topics

Oracle XML DB Features

Oracle XML DB 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.

• Query and Update of XML Data

There are many ways for applications to query and update XML data that is in Oracle Database, both XML schema-based and non-schema-based.

Lazy Load of XML Data (Lazy Manifestation)

Lazy XML loading loads rows of data only when they are requested, enhancing scalability of your applications that involve large XML documents and many concurrent users.

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

SQL data type XMLType supports XML Schema.

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

You can use PL/SQL to create <code>XMLType</code> instances from data that is encoded in any Oracle-supported character set. To do this, you use the PL/SQL <code>XMLType</code> constructor or <code>XMLType</code> method <code>createXML()</code>.



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 APIs for XMLType: References

The PL/SQL Application Programming Interfaces (APIs) for XMLType are described.

Table 11-1 lists the reference documentation for the PL/SQL APIs that you can use to manipulate XML data. The main reference for PL/SQL 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
- Oracle Database XML C API Reference for information about C APIs for XML
- Oracle Database XML C++ API Referencefor information about C++ APIs for XML

API	Documentation	Description
XMLType	Oracle Database PL/SQL Packages and Types Reference, chapter "XMLType"	PL/SQL APIs with XML operations on XMLType data – validation, transformation.
Database URI types	<i>Oracle Database PL/SQL Packages and Types Reference</i> , chapter "Database URI TYPEs"	Functions used for various URI types.



API	Documentation	Description
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 recreate 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_ADMIN	Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XDB_ADMIN"	PL/SQL API for the management of Oracle XML DB Repository by database administrators.
DBMS_XDB_CONFIG	Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XDB_CONFIG"	PL/SQL API for managing Oracle XML DB configuration sessions.
DBMS_XDB_CONSTANTS	Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XDB_CONSTANTS"	PL/SQL constants for use with Oracle XML DB
DBMS_XDB_REPOS	Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XDB_REPOS"	PL/SQL API for the use of Oracle XML DB Repository by application developers.
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.
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.

Table 11-1 (Cont.) PL/SQL APIs Related to XML



API	Documentation	Description
DBMS_XMLSCHEMA_ANNOTATE	Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XMLSCHEMA_ANNOTATE"	PL/SQL API for adding and managing Oracle-specific XML Schema annotations.
DBMS_XMLSTORAGE_MANAGE	Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XMLSTORAGE_MANAGE"	PL/.SQL API managing and modifying storage of XML data after XML schema registration.
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.

Table 11-1 (Cont.) PL/SQL APIs Related to XML

PL/SQL DOM API for XMLType (DBMS_XMLDOM)

The PL/SQL DOM API for XMLType, DBMS_XMLDOM lets you operate on XMLType instances using a DOM.

- Overview of the W3C Document Object Model (DOM) Recommendation The Document Object Model (DOM) recommended by the World Wide Web Consortium (W3C) is a universal API for accessing the structure of XML documents.
- 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.
- Application Design Using Oracle XML Developer's Kit and Oracle XML DB When you build applications based on Oracle XML DB, you do not need the additional components provided by Oracle XML Developer's Kit (XDK). However, you can use XDK components with Oracle XML DB to deploy a full suite of XML-enabled applications that run end-to-end.
- Preparing XML Data to Use the PL/SQL DOM API for XMLType Create an XML schema, annotate it to map XML to SQL objects, and register the XML schema.
- XML Schema Types Are Mapped to SQL Object Types
 An XML schema must be registered with Oracle XML DB 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.
- Wrap Existing Data as XML with XMLType Views
 To make existing relational and object-relational data available to your XML applications,
 you can create XMLType views based on it. You can then access the resulting XML data
 using the PL/SQL DOM API.
- DBMS_XMLDOM Methods Supported by Oracle XML DB
 All DBMS_XMLDOM methods are supported by Oracle XML DB, with a few exceptions.
- PL/SQL DOM API for XMLType: Node Types
 The DOM specifies the way elements within an XML document are used to create an
 object-based tree structure. It defines and exposes interfaces to manage and use the



objects stored in XML documents. The DOM supports storage of documents in diverse systems.

- PL/SQL Function NEWDOMDOCUMENT and DOMDOCUMENT Nodes
 PL/SQL function newDOMDocument constructs a DOM document handle, given an XMLType value. The resulting handle is of type DOMDocument.
- DOM NodeList and NamedNodeMap Objects When you change the document structure underlying a DOMDocument instance, the changes are reflected in all relevant NodeList and NamedNodeMap objects.
- Overview of Using the PL/SQL DOM API for XMLType (DBMS_XMLDOM) Using PL/SQL package DBMS_XMLDOM typically involves creating DOM documents, traversing or extending the DOM tree, and creating and manipulating nodes.
- PL/SQL DOM API for XMLType Examples Examples are presented of using the PL/SQL DOM API for XMLType.
- Large Node Handling Using DBMS_XMLDOM
 Oracle XML DB provides abstract streams and stream-manipulation methods that you can use to handle XML nodes that are larger than 64 K bytes.
- Get-Push Model for Large Node Handling In this model, you retrieve the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.
- Get-Pull Model for Large Node Handling In this model, you retrieve the value of a DOM node, using a parser that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.
- Set-Pull Model for Large Node Handling In this model, you set the value of a DOM node, using a parser that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.
- Set-Push Model for Large Node Handling In this model, you set the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.
- Determining Binary Stream or Character Stream for Large Node Handling You can use subprogram DBMS_XMLDOM.useBinaryStream to determine whether to use a character stream or a binary stream to access the content of a large node.

🖍 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

The Document Object Model (DOM) recommended by the World Wide Web Consortium (W3C) is a universal API for accessing the structure of XML documents.

The DOM 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 XML Developer's Kit Extensions to the W3C DOM Standard

Oracle XML Developer's Kit (XDK) extends the W3C DOM API. These extensions are supported by Oracle XML DB except for those relating to client-side operations that are not applicable in the database. See the Simple API for XML (SAX) interface in the Oracle XML Developer's Kit Java and C components.

- Supported W3C DOM Recommendations
 All Oracle XML DB APIs for accessing and manipulating XML data comply with standard XML processing requirements as approved by the W3C. The PL/SQL DOM supports Levels 1 and 2 of the W3C DOM specifications.
- Difference Between DOM and SAX DOM is the primary generic tree-based API for XML. SAX (Simple API for XML) is the primary generic event-based programming interface between an XML parser and an XML application.

Oracle XML Developer's Kit Extensions to the W3C DOM Standard

Oracle XML Developer's Kit (XDK) extends the W3C DOM API. These extensions are supported by Oracle XML DB except for those relating to client-side operations that are not applicable in the database. See the Simple API for XML (SAX) interface in the Oracle XML Developer's Kit 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 data comply with standard XML processing requirements as approved by the W3C. The PL/SQL DOM supports Levels 1 and 2 of the W3C DOM specifications.

- **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.

Oracle support for DOM is as follows:

- In Oracle9i release 1 (9.0.1), Oracle XML Developer's Kit for PL/SQL implemented DOM Level 1.0 and parts of DOM Level 2.0.
- In Oracle9i release 2 (9.2) and Oracle Database 10g 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.



Difference Between DOM and SAX

DOM is the primary generic tree-based API for XML. 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. The child objects have their own 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:

Examples of structural manipulations are:

- Reordering elements
- Adding or deleting elements
- Adding or deleting attributes
- Renaming elements

See Also:

- Document Object Model (DOM)
- SAX Project

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) does 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 object-relational 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 , for a description of W3C DOM Recommendations.



PL/SQL DOM API Support for XML Schema

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.

Enhanced DOM 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.

PL/SQL DOM API Support for XML Schema

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 DOM 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.

Related Topics

Java DOM API for XMLType

The Java DOM API for XMLType lets you operate on XMLType instances using a DOM. You can use it to manipulate XML data in Java, including fetching it through Java Database Connectivity (JDBC).

Application Design Using Oracle XML Developer's Kit and Oracle XML DB

When you build applications based on Oracle XML DB, you do not need the additional components provided by Oracle XML Developer's Kit (XDK). However, you can use XDK components with Oracle XML DB to deploy a full suite of XML-enabled applications that run end-to-end.



These XDK features are particularly useful for developing XML applications based on Oracle XML DB.

- 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 XML Developer's Kit contains 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 XML Developer's Kit is available for Java, C, and C++. Oracle XML Developer's Kit is fully supported and comes with a commercial redistribution license.

Oracle XML Developer's Kit 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.
- 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 XSL stylesheets.
- 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

Create an XML schema, annotate it to map XML to SQL objects, and register the XML schema.

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 Wrap Existing Data as XML with XMLType Views.
- 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.

XML Schema Types Are Mapped to SQL Object Types

An XML schema must be registered with Oracle XML DB 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.

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 database. 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.

DOM Fidelity for XML Schema Mapping

Elements and attributes declared within an 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.

Related Topics

XML Schema Storage and Query: Basic

XML Schema is a standard for describing the content and structure of XML documents. You can register, update, and delete an XML schema used with Oracle XML DB. You can define storage structures to use for your XML schema-based data and map XML Schema data types to SQL data types.

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 an 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, attribute SYS_XDBPD\$ has been omitted from most examples, for simplicity. However, the attribute is always present in SQL object types generated by schema registration.

Related Topics

SYS_XDBPD\$ and DOM Fidelity for Object-Relational Storage

In order to provide DOM fidelity for XML data that is stored object-relationally, Oracle XML DB records all information that cannot be stored in any of the other object attributes as instance-level metadata using the system-defined binary object attribute SYS_XDBPD\$ (positional descriptor, or PD).

Wrap Existing Data as XML with XMLType Views

To make existing relational and object-relational data available to your XML applications, you can create XMLType views based on it. You can then access the resulting 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.

Related Topics

• XMLType Views

You can create XMLType views over relational and object-relational data.

DBMS_XMLDOM Methods Supported by Oracle XML DB

All DBMS XMLDOM methods are supported by Oracle XML DB, with a few exceptions.

These methods are not supported by Oracle XML DB:

- 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

The DOM specifies the way elements within an XML document are used to create an objectbased tree structure. It defines and exposes interfaces to manage and use the objects stored in XML documents. The DOM supports storage of documents in diverse systems.

In the DOM specification, the term "**document**" describes a container for many different kinds of information or data, which the DOM objectifies.

When a request such as getNodeType (myNode) is invoked, it returns myNodeType, which is the node type supported by the parent node. The following 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 11-2 shows the node types for XML and HTML, and the allowed corresponding child node types.

Table 11-2 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



Node Type	Children Node Types
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
CDATASection	No children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	No children

Table 11-2 (Cont.) XML and HTML DOM Node Types and Their Child Node Types

Oracle XML DB DOM API for XMLType also specifies the following 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.

PL/SQL Function NEWDOMDOCUMENT and DOMDOCUMENT Nodes

PL/SQL function newDOMDocument constructs a DOM document handle, given an XMLType value. The resulting handle is of type DOMDocument.

A typical usage scenario for a PL/SQL application is:

- 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

When you change the document structure underlying a DOMDocument instance, the changes 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 existing children) then those changes automatically propagate to 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.

Overview of Using the PL/SQL DOM API for XMLType (DBMS_XMLDOM)

Using PL/SQL package DBMS_XMLDOM typically involves creating DOM documents, traversing or extending the DOM tree, and creating and manipulating nodes.

Figure 11-1 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. Procedure newDOMDocument processes the XMLType instance or empty document.

This creates a DOMDocument instance.

- You can use DOM API PL/SQL methods such as createElement(), createText(), createAttribute(), and createComment() to traverse and extend the DOM tree.
- 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.



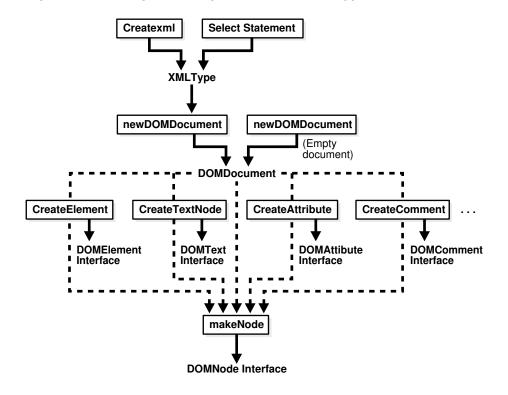


Figure 11-1 Using the PL/SQL DOM API for XMLType

PL/SQL DOM API for XMLType – Examples

Examples are presented 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 11-1 creates a hierarchical, representation of an XML document in dynamic memory: a DOM document.

Example 11-1 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 11-2 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 11-1 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);
          := DBMS XMLDOM.makeNode(doc);
  ndoc
  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>
   </PERSON>
```



This query confirms that the data has changed:

```
SELECT * FROM person;
SYS_NC_ROWINFO$
------
<PERSON>
<NAME>raj</NAME>
</PERSON>
```

```
1 row selected.
```

Example 11-2 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.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

Oracle XML DB provides abstract streams and stream-manipulation methods that you can use to handle XML nodes that are larger than 64 K bytes.

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.

 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.

- **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.
- **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.
- **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.

Your application acts as the client, with the parser as its service provider. The parser mode determines whether the parser or your application drives the stream dataflow.

- For a parser in push mode, your application pushes data to the parser in an output stream, and the parser returns the result of the requested operation.
- For a parser in pull mode, your application pulls data from the parser in an input stream. Each data item in the stream is the result of a parsing event.

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. The name of each subprogram reflects whether it reads ("get") or writes ("set") data, and whether the parser is being used in push ("push") or pull ("pull") mode.

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.

Related Topics

- Get-Push Model for Large Node Handling In this model, you retrieve the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.
- Get-Pull Model for Large Node Handling In this model, you retrieve the value of a DOM node, using a parser that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.
- Set-Pull Model for Large Node Handling
 In this model, you set the value of a DOM node, using a parser that is in pull mode. Oracle
 XML DB reads the event data from an input stream written by the parser.
- Set-Push Model for Large Node Handling In this model, you set the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.

Large XML Node Handling with Java

Oracle XML DB provides abstract streams and stream-manipulation methods that you can use to handle XML nodes that are larger than 64 K bytes. Use Java classes XMLNode and XMLAttr, together with a thick or kprb connection, to manipulate large nodes.

See Also:

- 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 for Large Node Handling

In this model, you retrieve the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.

To read a node value in this model, your 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 adding node data to 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 can be any that is supported by Oracle XML DB. 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.

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 11-3 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 11-4 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 for Large Node Handling

In this model, you retrieve the value of a DOM node, using a parser that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.

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.

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 for Large Node Handling. The node data type must be RAW or BLOB or else an exception is raised.

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 11-5 illustrates reading a node value as binary data in 50-byte increments:

Example 11-6 illustrates reading a node value as character data in 50-character increments:

Example 11-5 Get-Pull of Binary Data

```
DECLARE
 istream SYS.utl_BinaryInputStream;
            DBMS XMLDOM.domnode;
 node
 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 11-6 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 TF:
-- Process next 50 characters of node value in buffer
END LOOP
. . .
END;
```

Set-Pull Model for Large Node Handling

In this model, you set the value of a DOM node, using a parser that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.

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.

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.

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 11-7 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:

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;
```

Example 11-7 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);
```

Set-Push Model for Large Node Handling

In this model, you set the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.

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.



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.

This function returns an instance of the PL/SQL <code>SYS.utl_CharacterOutputStream</code> 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 <code>CLOB</code>, the character data written to the stream is converted to the node data type. If the data type of the node is character or <code>CLOB</code>, then the character data written to the stream is converted from PL/SQL session character set to the character set of the node

Example 11-8 illustrates setting the value of a node to binary data by writing 50-byte segments into the SYS.utl BinaryOutputStream:

Example 11-8 Set-Push of Binary Data

```
DECLARE
 ostream SYS.utl_BinaryOutputStream;
node DBMS_XMLDOM.domnode;
buffer raw(500);
segment raw(50);
  numBytes pls integer;
              pls_integer;
  offset
  . . .
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 for Large Node Handling

You can use subprogram DBMS_XMLDOM.useBinaryStream to determine whether to use a character stream or a binary stream to access the content of a large node.

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)

The PL/SQL Parser API for XMLType (DBMS_XMLPARSER) builds a parsing result tree that can be accessed by PL/SQL APIs. If parsing fails, it raises an error.

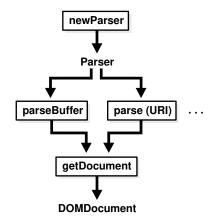
A software module called an XML parser or processor reads XML documents and provides access to their content and structure. An XML parser usually does its work on behalf of another module, typically the application.

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 constitute character data and some of which act as 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.

Figure 11-2 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 11-2 Using the PL/SQL Parser API for XMLType



Example 11-9 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.



```
    Note:
Method DBMS_XMLPARSER.setErrorLog() is not supported.
    See Also:
Oracle Database PL/SQL Packages and Types Reference for descriptions of
individual DBMS_XMLPARSER methods
```

Example 11-9 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)

You can use PL/SQL package DBMS_XSLPROCESSOR to transform one XML document to another or to convert XML data into HTML, PDF, or other formats. This package traverses the DOM tree for the document and applies Extensible Stylesheet Language Transformation (XSLT) statements to it to produce the transformed document.

The W3C XSL Recommendation describes rules for transforming a source tree into a result tree. A transformation expressed in XSL is called an **XSLT stylesheet**. The transformation specified is achieved by associating patterns with templates defined in the XSLT stylesheet. A template is instantiated to create part of the result tree. XSLT is widely used to convert XML data to HTML for web browser display.



Note:

Oracle XML DB applications do not require a separate XML parser. However, applications requiring external processing can still use the XML Parser for PL/SQL first to expose the document structure. The XML Parser for PL/SQL in Oracle XML Developer's Kit 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 of running applications on the server include performance improvements and enhanced access and manipulation options.

PL/SQL XSLT Processor for XMLType: Features

PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) follows the W3C XSLT final recommendation (REC-xslt-19991116). It provides a convenient and efficient way of applying a single XSL stylesheet to multiple documents.

 Using the PL/SQL XSLT Processor API for XMLType (DBMS_XSLPROCESSOR) You create an XSLT processor, build a STYLESHEET object from a DOM document, transform the document using the processor and the stylesheet, and use the DOM API for XMLType to manipulate the result of XSLT processing.

Related Topics

• Transformation and Validation of XMLType Data There are several Oracle SQL functions and XMLType APIs for transforming XMLType data using XSLT stylesheets and for validating XMLType instances against an XML schema.

PL/SQL XSLT Processor for XMLType: Features

PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) follows the W3C XSLT final recommendation (REC-xslt-19991116). It provides a convenient and efficient way of applying a single XSL stylesheet to multiple documents.

The methods in PL/SQL package DBMS_XSLPROCESSOR 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)

You create an XSLT processor, build a STYLESHEET object from a DOM document, transform the document using the processor and the stylesheet, and use the DOM API for XMLType to manipulate the result of XSLT processing.

Figure 11-3 illustrates how to use the XSLT Processor for XMLType (DBMS XSLPROCESSOR).

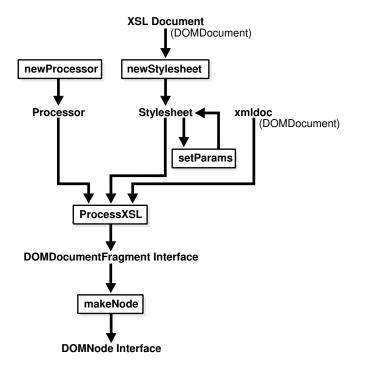


Figure 11-3 Using the PL/SQL XSLT Processor for XMLType

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.
- 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.

Example 11-10 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 stylesheet. An XSL processor instance is created, which applies the stylesheet 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.

```
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"</pre>
                               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);
            := DBMS XSLPROCESSOR.newStyleSheet(xsltdomdoc, '');
  xsl
  proc
            := DBMS XSLPROCESSOR.newProcessor;
  --apply stylesheet to DOM document
  outdomdocf := DBMS XSLPROCESSOR.processXSL(proc, xsl, indomdoc);
            := DBMS XMLDOM.makeNode(outdomdocf);
  outnode
  -- 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);
```

Example 11-10 Transforming an XML Document Using an XSL Stylesheet



END; /

This produces the following output:

```
<emp>
<empno>1</empno>
<fname>robert</fname>
<job>engineer</job>
<lname>smith</lname>
<sal>1000</sal>
</emp>
```



12 PL/SQL Package DBMS_XMLSTORE

You can use the PL/SQL package DBMS_XMLSTORE to insert, update, or delete data from XML documents stored object-relationally. It uses a canonical XML mapping similar to the one produced by package DBMS_XMLGEN. It converts the mapping to object-relational constructs and then inserts, updates or deletes the corresponding values in relational tables.

Note:

The PL/SQL package DBMS XMLSTORE is deprecated in Oracle Database 23ai.

This package is deprecated, and can be desupported in a future release. Oracle recommends that you use regular SQL DML and with standard XQuery and SQL/XML to store and manage XML data. Using standard functionality provides future-proof way to store and manipulate XML data.

Using Package DBMS_XMLSTORE

Create a context handle, perform one or more insertions, updates, or deletions, and then close the context,

Inserting an XML Document Using DBMS_XMLSTORE

You can use PL/SQL package DBMS_XMLSTORE to insert an XML document into a table or view. It parses the document and creates an INSERT statement into which it binds the values. By default, it inserts values into all of the columns represented by elements in the XML document.

Updating XML Data Using DBMS_XMLSTORE

You can use PL/SQL package DBMS_XMLSTORE to update (modify) existing data. You specify which rows to update by calling procedure setKeyColumn once for each of the columns that are used collectively to identify the row. (In SQL, you would specify the rows using a WHERE clause in an UPDATE statement.)

Deleting XML Data Using DBMS_XMLSTORE

You can use PL/SQL package DBMS_XMLSTORE to delete existing data. You specify which rows to delete by calling procedure setKeyColumn once for each of the columns that are used collectively to identify the row. (In SQL, you would specify the rows using a WHERE clause in an UPDATE statement.)

Using Package DBMS_XMLSTORE

Create a context handle, perform one or more insertions, updates, or deletions, and then close the context,

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 quotation mark (") the string that is passed to the function.



By default, XML documents are expected to use the <ROW> tag to identify rows. This is the same default used by package DBMS_XMLGEN when generating XML data. You can use function setRowTag to override this behavior.

- 2. (Optional) Perform one or more insertions, updates or deletions (you can *repeat* this step):
 - For insertions, to improve performance you can specify the list of columns to insert by calling procedure DBMS_XMLSTORE.setUpdateColumn for each column. The default behavior (if you do not specify the list of columns) is to insert values for each column whose corresponding element is present in the XML document.
 - For updates, use function DBMS_XMLSTORE.setKeyColumn to specify one or more (pseudo-) key columns, which are used to specify the *rows* to update. You do this in the WHERE clause of a SQL UPDATE statement. The columns that you specify need not be keys of the table, but together they must uniquely specify the rows to update.

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.

To improve performance, you can also specify the list of update columns using DBMS_XMLSTORE.setUpdateColumn. 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.

- For *deletions* you specify (pseudo-) key columns to identify the row(s) to delete. You do this the same way you specify rows to update (see previous).
- 3. Provide a document to DBMS_XMLSTORE function insertXML, updateXML, or deleteXML. You can repeat this step to update several XML documents.
- 4. Close the context by calling function DBMS XMLSTORE.closeContext.

Inserting an XML Document Using DBMS_XMLSTORE

You can use PL/SQL package DBMS_XMLSTORE to insert an XML document into a table or view. It parses the document and creates an INSERT statement into which it binds the values. By default, it inserts values into all of the columns represented by elements in the XML document.

Example 12-1 uses DBM_XMLSTORE to insert the information for two new employees into the employees table. The information to insert is provided as XML data.

Example 12-1 Inserting Data with Specified Columns

<pre>SELECT employee_id AS EMP_ID, salary, hire_date, job_id, email, last_name FROM employees WHERE department_id = 30;</pre>						
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 XML Data Using DBMS_XMLSTORE

You can use PL/SQL package DBMS_XMLSTORE to update (modify) existing data. You specify which rows to update by calling procedure setKeyColumn once for each of the columns that are used collectively to identify the row. (In SQL, you would specify the rows using a WHERE clause in an UPDATE statement.)

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 12-2 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.

The following UPDATE statement is equivalent to the use of DBM XMLSTORE in Example 12-2:

```
UPDATE hr.employees SET first_name = 'Pat' WHERE employee_id = 188;
```

Example 12-2 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
```



Deleting XML Data Using DBMS_XMLSTORE

You can use PL/SQL package DBMS_XMLSTORE to delete existing data. You specify which rows to delete by calling procedure setKeyColumn once for each of the columns that are used collectively to identify the row. (In SQL, you would specify the rows using a WHERE clause in an UPDATE statement.)

Example 12-3 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.

13 Java DOM API for XMLType

The Java DOM API for XMLType lets you operate on XMLType instances using a DOM. You can use it to manipulate XML data in Java, including fetching it through Java Database Connectivity (JDBC).

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.

- Access to XMLType Data Using JDBC Java Database Connectivity (JDBC) is a SQL-based way for Java applications to access any data in Oracle Database, including XML documents in Oracle XML DB.
- Manipulating XML Database Documents Using JDBC You can update, insert, and delete XMLType data stored in the database using Java Database Connectivity (JDBC) with Oracle XML DB.
- Loading a Large XML Document into the Database Using JDBC To load a large XML document into the database using Java Database Connectivity (JDBC), use a Java CLOB object to hold the document, and use Java method insertXML() to perform the insertion.
- MS Windows Java Security Manager Permissions for Java DOM API with a Thick Connection

If you use Java Security Manager (class SecurityManager) on MS Windows to implement a security policy for your application, then you must add certain permissions to your security policy file, in order to use the Java DOM API for XMLType with a thick connection.

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. 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.

• XMLType Instance Representation in Java (JDBC or SQLJ)

An XMLType instance is represented in Java by oracle.xdb.XMLType. When an instance of XMLType is fetched using JDBC or a SQLJ client, it is automatically manifested as an object of the provided XMLType class.

Classes of Java DOM API for XMLType

Oracle XML DB supports the W3C DOM Level 2 Recommendation. It also provides Oracle-specific extensions, to facilitate interfacing your application with Oracle XML Developer's Kit for Java. The Java DOM API for XMLType provides classes that implement W3C DOM interfaces.

• Using the Java DOM API for XMLType

Retrieve data from an XMLType table or column and obtain a Java XMLDocument instance from it. Manipulate elements of the DOM tree for the data using the Java DOM API for XMLType.



Large XML Node Handling with Java

Oracle XML DB provides abstract streams and stream-manipulation methods that you can use to handle XML nodes that are larger than 64 K bytes. Use Java classes XMLNode and XMLAttr, together with a thick or kprb connection, to manipulate large nodes.

Using the Java DOM API and JDBC with Binary XML
 You can use the Java DOM API for XML and Java Database Connectivity (JDBC) 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.

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 Schemabased and non-schema-based documents.

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. As discussed in 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 handles all well-formed XML documents stored in Oracle XML DB. It presents a uniform view of an XML document, whether it is XML Schema-based or non-schema-based and whatever the underlying XMLType storage model. The Java DOM API works on both client and server.

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 schemabased and non-schema-based documents.

To access XMLType data using JDBC, use the class oracle.xdb.XMLType.

The Java DOM API for XMLType is implemented using Java package oracle.xml.parser.v2.

See Also:

Oracle Database XML Java API Reference

Access to XMLType Data Using JDBC

Java Database Connectivity (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 or Java interface java.sql.SQLXML to create XML data.

The JDBC 4.0 standard data type for XML data is java.sql.SQLXML. Method getObject() returns an object of type oracle.xdb.XMLType. Starting with Oracle Database 11g Release 2 (11.2.0.3), oracle.xdb.XMLType implements interface java.sql.SQLXML.



Using JDBC to Access XML Documents in Oracle XML DB JDBC users can guery 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.

Using 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 13-1 illustrates how to use JDBC to query an XMLType table.

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.jdbc.OracleClob or java.lang.String instance in Java. The Java snippet in Example 13-2 illustrates this.
- Call method getSQLXML() in the ResultSet to obtain the whole SQLXML instance. The return value of this method is of type java.sql.SQLXML. Then you can use Java methods in interface SQLXML to access the data. Example 13-3 shows how to do this.

Example 13-3 shows the use of method getObject() to directly obtain an XMLType instance from ResultSet.

Example 13-4 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 13-1 Querying an XMLType Table Using JDBC

```
PreparedStatement statement = connection.prepareStatement(
   "SELECT e.poDoc FROM po_xml_tab e");
ResultSet resultSet = statement.executeQuery();
while(resultSet.next())
{
   // Get result as SQLXML data.
   // Use that to get a DomSource instance.
   SQLXML sqlXml = resultSet.getSQLXML(1);
   DomSource source = sqlXml.getSource(DOMSource.class);
   // Get document from the DomSource instance as a DOM node.
   Document document = (Document) source.getNode();
   // Use the document object
   ...
}
```

Example 13-2 Selecting XMLType Data Using getString() and getCLOB()

```
PreparedStatement statement = connection.prepareStatement(
   "SELECT XMLSerialize(DOCUMENT e.poDoc AS CLOB) poDoc, " +
   "XMLSerialize(DOCUMENT e.poDoc AS VARCHAR2(2000)) poString " +
   " FROM po xml tab e");
```

```
ResultSet resultSet = statement.executeQuery();
while(resultSet.next())
{
    // The first result is an OracleClob instance
    OracleClob clob = resultSet.getClob(1));
    // The second result is a String instance
    String poString = resultSet.getString(2);
    // Use clob and poString
    ...
}
```

Example 13-3 Returning XMLType Data Using getSQLXML()

```
PreparedStatement statement = connection.prepareStatement(
                "SELECT e.poDoc FROM po_xml_tab e");
ResultSet resultSet = statement.executeQuery();
while(resultSet.next())
{
    // Get the SQLXML
    SQLXML sqlXml = resultSet.getSQLXML(1);
    // Convert the SQLXML to an xmlString instance
    String xmlString = sqlXml.getString();
    //Use the xmlString instance
    ...
}
```

Example 13-4 Returning XMLType Data Using an Output Parameter

```
public void doCall (String[] args) throws Exception
{
  // CREATE OR REPLACE FUNCTION getPurchaseOrder(reference VARCHAR2)
  // RETURN XMLTYPE
  // AS
  11
     xml XMLTYPE;
  // BEGIN
  // SELECT OBJECT VALUE INTO xml
  11
        FROM purchaseorder
         WHERE XMLCast(
  11
  11
                 XMLQuery('$p/PurchaseOrder/Reference'
  11
                          PASSING OBJECT VALUE AS "p" RETURNING CONTENT)
  11
                 AS VARCHAR2(30))
  11
               = reference;
  11
         RETURN xml;
 // END;
 String SQLTEXT = "{? = call getPurchaseOrder('BLAKE-2002100912333601PDT')}";
  super.doSomething(args);
  createConnection();
```



```
try
  {
    System.out.println("SQL := " + SQLTEXT);
    CallableStatement sqlStatement = getConnection().prepareCall(SQLTEXT);
    sqlStatement.registerOutParameter (1, java.sql.Types.SQLXML);
    sqlStatement.execute();
   SQLXML sqlXml = sqlStatement.getSQLXML(1);
    System.out.println(sqlXml.getString());
  }
  catch (SQLException exception)
  {
    if (sqlStatement != null)
    {
     sqlStatement.close();
     throw exception;
    }
  }
}
```

Manipulating XML Database Documents Using JDBC

You can update, insert, and delete XMLType data stored in the database using Java Database Connectivity (JDBC) with Oracle XML DB.

```
Note:
XMLType method transform() works 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 string to an INSERT, UPDATE, or DELETE statement, and use the XMLType constructor inside SQL to construct the XML instance. Example 13-5 illustrates this.
- Use setSQLXML() in a PreparedStatement instance to set an entire XMLType instance.
 Example 13-6 illustrates this.

When selecting SQLXML values, JDBC describes the column as SQLXML. You can select the column type name and compare it with SQLXML to see whether you are dealing with a SQLXML instance. Example 13-7 illustrates this.

Example 13-8 updates element discount inside element PurchaseOrder stored in an XMLType column. It uses JDBC and SQLXML. It uses the XML parser to update a DOM tree and write the updated XML value to the XMLType column.

Example 13-9 shows the updated purchase order that results from Example 13-8.

Example 13-5 Updating an XMLType Column Using SQL Constructor XMLType and Java String

```
PreparedStatement statement =
    connection.prepareStatement(
        "UPDATE po_xml_tab SET poDoc = XMLType(?)");
String poString = "<PO><PONO>200</PONO><PNAME>PO_2</PNAME></PO>";
// Bind the string
statement.setString(1,poString);
statement.execute();
```

Example 13-6 Updating an XMLType Column Using SQLXML

```
PreparedStatement statement =
    connection.prepareStatement("UPDATE po_xml_tab SET poDoc = ?");
String xmlString = "<PO><PONO>200</PONO><PNAME>PO_2</PNAME></PO>";
SQLXML sqlXml = connection.createSQLXML();
sqlXml.setString(xmlString);
// Bind the SQLXML
statement.setSQLXML(1, sqlXml);
```

```
statement.execute();
```

Example 13-7 Retrieving Metadata About an XMLType Column Using JDBC

```
PreparedStatement statement =
    connection.prepareStatement("SELECT poDoc FROM po_xml_tab");
ResultSet resultSet = statement.executeQuery();
// Get the resultSet metadata
ResultSetMetaData mdata = (ResultSetMetaData)resultSet.getMetaData();
// The column type is SQLXML
if (mdata.getColumnType(1) == java.sql.Types.SQLXML)
{
    // It is a SQLXML instance
}
```

Example 13-8 Updating an XMLType Column Using JDBC

```
public class UpdateXMLType
{
  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)
   {
}
```



```
System.out.println("\n===========");
 System.out.println(xmlTypeStr);
 String outXML = null;
 try
 {
   DOMParser parser = new DOMParser();
   parser.setValidationMode(false);
   parser.setPreserveWhitespace (true);
   parser.parse(new StringReader(xmlTypeStr));
   System.out.println("XML string is well-formed");
   XMLDocument document = parser.getDocument();
   NodeList nl = document.getElementsByTagName("DISCOUNT");
   for(int i=0;i<nl.getLength();i++)</pre>
                                       {
     XMLElement discount = (XMLElement)nl.item(i);
     XMLNode textNode = (XMLNode)discount.getFirstChild();
     textNode.setNodeValue("10");
   }
   StringWriter sw = new StringWriter();
   document.print(new PrintWriter(sw));
   outXML = sw.toString();
   //Print modified xml
   System.out.println("\n=========");
   System.out.println("Updated PurchaseOrder:");
   System.out.println(outXML);
   }
 catch (Exception e)
  {
   e.printStackTrace(System.out);
 }
 return outXML;
}
public static void main(String args[]) throws Exception
{
 try
  {
   PreparedStatement statement = connection.createStatement();
   ResultSet resultSet = statement.executeQuery(qryStr);
   while(orset.next())
   {
     //retrieve PurchaseOrder xml document from database
     SQLXML sqlXml = resultSet.getSQLXML(1);
     //store this PurchaseOrder in po xml hist table
     statement = connection.prepareStatement(
                  "INSERT INTO po xml hist VALUES(?)");
     statement.setSQLXML(1,sqlXml); // bind the SQLXML instance
     statement.execute();
```

```
//update "DISCOUNT" element
   String newXML = updateXML(sqlXml.getString());
   //\ {\rm create} a new instance of an XMLtype from the updated value
   SQLXML sqlXml2 = connection.createSQLXML();
   sqlXml2.setString(newXml);
   // update PurchaseOrder xml document in database
   statement = connection.prepareStatement(
                   "UPDATE po xml tab x SET x.poDoc =? WHERE " +
                   "XMLCast(XMLQuery('/PO/PONO/text()'" +
                   " PASSING value(xmltab) RETURNING CONTENT)" +
                   " AS NUMBER) " +
                   "= 200");
   statement.setSQLXML(1, sqlXml2); // bind the XMLType instance
   statement.execute();
   connection.commit();
   System.out.println("PurchaseOrder 200 Updated!");
  }
 //delete PurchaseOrder 1001
 statement.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);
}
```

Example 13-9 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>
    <OUANTITY>2</OUANTITY>
    <DISCOUNT>10</DISCOUNT>
   </LINEITEM TYP>
  </LINEITEMS>
  <SHIPTOADDR>
   <STREET>55 Madison Ave</STREET>
   <CITY>Madison</CITY>
  <STATE>WI</STATE>
  <ZIP>53715</ZIP>
  </SHIPTOADDR>
</PurchaseOrder>
```

Loading a Large XML Document into the Database Using JDBC

To load a large XML document into the database using Java Database Connectivity (JDBC), use a Java CLOB object to hold the document, and use Java method insertXML() to perform the insertion.

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 OracleClob object to hold the large XML document. Example 13-10 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 an OracleClob object that contains the XML document. Interface OracleClob is a sub-interface of the standard JDBC interface java.sql.Clob. Method insertXML() binds the OracleClob 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:

- xmlString XML data to be inserted into the XMLType column
- connection database connection object (Oracle Connection Object)

Java method <code>insertXML()</code> calls method <code>getCLOB()</code> to create and return the <code>CLOB</code> object that holds the XML data. The formal parameters of method <code>getCLOB()</code>, which is defined in Example 13-11, are as follows:

- xmlString XML data to be inserted into the XMLType column
- connection database connection object (Oracle Connection Object)

```
See Also:
Oracle Database SecureFiles and Large Objects Developer's Guide
```

Example 13-10 Inserting an XMLType column using JDBC

```
private void insertXML (Connection connection, String xmlString)
{
 OracleClob clob = null;
 try
  {
   String query = "INSERT INTO potable (purchaseOrder) VALUES (XMLType(?)) ";
    // Get the statement Object
    PreparedStatement statement = connection.prepareStatement(query);
    // Get the OracleClob instance from xmlString
    clob = getOracleClob(connection, xmlString);
    statement.setObject(1, clob);
    // Execute the prepared statement
   if (statement.executeUpdate () == 1)
    {
      System.out.println ("Successfully inserted a Purchase Order");
    }
  }
  catch (Exception exp)
  {
   exp.printStackTrace();
  }
  finally
 {
   if(clob !=null)
     clob.close();
  }
}
```

Example 13-11 Converting an XML String to an OracleClob Instance

```
private OracleClob getOracleClob(Connection connection, String xmlString) throws
SQLException
{
    OracleClob clob = (OracleClob) connection.createClob();
    clob.setString(1, xmlString);
    return clob;
}
```



MS Windows Java Security Manager Permissions for Java DOM API with a Thick Connection

If you use Java Security Manager (class SecurityManager) on MS Windows to implement a security policy for your application, then you must add certain permissions to your security policy file, in order to use the Java DOM API for XMLType with a thick connection.

Example 13-12 shows the contents of such a policy file, where the workspace folder that contains the jars related to Oracle XML DB is c:\myworkspace. (The policy file must be in the same folder.)

The libraries used in Example 13-12 are orageneric12 and oraxml12. The last two characters (12 here) must correspond to your major database release number (so for Oracle Database 13 Release 2, for example, you would use orageneric13 and oraxml13).

After you have created the policy file, you can invoke your program using the following command-line switches:

```
-Djava.security.manager=default -
Djava.security.policy=c:\myworkspace\ojdbc.policy
```

Example 13-12 Policy File Granting Permissions for Java DOM API

```
grant codeBase "file:c:\myworkspace" {
   permission java.lang.RuntimePermission "loadLibrary.orageneric12";
   permission java.lang.RuntimePermission "loadLibrary.oraxml12";
}
grant codeBase "file:c:\myworkspace\xdb6.jar" {
   permission java.lang.RuntimePermission "loadLibrary.orageneric12";
   permission java.lang.RuntimePermission "loadLibrary.oraxml12";
}
grant codeBase "file:c:\myworkspace\ojdbc6.jar" {
   permission java.lang.RuntimePermission "loadLibrary.orageneric12";
   permission java.lang.RuntimePermission "loadLibrary.oraxml12";
}
```

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. 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 13-13 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 13-13 Creating a DOM Object with the Java DOM API

```
PreparedStatement statement =
  connection.prepareStatement(
    "update po xml XMLTypetab set poDoc = ? ");
String xmlString = "<PO><PONO>200</PONO><PNAME>PO 2</PNAME></PO>";
OracleClob clob = (OracleClob)connection.createClob();
clob.setString(1, xmlString);
SQLXML sqlXml = clob.toSQLXML();
DOMSource domSource = sqlXml.getSource(DOMSource.class);
Document document = (Document) domSource.getNode();
Element rootElem = document.createElement("PO");
document.insertBefore(document, rootElem, null);
SQLXML sqlXml2 = clob.toSQLXML();
DOMResult domResult = sqlXml2.setResult(DomResult.class);
domResult.setNode(document);
statement.setSQLXML(1, sqlXml2);
statement.execute();
```

XMLType Instance Representation in Java (JDBC or SQLJ)

An XMLType instance is represented in Java by oracle.xdb.XMLType. When an instance of XMLType is fetched using JDBC or a SQLJ client, it is automatically manifested as an object of the provided XMLType class.

You can bind objects of this class as values to Data Manipulation Language (DML) statements where an XMLType is expected.

Classes of Java DOM API for XMLType

Oracle XML DB supports the W3C DOM Level 2 Recommendation. It also provides Oraclespecific extensions, to facilitate interfacing your application with Oracle XML Developer's Kit for Java. The Java DOM API for XMLType provides classes that implement W3C DOM interfaces.



XMLDocument is a class that represents the DOM for the instantiated XML document. You can retrieve a SQLXML instance from a document and a connection object as follows:

```
SQLXML sqlXml = connection.createSQLXML();
DOMResult domResult = sqlXml.setResult(DOMResult.class);
domResult.setNode(document);
```

Table 13-1 lists the Java DOM API for XMLType classes and the W3C DOM interfaces they implement. The Java DOM API classes are in package oracle.xml.parser.v2.

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		
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		

Table 13-1Java DOM API for XMLType: Classes

See Also:

Oracle XML DB on OTN for Oracle extensions for interfacing an application with Oracle XML Developer's Kit for Java

Using the Java DOM API for XMLType

Retrieve data from an XMLType table or column and obtain a Java XMLDocument instance from it. Manipulate elements of the DOM tree for the data using the Java DOM API for XMLType.

The Java DOM API for XMLType lets you find and retrieve nodes within a document at any level. You can use it to create XML documents programmatically, including on the fly (dynamically). These documents can conform to a registered XML schema or not. The Java API for XMLType conforms to the DOM 2.0 recommendation, and it is namespace-aware.

Figure 13-1 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 a Document instance. You can then use method getNode() to retrieve an XMLDocument instance.
- 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 free() in the respective classes. This frees any underlying memory that is held.

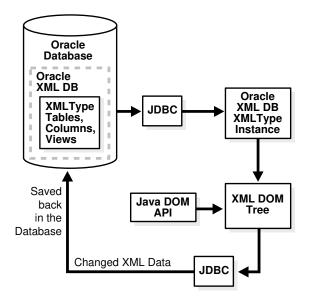


Figure 13-1 Using the Java DOM API for XMLType

Large XML Node Handling with Java

Oracle XML DB provides abstract streams and stream-manipulation methods that you can use to handle XML nodes that are larger than 64 K bytes. Use Java classes XMLNode and XMLAttr, together with a thick or kprb connection, to manipulate large nodes.

¹ This assumes that your XML data is pre-registered with an XML schema, and that it is stored in an XMLType column.

Note:

The large-node feature works only with a thick or kprb connection. It does not work with a thin connection.

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.

The former restrictions on the size of nodes were 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);
```

Prior to Release 11.1, 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);
```

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.

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.

Related Topics

Large Node Handling Using DBMS_XMLDOM

Oracle XML DB provides abstract streams and stream-manipulation methods that you can use to handle XML nodes that are larger than 64 K bytes.

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

You can use methods getNodeValueAsBinaryStream() and

getNodeValueAsCharacterStream() to retrieve the value of a DOM node, using a parser



that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.

- Get-Push Model In this model, you retrieve the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.
- Set-Pull Model

In this model, you set the value of a DOM node, using a parser that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.

Set-Push Model

In this model, you set the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.

Get-Pull Model

You can use methods getNodeValueAsBinaryStream() and getNodeValueAsCharacterStream() to retrieve the value of a DOM node, using a parser that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.

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

In this model, you retrieve the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.

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

In this model, you set the value of a DOM node, using a parser that is in pull mode. Oracle XML DB reads the event data from an input stream written by the parser.

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

In this model, you set the value of a DOM node, using a parser that is in push mode. Oracle XML DB writes the node data to an output stream that the parser reads.

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 XML Developer's Kit.

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 XML Developer's Kit.

```
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();
```

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().

See Also:

- Oracle Database XML Java API Reference
- Oracle Database XML C API Reference for information about C functions for large nodes

Using the Java DOM API and JDBC with Binary XML

You can use the Java DOM API for XML and Java Database Connectivity (JDBC) 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.

XML data can be stored in Oracle XML DB using data type XMLType, and one of the storage models for this abstract data type is binary XML, a compact, XML Schema-aware encoding of XML data. You can use binary XML 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.

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.



 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 be encoded in the binary XML format. This applies to a DOM document (class oracle.xdb.XMLType). If you do not specify binary XML, the data is sent to the database as text.
- 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 13-14 illustrates this.

See Also: Oracle XML Developer's Kit Programmer's Guide

Example 13-14 Using the Java DOM API with a Binary XML Column

```
class PrintBinaryXML
{
    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 connection = 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(connection);

```
// Query XML data stored in SQLXML column as binary XML
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT doc FROM po_binxmltab");
// Get the SQLXML object
while (resultSet.next())
{
    SQLXML sqlXml = resultSet.getSQLXML(1);
    // Convert SQLXML to a String
    String xmlString = sqlXml.getString();
    System.out.println(xmlString);
}
resultSet.close();
statement.close();
System.out.println("Completed Binary XML Java Example");
```

Related Topics

}

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.



14 C DOM API for XMLType

XML.

The C DOM API for XMLType lets you operate on XMLType instances using a DOM in C.

- Overview of the C DOM API for XMLType The C DOM API for XMLType is a DOM API that is used for Oracle XML Developer's Kit (XDK) and Oracle XML DB. You can use it for XML data that is inside or outside the database.
- Access to XMLType Data Stored in the Database Using OCI 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
- Creating XMLType Instances on the Client You can construct new XMLType instances on the client side using the C DOM API methods XMLCreateDocument() and XmlLoadDom().
- XML Context Parameter for C DOM API Functions An XML context is a required parameter for 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 XML Developer's Kit applications and Oracle XML DB.
- Initializing and Terminating an XML Context
 An example illustrates a C program that uses the C DOM API to construct an XML
 document and save it to Oracle Database.
- Using the C API for XML with Binary XML
 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.
- Using the Oracle XML Developer's Kit Pull Parser with Oracle XML DB You can use the Oracle XML Developer's Kit 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.
- Common XMLType Operations in C Common XML operations are provided by the C API for XML.

Overview of the C DOM API for XMLType

The C DOM API for XMLType is a DOM API that is used for Oracle XML Developer's Kit (XDK) and Oracle XML DB. You can use it for XML data that is inside or outside the database.

DOM refers to compliance with the World Wide Web Consortium (W3C) DOM 2.0 Recommendation.

The C DOM API for XMLType also includes performance-improving extensions that you can use in XDK for traditional storage of XML data, or in Oracle XML DB for storage as an XMLType column in a table.



Note:

C DOM functions from releases prior to Oracle Database 10g Release 1 are supported only for backward compatibility.

The C DOM API for XMLType 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 DOM API for XMLType is a combined programming interface that includes all of the functionality needed by Oracle XML Developer's Kit 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().

The C DOM API for XMLType 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.

The reference documentation for the C and C++ Application Programming Interfaces (APIs) that you can use to manipulate XML data is *Oracle Database XML C API Reference*, and *Oracle Database XML C++ API Reference*.

See Also:

- Oracle Database PL/SQL Packages and Types Reference for information about PL/SQL APIs for XML
- Oracle Database XML Java API Reference for information about Java APIs for XML

Access to XMLType Data Stored in the Database Using OCI

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.

Related Topics

XML Context Parameter for C DOM API Functions

An *XML context* is a required parameter for 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 XML Developer's Kit applications and Oracle XML DB.

Creating XMLType Instances on the Client

You can construct new XMLType instances on the client side using the C DOM API methods XMLCreateDocument() and XmlLoadDom().

You can construct empty XMLType instances using XMLCreateDocument(). This is similar to using OCIObjectNew() for other types.

You construct a non-empty XMLType instance using XmlLoadDom(), as follows:

- 1. Initialize the xmlctx as in Example 14-1.
- 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.
- If required, you can cast (xmldocnode *) to (void*) and provide it directly as the bind value.

XML Context Parameter for C DOM API Functions

An *XML context* is a required parameter for 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 XML Developer's Kit applications and Oracle XML DB.

For Oracle XML DB, OCI functions OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx(), respectively, initialize and terminate an XML context.

- OCIXmlDbInitXmlCtx() Syntax
 OCI function OCIXmlDbInitXmlCtx() initializes an XML context.
- OCIXmlDbFreeXmlCtx() Syntax OCI function OCIXmlDbFreeXmlCtx() terminates an XML context.

OCIXmlDbInitXmlCtx() Syntax

OCI function OCIXmlDbInitXmlCtx() initializes an XML context.

The syntax of OCIXmlDbInitXmlCtx() is as follows:

xmlctx	*OCIXmlDbInitXMlCtx	(OCIEnv	*envhp,
		OCISvcHp	*svchp,
		OCIError	*errhp,
		ocixmldbparam	*params,
		ub4	<pre>num_params);</pre>

Table 14-1 describes the parameters.

Table 14-1	OCIXmIDbInitXMICtx() Parameters
------------	---------------------------------

Parameter	Description	
envhp (IN)	The OCI environment handle.	



Parameter	Description
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:
	<pre>void (*err_handler) (sword errcode,</pre>
num_params (IN)	Number of parameters to be read from params.

Table 14-1 (Cont.) OCIXmIDbInitXMICtx() Parameters

OCIXmlDbFreeXmlCtx() Syntax

OCI function OCIXmlDbFreeXmlCtx() terminates an XML context.

The syntax of OCIXmlDbFreeXmlCtx() is as follows, where parameter xctx (IN) is the XML context to terminate.:

void OCIXmlDbFreeXmlCtx (xmlctx *xctx);

Initializing and Terminating an XML Context

An example illustrates a C program that uses the C DOM API to construct an XML document and save it to Oracle Database.

Example 14-1 shows this. The document constructed is stored in table my_table. OCI functions OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx() are used to initialize and terminate the XML context, respectively. These functions are defined in header file ocixmldb.h.

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 Oracle-Supplied XML Schemas and Examples, Initializing and Terminating an XML Context (OCI).

The C code in Example 14-1 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 14-4 queries table my table to show the data that was inserted by Example 14-1.



Example 14-1 Using OCIXMLDBINITXMLCTX() and OCIXMLDBFREEXMLCTX()

```
#ifndef S ORACLE
#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 */
static sword exec bind xml(OCISvcCtx *svchp,
                           OCIError *errhp,
                           OCIStmt *stmthp,
                           void *xml,
                           OCIType *xmltdo,
                           OraText *sqlstmt);
/* Helper function 2: Initialize OCI handles and connect */
static sword init_oci_handles(test ctx *ctx);
/* Helper function 3: Free OCI handles and disconnect */
static 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. 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. 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,
```

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.



}

Using the C API for XML with Binary XML

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 a compact, XML Schema-aware encoding of XML data. You can use binary XML as a storage model for XMLType data in the database, but you can also use it for XML data located outside the database.

Binary XML data is XML Schema-aware, and it can use various encoding schemes, depending on your needs. 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 (OCISVeCtx) 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. As is illustrated by Example 14-2, the overall sequence of actions is 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.

See Also:

Oracle XML Developer's Kit Programmer's Guide

```
/* 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
static void checkerr(OCIError *errhp, sword status);
static sword create env(OraText *user, ub2 user len, OraText *pwd, ub2 pwd len,
                      OCIEnv **envhp, OCISvcCtx **svchp, OCIError **errhp);
static sword run example (OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp,
                        OCIDuration dur);
static void cleanup(OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp);
int main (int argc, char *argv[])
{
           *envhp;
 OCIEnv
 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;
}
static 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,
```

Example 14-2 Using the C API for XML with Binary XML



```
(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;
}
static sword run example (OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp,
                        OCIDuration dur)
{
          *xmltdo = (OCIType *)0;
 OCIType
 OCIStmt *stmthp;
 OCIDefine *defnp;
 xmldocnode *xmldoc = (xmldocnode *)0;
 ub4
           xmlsize = 0;
 text
            *selstmt = (text *)"SELECT doc FROM po binxmltab";
        status;
 sword
 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 */
   . .
}
```

Related Topics

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.

Using the Oracle XML Developer's Kit Pull Parser with Oracle XML DB

You can use the Oracle XML Developer's Kit 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 XML Developer's Kit pull parser to perform stream-based XML Schema validation.

Example 14-3 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 XML Developer's Kit pull parser
- Oracle XML Developer's Kit Programmer's Guide for information on using the pull parser for stream-based validation

Example 14-3 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";
 /* 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] = ' \setminus 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>Salesperson</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: Salesperson
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

Common XML operations are provided by the C API for XML.

Table 14-2 provides the XMLType functional equivalent of common XML operations.

Table 14-2	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 stylesheet	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 14-4 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 Oracle-Supplied XML Schemas and Examples, Example A-6 for the definitions of helper functions <code>exec_bind_xml</code>, free_oci_handles, and <code>init_oci_handles</code>.

Example 14-4 Using the DOM to Count Ordered Parts

```
#ifndef S_ORACLE
#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 */
static sword exec bind xml(OCISvcCtx *svchp,
                           OCIError *errhp,
                           OCIStmt *stmthp,
                           void *xml,
                           OCIType *xmltdo,
                           OraText *sqlstmt);
/* Helper function 2: Initialize OCI handles and connect */
static sword init oci handles(test ctx *ctx);
/* Helper function 3: Free OCI handles and disconnect */
static 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
```



}

15

Oracle XML DB and Oracle Data Provider for .NET

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.

- Oracle XML DB and ODP.NET XML
 ODP.NET supports XML data natively in the database, through Oracle XML DB.
- Using XMLType Data with ODP.NET An example illustrates passing XMLType data from the database to .NET

Oracle XML DB and ODP.NET XML

ODP.NET supports XML data natively in the database, through Oracle XML DB.

ODP.NET:

- 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.
- Provides the following XML-specific classes:
 - OracleXmlType
 - OracleXmlStream
 - OracleXmlQueryProperties
 - OracleXmlSaveProperties
- Enhances classes OracleCommand, OracleConnection, and OracleDataReader.

Using XMLType Data with ODP.NET

An example illustrates passing XMLType data from the database to .NET

Example 15-1 retrieves XMLType data from the database to .NET and outputs the results:



See Also:

Oracle Data Provider for .NET Developer's Guide. for complete information about Oracle .NET support for Oracle XML DB.

Example 15-1 Retrieve XMLType Data to .NET

```
//Create OracleCommand and query XMLType
OracleCommand xmlCmd = new OracleCommand();
poCmd.CommandText = "SELECT po FROM po tab";
poCmd.Connection = conn;
//\ {\tt 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);
```

Part V

•

•

XML Schema and Object-Relational XMLType

The use of XML Schema and object-relational storage of XMLType data is covered.

- Choice of XMLType Storage and Indexing Important design choices for your application include what XMLType storage model to use and which indexing approaches to use.
- XML Schema Storage and Query: Basic XML Schema is a standard for describing the content and structure of XML documents. You can register, update, and delete an XML schema used with Oracle XML DB. You can define storage structures to use for your XML schema-based data and map XML Schema data types to SQL data types.
- XML Schema Storage and Query: Object-Relational Storage

Advanced techniques for XML Schema-based data include using object-relational storage; annotating XML schemas; mapping Schema data types to SQL; using complexType extensions and restrictions; creating, specifying relational constraints on, and partitioning XML Schema-based data, storing XMLType data out of line, working with complex or large schemas, and debugging schema registration.

XPath Rewrite for Object-Relational Storage For XMLType data stored object-relationally, queries involving XPath expression arguments to various SQL functions can often be automatically rewritten to queries against the underlying SQL tables, which are highly optimized.

XML Schema Evolution You can use XML schema evolution to update your XML schema after you have registered it with Oracle XML DB.



16 Choice of XMLType Storage and Indexing

Important design choices for your application include what XMLType storage model to use and which indexing approaches to use.

- Introduction to Choosing an XMLType Storage Model and Indexing Approaches XMLType is an abstract SQL data type that provides different storage and indexing models to best fit your XML data and your use of it. Because it is an abstract data type, your applications and database queries gain in flexibility: the same interface is available for all XMLType operations.
- XMLType Use Case Spectrum: Data-Centric to Document-Centric When choosing an XMLType storage model, consider the nature of your XML data and the ways you use it. There is a spectrum of use cases, ranging from most data-centric to most document-centric.
- Common Use Cases for XML Data Stored as XMLType Recommendations are provided for application use cases that correspond to common use cases for XML data stored as XMLType.
- XMLType Storage Model Considerations
 For most use cases, Oracle recommends that you use binary XML storage of XMLType.
 Object-relational storage is appropriate in special cases.
- XMLType Indexing Considerations For XMLType data stored object-relationally, create B-tree and bitmap indexes just as you would for relational data. Use XMLIndex indexing with XMLType data that is stored as binary XML.
- XMLType Storage Options: Relative Advantages
 Each XMLType storage model has particular advantages and disadvantages.

Introduction to Choosing an XMLType Storage Model and Indexing Approaches

XMLType is an abstract SQL data type that provides different storage and indexing models to best fit your XML data and your use of it. Because it is an abstract data type, your applications and database queries gain in flexibility: the same interface is available for all XMLType operations.

Different applications use XML data in different ways. Sometimes it is constructed from relational data sources, so it is relatively structured. Sometimes it is used for extraction, transformation, and loading (ETL) operations, in which case it is also quite structured. Sometimes it is used for free-form documents (unstructured or semi-structured) such as books and articles.

Retrieval approaches can also be different for different kinds of data. Data-centric use cases often involve a fixed set of queries, whereas document-centric use cases often involve arbitrary (ad-hoc) queries.

Because there is a broad spectrum of XML usage, there is no one-size-fits-all storage model that offers optimal performance and flexibility for every use case. Oracle XML DB offers two



storage models for XMLType, and several indexing methods appropriate to these different storage models. You can tailor performance and functionality to best fit the kind of XML data you have and the ways you use it.

Therefore, one key decision to make is which XMLType storage model to use for which XML data. This chapter helps you choose the best storage option for a given use case.

XMLType tables and columns can be stored in the following ways:

- Compact Schema-aware Binary XML (CSX) storage (the default) This is also referred to as post-parse persistence. It is the default storage model for Oracle XML DB. It is a post-parse, binary format designed specifically for XML data. Binary XML is compact and XML schema-aware. The biggest advantage of Binary XML storage is *flexibility*: you can use it for XML schema-based documents or for documents that are not based on an XML schema. You can use it with an XML schema that allows for high data variability or that evolves considerably or unexpectedly. This storage model also provides efficient partial updating and streamable query evaluation.
- Transportable Binary XML (TBX) storage XMLType storage option, a variant built on top of compact schema-aware Binary XML (CSX). Oracle Database 23ai introduces Transportable Binary XML (TBX), which is scalable and supports sharding, TBX data replication, and search index. User can create sharded tables with TBX columns, but not sharded TBX tables. User can also create virtual TBX columns in sharded tables, but they cannot be a sharded key.
- Object-relational storage This is also referred to as structured storage and objectbased persistence. This storage model represents an entity-relationship (ER) decomposition of the XML data. It provides the best performance for highly structured data with a known and more or less fixed set of queries. Query performance matches that of relational data, and updates can be performed in place.

Note:

Starting with Oracle Database 12c Release 1 (12.1.0.1), the unstructured (CLOB) storage model for XMLType is *deprecated*. Use binary XML storage instead.

If you have exising XMLType data that is stored as CLOB data then consider moving it to binary XML storage format using Oracle GoldenGate. If document fidelity is important for a particular XML document then store a copy of it in a relational CLOB column.

Oracle XML DB supports the following kinds of indexes on XMLType data.

- B-tree functional indexes on object-relational storage
- XML search index on binary XML storage
- XMLIndex with structured and unstructured components on binary XML storage
- B-tree indexes on the secondary tables created automatically for XMLIndex (both structured and unstructured components) on binary XML storage

Different use cases call for different combinations of XMLType storage model and indexes.



Related Topics

Indexes for 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. There are various ways that you can index XMLType data, whether it is XML schema-based or non-schema-based, and regardless of the XMLType storage model you use.

• XMLType Storage Options: Relative Advantages Each XMLType storage model has particular advantages and disadvantages.

XMLType Use Case Spectrum: Data-Centric to Document-Centric

When choosing an XMLType storage model, consider the nature of your XML data and the ways you use it. There is a spectrum of use cases, ranging from most data-centric to most document-centric.

This is illustrated in Figure 16-1 , which shows the most data-centric cases at the left and the most document-centric cases at the right.

Figure 16-1 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	Variable, free-form data, with some fixed embedded structures	Variable, free-form data
Typical Data	Employee record	Technical article, with author, date, and title fields	Web document or book chapter
Storage Model	Object-Relational (Structured)	Binary XML	
Indexing	B-tree index	XMLIndex index with structured and unstructured components XML search index	XMLIndex index with unstructured component XML search index

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

Document-centric data can be divided into two cases:

- The data is generally without structure or is of variable structure. This includes the case of documents that have both structured and unstructured parts. Document structure can vary over time (evolution), and the content can be **mixed** (**semi-structured**), with many elements containing 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.
- The data is relatively structured, but your applications do not take advantage of that structure: they treat the data as if it were without structure.



Common Use Cases for XML Data Stored as XMLType

Recommendations are provided for application use cases that correspond to common use cases for XML data stored as XMLType.

If your use case is *not* a common one, so that it is not covered here, then refer to the rest of this chapter for information about special cases.

Note:

This section is about the use of XML data that is persisted as XMLType. One common use case for XML data involves the generation of XML data from relational data. That case is not covered here, as it involves relational storage and the generated XML data is not necessarily persisted.

(For cases where generated XML data is persisted as XMLType, see XMLType Use Case: Staged XML Data for ETL.)

XMLType Use Case: No XML Fragment Updating or Querying

In this use case there is no requirement to update or query fragments of XML data that is stored in the database.

- XMLType Use Case: Data Integration from Diverse Sources with Different XML Schemas If your XML data comes from multiple data sources that use different XML schemas then use binary XML storage.
- XMLType Use Case: Staged XML Data for ETL

In this use case, data is extracted from outside sources, transformed to fit operational needs (typically relational), and then loaded into the database: *extract, transform, load* (ETL). In particular, transformation distinguishes this use case.

• XMLType Use Case: Semi-Structured XML Data

In this use case, either your XML data is of variable form or large portions of it are not well defined. There might not be an associated XML schema, or the XML schema might allow for high data variability or evolve considerably or in unexpected ways.

- XMLType Use Case: Business Intelligence Queries
 To enable business-intelligence (BI) queries over XML data, you can use SQL/XML
 function XMLTable to project values contained in the data as columns of a virtual table.
 Then use analytic-function windows, together with SQL ORDER BYand GROUP BY, to operate
 on columns of the virtual table.
- XMLType Use Case: XML Queries Involving Full-Text Search If your application needs to perform full-text searches on XML data then use *binary XML storage* and create XML search indexes that correspond to your queries.

Related Topics

- XMLType Indexing Considerations
 For XMLType data stored object-relationally, create B-tree and bitmap indexes just as you
 would for relational data. Use XMLIndex indexing with XMLType data that is stored as binary
 XML.
- XMLType Storage Options: Relative Advantages Each XMLType storage model has particular advantages and disadvantages.

XMLType Use Case: No XML Fragment Updating or Querying

In this use case there is no requirement to update or query fragments of XML data that is stored in the database.

You have these options for this use case:

- Store it as XMLType using binary XML storage.
- Store it in a *relational* BLOB or CLOB column, preferably a SecureFiles LOB.

If you store the XML data in a relational LOB column, not as XMLType, Oracle Database does not parse the data and it cannot guarantee its validity. (And you cannot perform XMLType operations on the data.)

XMLType Use Case: Data Integration from Diverse Sources with Different XML Schemas

If your XML data comes from multiple data sources that use different XML schemas then use *binary XML storage*.

This use case has three subcases:

- If the XML data contains islands of structured, predictable data, and your queries are known, then use XMLIndex with a structured component to index the structured islands (even if the data surrounding these islands is unstructured). A structured index component reflects the queries you use. An RSS news aggregator is an example of such a use case.
- If there are no such structured islands or your queries are unknown ahead of time (ad hoc) then use XMLIndex with an unstructured component.
- If you use queries that involve full-text search then use an XML search index, together with XQuery pragma ora:no_schema.

Related Topics

• 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.

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.

Indexing XML Data for Full-Text Queries (pre-23ai)

When you need full-text search over XML data, Oracle recommends that you store your XMLType data as binary XML and you use XQuery Full Text (XQFT). You use an XML search index for this. This is the topic of this section.

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 several such pragmas.



XMLType Use Case: Staged XML Data for ETL

In this use case, data is extracted from outside sources, transformed to fit operational needs (typically relational), and then loaded into the database: *extract, transform, load* (ETL). In particular, transformation distinguishes this use case.

ETL use cases often integrate data from multiple applications that are maintained or hosted by multiple parties using different software and hardware systems. The data that is extracted is often the responsibility of parties other than those who transform it or use it after transformation.

The XML data involved is typically highly structured and conforms to an XML schema. This use case covers both producing relational data from XML data and generating XML data from relational data.

A subset of ETL use cases involve the need to efficiently *update* the XML data. Updating can involve replacement of an entire XML document or changes to only fragments of a document (partial updating).

Object-relational storage of XMLType data is generally appropriate for this use case.

Related Topics

- Relational Views over XML Data Relational database views over XML data provide conventional, relational access to XML content.
- Generation of XML Data from Relational Data

Oracle XML DB provides features for generating (constructing) XML data from relational data in the database. There are both SQL/XML standard functions and Oracle-specific functions and packages for generating XML data from relational content.

XMLType Use Case: Semi-Structured XML Data

In this use case, either your XML data is of variable form or large portions of it are not well defined. There might not be an associated XML schema, or the XML schema might allow for high data variability or evolve considerably or in unexpected ways.

Binary XML storage of XMLType data is generally appropriate for this use case.

Use structured-component XMLIndex indexing when query paths are known, and use pathsubsetted unstructured-component XMLIndex indexing when paths are not known beforehand (ad hoc queries). Use an XML search index for XQuery Full-Text queries.

Related Topics

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.

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.



Indexing XML Data for Full-Text Queries (pre-23ai)

When you need full-text search over XML data, Oracle recommends that you store your x_{MLType} data as binary XML and you use XQuery Full Text (XQFT). You use an XML search index for this. This is the topic of this section.

XMLType Use Case: Business Intelligence Queries

To enable business-intelligence (BI) queries over XML data, you can use SQL/XML function XMLTable to project values contained in the data as columns of a virtual table. Then use analytic-function windows, together with SQL ORDER BYand GROUP BY, to operate on columns of the virtual table.

For business-intelligence queries, you will generally do all of the following:

- Store your XMLType data as binary XML.
- Use an XMLIndex index with a structured component.
- Create relational views over the data using SQL/XML function XMLTable, where the views
 project all columns of interest to the BI application.
- Write your application queries against these relational views.

If the XMLIndex index is created in one-to-one correspondence to these views, Oracle Database automatically translates queries over the views to queries over the relational tables of the structured XMLIndex component, providing relational performance.

When you use analytic-function windows, ORDER BY, or GROUP BY on a column of the virtual table, these operations are translated to windows, ORDER BY, and GROUP BY operations on the corresponding physical columns of the structured-component XMLIndex tables.

Related Topics

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.

 Relational Views over XML Data Relational database views over XML data provide conventional, relational access to XML content.

XMLType Use Case: XML Queries Involving Full-Text Search

If your application needs to perform full-text searches on XML data then use *binary XML storage* and create XML search indexes that correspond to your queries.

Related Topics

Indexing XML Data for Full-Text Queries (pre-23ai)
 When you need full-text search over XML data, Oracle recommends that you store your XMLType data as binary XML and you use XQuery Full Text (XQFT). You use an XML search index for this. This is the topic of this section.



XMLType Storage Model Considerations

For most use cases, Oracle recommends that you use binary XML storage of XMLType. Object-relational storage is appropriate in special cases.

Object-relational storage is not appropriate unless *all* of the following are true:

- You have an XML schema that rigorously specifies the detailed data format of all XML documents that you intend to store in a given XMLType column or table. Your applications are data-centric.
- You do not expect your XML schema to evolve frequently in ways that do not allow in-place schema evolution.
- Your data is not especially sparse (does not include many elements that are empty or missing).
- You do not necessarily insert and select whole XML documents at a time. Partial updates and selections are common.
- You do not need document fidelity (DOM fidelity is sufficient).

Table 16-1 provides more detail about this. The guidelines it presents for choosing an XMLType storage model are *not* independent: follow them *in the order presented*, row by row, until a requirement in column **If...** is satisfied.

lf	Then
1. You need the property of document fidelity, preserving all original whitespace.	Use binary XML storage for database use and XML processing. But also store a copy of the original documents in a CLOB (relational) column.
	(It is your responsibility to keep the two versions synchronized, if you update the data.)
2. You rarely need to select or update only a portion of your XML data. Instead, you typically insert and select whole XML documents at a time.	Use binary XML storage.
3. You need to store XMLType instances that conform to different XML schemas in the same XMLType table or column.	Use binary XML storage.
(Oracle does <i>not</i> recommend this practice in general, because it prohibits Oracle XML DB from using the XML schemas to optimize XML queries and other operations.)	
4. You do not have an XML schema for your data.	Use binary XML storage. If you think that your data could benefit from XML schema validation, then consider also whether you can generate an XML schema for it using a schema-generation tool.
5. You expect your XML schema to evolve frequently or in unexpected ways, and you <i>cannot</i> take advantage of in-place XML schema evolution. In-place evolution is generally permitted only if the changes do not invalidate existing documents and they do not involve changing the storage model. See XML Schema Evolution.	Use binary XML storage. Use PL/SQL procedure DBMS_XMLSCHEMA.copyEvolve to update the XML schema.

Table 16-1 XMLType Storage Model Considerations



lf	Then
6. Your XML data is very sparse.	Use binary XML storage.
7. Your XML schema does <i>not</i> make use of constructs such as elements any and choice, which do not provide a detailed specification of the data format.	Use object-relational storage.
(XML schema generators often include such constructs in the generated schemas.)	
8. You can modify your XML schema to remove constructs such as any and choice that prevent a rigorous definition of the structure of your XML data.	Remove such constructs, then use object-relational storage.
9. You cannot remove such constructs.	Use binary XML storage.

Table 16-1 (Cont.) XMLType Storage Model Considerations

XMLType Indexing Considerations

For XMLType data stored object-relationally, create B-tree and bitmap indexes just as you would for relational data. Use XMLIndex indexing with XMLType data that is stored as binary XML.

For general indexing of document-centric XML data, use XMLIndex with an *unstructured component*. This is appropriate for queries that are ad hoc (arbitrary).

For data that contains predictable, fixed parts that you query frequently, use XMLIndex with *structured components* for those parts. An example of this use case is a specification that is generally free-form but that has fixed fields for the author, date, and title.

To handle islands of structure within generally unstructured content, create an XMLIndex index that has both structured and unstructured components. 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 structured component of the index would be used for a query WHERE clause condition that checks for the structured data. The unstructured component would be used for the fragment extraction.

Table 16-2 provides simple guidelines for indexing XMLType data that is stored as binary XML. These guidelines are *independent*: you can use a combination of indexing approaches if their **If...** conditions are satisfied.

Table 16-2	XMLType Indexing Considerations
------------	---------------------------------

If	Then
Your data contains predictable islands of structured data.	Use XMLIndex, with a structured component for each of the structured islands.
You need to support full-text queries.	Use XML search indexes.
You need to support ad-hoc XML queries involving predicates.	Use ${\tt XMLIndex},$ with an unstructured component.

XMLType Storage Options: Relative Advantages

Each XMLType storage model has particular advantages and disadvantages.

ORACLE

Table 16-3 summarizes the advantages and disadvantages of each XMLType storage model. Symbols + and – provide a rough indication of strength and weakness, respectively.

Table 16-3	XMLType Storage Models: Relative Advantages
------------	---

Quality	Binary XML Storages: compact schema- aware (CSX) and transportable binary (TBX)	Object-Relational Storage
Throughput	(+) High throughput. Fast DOM loading. There is a slight overhead from the binary encoder/ decoder.	(-) XML decomposition can result in reduced throughput when ingesting or retrieving the entire content of an XML document.
Indexing support	XMLIndex and XML search indexes.	B-tree, bitmap, and Oracle Text indexes on specific elements or attributes.
Queries	(+) Fast when using XMLIndex. Queries that cannot use an index use streaming XPath evaluation, which can also be fast.	(++) Relational query performance. You can create B-tree indexes on the underlying object-relational columns.
Update operations (DML)	(+) In-place, piecewise update for SecureFiles LOB storage.	(++) Relational update performance. Columns are updated in place.
Data flexibility	(+) Flexibility in the structure of the XML documents that can be stored in an XMLType column or table.	(-) Limited flexibility. Only documents that conform to the XML schema can be stored.
XML schema flexibility	(++) Both XML schema-based and non-schema- based documents can be stored. Documents conforming to any XML schemas that have been registered can be stored in the same XMLType table or column.	XML schema can be stored in a given XMLType
	(-)Transportable binary XML (TBX) does not support XML schema.	
Validation upon insert	(++) XML schema-based data can be fully validated when it is inserted, but this takes time.	(+) XML data is partially validated when it is inserted.
Compression and Encryption	(+) Binary XML with SecureFiles LOB storage can be compressed/encrypted.	(++) Each XML element/attribute can be compressed/encrypted individually.

17 XML Schema Storage and Query: Basic

XML Schema is a standard for describing the content and structure of XML documents. You can register, update, and delete an XML schema used with Oracle XML DB. You can define storage structures to use for your XML schema-based data and map XML Schema data types to SQL data types.

The XML Schema Recommendation was created by the World Wide Web Consortium (W3C) to describe the content and structure of XML documents. 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.

Overview of XML Schema

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 as the metadata that describes a class of XML documents.

Overview of Using XML Schema with Oracle XML DB

Oracle XML DB supports registering XML schemas, validating documents against an XML schema, generating XML schemas from SQL object types, mapping from XML Schema to SQL, creating and querying XML Schema-based tables, views, and columns, and automatically inserting data when XML Schema-based documents are inserted into Oracle XML DB Repository.

XML Schema Registration with Oracle XML DB

Before an XML schema can be used by Oracle XML DB, you must register it. It can then be used to create XMLType tables and columns and to validate XML documents. If schema registration fails then the database is restored to the state it had prior to the registration attempt.

Creation of XMLType Tables and Columns Based on XML Schemas

You can create XMLType tables and columns that are constrained to a global element defined by an XML schema. After an XMLType column has been constrained to a particular element and a particular schema, it can only contain documents that are compliant with the schema definition of that element.

Ways to Identify XML Schema Instance Documents

Before an XML document can be inserted into an XML Schema-based XMLType table or column, the associated XML schema must be identified. You can do this when you create the table or column, or you can use XMLSchema-instance to explicitly add the required schema identification to the XML instance document.

• XML Schema Data Types Are Mapped to Oracle XML DB Storage

Data that conforms to an XML schema uses 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.



See Also:

- XML Schema Storage and Query: Object-Relational Storage for more advanced information about using XML Schema with Oracle XML DB
- XPath Rewrite for Object-Relational Storage for information about the optimization of XPath expressions in Oracle XML DB
- XML Schema Part 0: Primer Second Edition for an introduction to XML Schema

Overview of XML Schema

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 as the metadata that describes a class of XML documents.

This documentation refers to an XML Schema instance definition as an **XML schema** (lowercase).

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.

XML Schema Features

XML Schema defines 47 scalar data types, for strong typing of elements and attributes. It supports object-oriented inheritance and extension, so you can design an XML schema with complex objects from base data types. It includes constructs for defining and ordering, default values, mandatory content, nesting, repeated sets, and redefines.

XML Instance Documents

Documents conforming to an XML schema can be considered as instances of the class defined by that XML schema. A common use of an XML schema is to validate that a given such **instance document** conforms to the rules defined by the XML schema.

XML Namespaces and XML Schemas

An XML schema can specify a targetNamespace attribute, whose value is a URL. If omitted, the schema has no target namespace. The target namespace is the namespace for everything defined in the XML schema. The targetNamespace value is typically a URL where the XML schema can be accessed.

 Overview of Editing XML Schemas You can author and edit XML schemas anyway you like.

See Also:

XML Schema Part 0: Primer for a general description of the XML Schema recommendation



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 https://www.w3.org/2001/XMLSchema.

XML Schema Features

XML Schema defines 47 scalar data types, for strong typing of elements and attributes. It supports object-oriented inheritance and extension, so you can design an XML schema with complex objects from base data types. It includes constructs for defining and ordering, default values, mandatory content, nesting, repeated sets, and redefines.

Oracle XML DB supports all of the constructs defined by XML Schema, except for redefines.

XML Instance Documents

Documents conforming to an XML schema can be considered as instances of the class defined by that XML schema. A common use of an XML schema is to validate that a given such **instance document** conforms to the rules defined by the XML schema.

XML Namespaces and XML Schemas

An XML schema can specify a targetNamespace attribute, whose value is a URL. If omitted, the schema has no target namespace. The target namespace is the namespace for everything defined in the XML schema. The targetNamespace value is typically a URL where the XML schema can be accessed.

An XML instance document must specify the namespace of the root element of the document (same as the target namespace of the XML schema that the instance conforms to) and the location (URL) of the XML schema that defines this root element. This information is specified by attribute xsi:schemalocation. When the XML schema has no target namespace, use attribute xsi:noNamespaceSchemalocation to specify the schema URL.

Overview of Editing XML Schemas

You can author and edit XML schemas anyway you like.

For example, you can use 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

Figure 17-1 shows a purchase-order XML schema being edited using XMLSpy. XMLSpy is a graphical XML tool from Altova Corporation that you can use to create and edit XML schemas and other XML documents. See Altova.com for details.¹

¹ XMLSpy also supports WebDAV and FTP protocols, so you can use it to directly access and edit content stored in Oracle XML DB Repository.



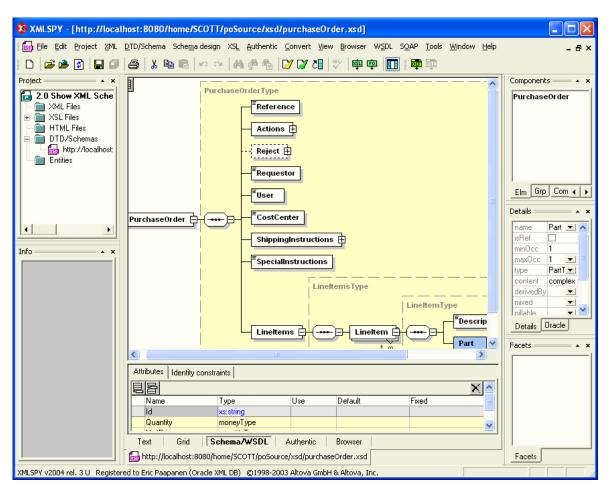


Figure 17-1 XMLSpy Graphical Representation of a Purchase-Order XML Schema

Overview of Using XML Schema with Oracle XML DB

Oracle XML DB supports registering XML schemas, validating documents against an XML schema, generating XML schemas from SQL object types, mapping from XML Schema to SQL, creating and querying XML Schema-based tables, views, and columns, and automatically inserting data when XML Schema-based documents are inserted into Oracle XML DB Repository.

XML schemas are stored in Oracle XML DB as XMLType instances, just like the XML documents that reference them. You must *register* an XML schema with Oracle XML DB in order to use it with XML data that is stored in the database.

Oracle XML DB takes advantage of the strong typing and other features of XML Schema to process XML database data safely and efficiently.

To be registered with Oracle XML DB, an XML schema must conform to the **root XML** Schema, XDBSchema.xsd. This is the XML schema for Oracle XML DB XML schemas. You can access XDBSchema.xsd at Oracle XML DB Repository location /sys/schemas/PUBLIC/ xmlns.oracle.com/xdb/XDBSchema.xsd.

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 respect the XML Schema standard.

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.
- 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 other than SQL.
- Why Use XML Schema with Oracle XML DB? Common reasons to use XML Schema include validation, constraint definition, storage specification, and optimization of document insertion, storage, and access.
- Overview of Annotating an XML Schema to Control Naming, Mapping, and Storage The W3C XML Schema Recommendation lets vendor-specific annotations be added to an XML schema. Oracle XML DB uses annotations to control the mapping between an XML schema and various database features. You can use annotations to specify which tables store XML data. Annotation is especially useful for object-relational storage.
- DOM Fidelity

DOM fidelity means that all information in an XML document is preserved except whitespace that is insignificant. You can use DOM fidelity to ensure the accuracy and integrity of XML documents stored in Oracle XML DB.

 XMLType Methods Related to XML Schema The most important XMLType methods for working with XML schemas are: isSchemaBased(), getSchemaURL(), schemaValidate(), isSchemaValid(), isSchemaValidated().

Why Use XML Schema with Oracle XML DB?

Common reasons to use XML Schema include validation, constraint definition, storage specification, and optimization of document insertion, storage, and access.

- The most common use of XML Schema is as a mechanism for validating that XML instance documents conform to a given XML schema, that is, verify that your XML data conforms to its intended definition. This definition includes data types, numbers of allowed item occurrences, and allowed lengths of items.
- An XML schema can also be used as a *constraint* when creating XMLType tables or columns. For example, the table or column can be constrained to store only XML documents that compliant with one of the global elements defined by the XML schema.



- 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. Both binary XML and objectrelational storage models for XMLType support the use of XML Schema. When XMLType data is stored object-relationally, XML Schema is used to efficiently map XML Schema data types to SQL data types and object-relational tables and columns.
- XML schema information can also improve the efficiency of document insertion when you storing XML Schema-based documents in Oracle XML DB using protocols FTP and HTTP(S).
- When XML instances must be handled without any prior information about them, XML schemas can be useful in predicting optimum storage, fidelity, and access.

Related Topics

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.

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

The W3C XML Schema Recommendation lets vendor-specific annotations be added to an XML schema. Oracle XML DB uses annotations to control the mapping between an XML schema and various database features. You can use annotations to specify which tables store XML data. Annotation is especially useful for object-relational storage.

You can use XML schema annotations with Oracle XML DB 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 SQL data types, for object-relational storage.
- Name the database objects and attributes that are created to store XML data (for objectrelational storage).

Example A-2 shows an annotated purchase-order XML schema. It defines the following two XML 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.

Before annotating an XML schema you *must* declare the Oracle XML DB namespace. The Oracle XML DB namespace is http://xmlns.oracle.com/xdb. Example A-2 makes use of the namespace *prefix* xdb to abbreviate the Oracle XML DB namespace.

Example A-2 uses several XML schema 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 annotation SQLType specifies that the name of the SQL data type generated from complexType element PurchaseOrderType is purchaseorder t.

The second occurrence of annotation SQLType specifies that the name of the SQL data type generated from complexType element LineItemType is lineitem_t.

- SQLCollType annotation. This specifies that the name of the SQL varray type that manages the collection of LineItem elements is lineitem v.
- SQLName annotation. This provides an explicit name for each SQL object attribute of purchaseorder_t.

Figure 17-2 shows the XMLSpy **Oracle** tab, which facilitates adding Oracle XML DB annotations to an XML schema while working in the graphical editor.

Figure 17-2 XMLSpy Support for Oracle XML DB Schema Annotations

	0/home/SCOTT/poSource/xsd/purchaseOrd	-	
	na Sche <u>m</u> aidesign XS <u>L</u> Authentic <u>C</u> onvert <u>V</u> iew		- & ×
D 🚅 🌦 🙆 🖬 💷 🍜 👗	🗈 🛍 🗠 🗠 🗛 🥠 🏝 🔤	× 👜 📫 🛄 👖 🚛	
Project ×		×	Components — 🔺 🗙
2.0 Show XML Sche	nt PurchaseOrder	ann:	PurchaseOrder
🗌 🛁 XML Files 🛛 📢 compl	exType PurchaseOrderType	ann:	
🕀 🏢 XSL Files 😽 📢 compl	exType LineItemsType	ann:	
HTML Files	exType LineItemType	ann:	
DTD/Schemas	exType PartType	ann:	
http://localhost:	Type ReferenceType	ann:	
efficies efficient	exType ActionsType	ann:	
-@compl	exType RejectionType	ann:	
-ti compl	exType ShippingInstructionsType	ann:	Elm Grp Com 4 🕨
simple	Type moneyType	ann:	
simple	Type quantityType	ann:	Details 🗕 🔺 🗙
simple	Type UserType	ann:	name Purchase 🔨
	Type RequestorType	ann:	type Purch
Info simple	Type CostCenterType	ann:	content complex
simple	Type VendorType	ann:	derivedBy 💌 🗏
simple	Type PurchaseOrderNumberType	ann:	mixed 💌
simple	Type SpecialInstructionsType	ann:	substGrp 📃
simple	Type NameType	ann:	abstract 🗾
simple	Type AddressType	ann:	nillable 👤 🚬
simple	Type TelephoneType	ann:	hlock V
simple		ann:	Details Oracle
simple	Type CommentsType	ann:	Franks
simple	Type DescriptionType	ann:	Facets × ×
Attribute	s Identity constraints		
		×	
		Default Fixed	
Text		Browser	
http:/	//localhost:8080/home/SCOTT/poSource/xsd/purchase	Order.xsd	Facets
KMLSPY v2004 rel. 3 U Registered to Eric P	aapanen (Oracle XML DB) ©1998-2003 Altova GmbH 8	k Altova, Inc.	

DOM Fidelity

DOM fidelity means that all information in an XML document is preserved except whitespace that is insignificant. You can use DOM fidelity to ensure the accuracy and integrity of XML documents stored in Oracle XML DB.

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.



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.

See Also: SYS_XDBPD\$ and DOM Fidelity for Object-Relational Storage for information about DOM fidelity and object-relational storage of XML data

XMLType Methods Related to XML Schema

The most important XMLType methods for working with XML schemas are: isSchemaBased(), getSchemaURL(), schemaValidate(), isSchemaValid(), isSchemaValidated(), and setSchemaValidated().

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.
<pre>schemaValidate() isSchemaValid() isSchemaValidated() setSchemaValidated()</pre>	Validation of an XMLType instance against a registered XML schema: validate, check validation status, or set recorded validation status. See Transformation and Validation of XMLType Data.

Table 17-1 XMLType Methods Related to XML Schema

XML Schema Registration with Oracle XML DB

Before an XML schema can be used by Oracle XML DB, you must register it. It can then be used to create XMLType tables and columns and to validate XML documents. If schema registration fails then the database is restored to the state it had prior to the registration attempt.

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 it had 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.

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. XML 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 the document location hint.

Note:

The act of registering an XML schema has *no effect* on the status of any instance documents that are *already loaded* into Oracle XML DB Repository and that reference that XML schema.

Such instance documents were treated as non XML-schema-based when they were loaded. They remain such. After schema registration, you must *delete* such documents and *reload* them, in order to obtain XML schema-based documents.

• XML Schema Registration Actions

As part of registering an XML schema, Oracle XML DB performs several actions that facilitate storing, accessing, and manipulating XML instances that conform to the XML schema.

Registering an XML Schema with Oracle XML DB

An example illustrates the use of PL/SQL procedure DBMS_XMLSCHEMA.registerSchema to register an XML schema. The main parameters of this procedure are the schema URL, the schema source document, the character-set ID of the source-document encoding, and options that specify how the schema should be registered.

 SQL Types and Tables Created During XML Schema Registration Registration of an XML schema results in the creation of object types and tables.

• Default Tables for Global Elements

By default, tables with system-generated names are created for all global elements. You can specify names to use instead. You can prevent the creation of default tables for particular elements, which can reduce processor time and space used, especially if an XML schema contains many global element definitions.

- Database Objects That Depend on Registered XML Schemas Several kinds of database object can depend on registered XML schemas: tables, views, other XML schemas, and cursors that reference an XML schema.
- Local and Global XML Schemas
 An XML schema can be registered as local (visible only to its owner, by default) or global
 (visible to all database users, by default).
- Fully Qualified XML Schema URLs Fully qualified XML schema URLs permit explicit reference to particular XML schemas. The name of the database user owning the XML schema is specified as part of the URL. Fully qualified schema URLs belong to the Oracle XML DB namespace.
- Deletion of an XML Schema You can delete a registered XML schema using procedure DBMS_XMLSCHEMA.deleteSchema.
- Listing All Registered XML Schemas
 An example lists all XML schemas that are registered with Oracle XML DB.



XML Schema Registration Actions

As part of registering an XML schema, Oracle XML DB performs several actions that facilitate storing, accessing, and manipulating XML instances that conform to the XML schema.

These 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 object-relational 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.

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.

Related Topics

• XML Schema Data Types Are Mapped to Oracle XML DB Storage

Data that conforms to an XML schema uses 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.

Default Tables Created during XML Schema Registration
You can create default tables as part of XML schema registration. Def

You can create default tables as part of XML schema registration. Default tables are most useful when documents are inserted using APIs and protocols such as FTP and HTTP(S), which do not provide any table specification.

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.

Registering an XML Schema with Oracle XML DB

An example illustrates the use of PL/SQL procedure DBMS_XMLSCHEMA.registerSchema to register an XML schema. The main parameters of this procedure are the schema URL, the schema source document, the character-set ID of the source-document encoding, and options that specify how the schema should be registered.

The main parameters to DBMS_XMLSCHEMA.registerSchema are as follows:

 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.
- The non-protocol part of the URL must be unique. The protocol part (for example, http or https) is ignored in the test for uniqueness.
- 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.

See Also:

Oracle Database PL/SQL Packages and Types Reference

Example 17-1 registers the annotated XML schema of Example A-2.

In Example A-2, 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.

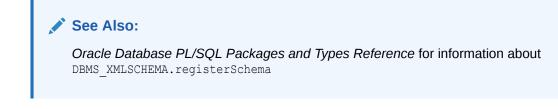
During XML schema registration, option SCHEMADOC specifies that PL/SQL constructor XDBURIType is to access the content of the XML schema document, based on its location in the repository. Other options passed to procedure registerSchema specify that the schema in Example A-2 is to be registered as a local XML schema (option LOCAL), and that SQL objects, and that tables are to be generated during the registration process (option GENTABLES).

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.

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.





Example 17-1 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;
/
```

Related Topics

- Local and Global XML Schemas
 An XML schema can be registered as local (visible only to its owner, by default) or global
 (visible to all database users, by default).
- SQL Types and Tables Created During XML Schema Registration Registration of an XML schema results in the creation of object types and tables.

SQL Types and Tables Created During XML Schema Registration

Registration of an XML schema results in the creation of object types and tables.

Example 17-2 shows the SQL type definitions that were created during an XML schema registration such as that of Example 17-1. 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.

Example 17-2 Objects Created During XML Schema Registration

DESCRIBE purchaseorder_t purchaseorder_t is NOT FINAL Name	Null?	Τγρε
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?	Туре
SYS_XDBPD\$ LINEITEM		XDB.XDB\$RAW_LIST_T LINEITEM_V
DESCRIBE lineitem v		
lineitem_v VARRAY(2147483647) OF LINEITEM LINEITEM_T is NOT FINAL	1_T	
Name	Null?	Туре
SYS_XDBPD\$ ITEMNUMBER DESCRIPTION PART		XDB.XDB\$RAW_LIST_T NUMBER(38) VARCHAR2(256 CHAR) PART_T

Default Tables for Global Elements

By default, tables with system-generated names are created for all global elements. You can specify names to use instead. You can prevent the creation of default tables for particular elements, which can reduce processor time and space used, especially if an XML schema contains many global element definitions.

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.

Database Objects That Depend on Registered XML Schemas

Several kinds of database object can depend on registered XML schemas: tables, views, other XML schemas, and cursors that reference an XML schema.

More precisely:



- 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.

Local and Global XML Schemas

An XML schema can be registered as local (visible only to its owner, by default) or global (visible to all database users, by default).

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 is local, meaning that it belongs to you alone, after you register it with Oracle XML DB. A reference to the XML schema document is stored in Oracle XML DB Repository under your user (database schema) name.

Global XML Schema

In contrast to local XML schemas, a privileged user can register an XML schema as global by specifying an argument to registration function DBMS_XMLSCHEMA. are visible to *all* users. They are stored under folder /sys/schemas/PUBLIC/ in Oracle XML DB Repository.

Local XML Schema

By default, an XML schema is local, meaning that it belongs to you alone, after you register it with Oracle XML DB. A reference to the XML schema document is stored in Oracle XML DB Repository under your user (database schema) name.

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.

For example, if the XML schema purchaseOrder.xsd is registered as a local schema 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 a given path name, in order to register the XML schema as a local XML schema. Some ways in which a local XML schema can be registered require one or more of the following privileges:

- ALTER SESSION
- CREATE PROCEDURE



- CREATE SESSION
- CREATE TABLE
- CREATE TRIGGER
- CREATE TYOE

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 (in this case, user QUINE).

Example 17-3 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;
/
```

Related Topics

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. You can create, set, and modify access control lists (ACLs).

Global XML Schema

In contrast to local XML schemas, a privileged user can register an XML schema as global by specifying an argument to registration function DBMS_XMLSCHEMA. 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 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 17-4 illustrates the registration of a global schema.

```
Example 17-4 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.

Fully Qualified XML Schema URLs

Fully qualified XML schema URLs permit explicit reference to particular XML schemas. The name of the database user owning the XML schema is specified as part of the URL. Fully qualified schema URLs belong to the Oracle XML DB namespace.

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.

The Oracle XML DB namespace is:

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 has a local registered XML schema with the same URL. Another user can reference the schema owned by QUINE has a local registered XML schema with the same URL. Another user can reference the schema owned by 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:

ORACLE

http://xmlns.oracle.com/xdb/schemas/PUBLIC/www.example.com/po.xsd

Related Topics

Local and Global XML Schemas
 An XML schema can be registered as local (visible only to its owner, by default) or global
 (visible to all database users, by default).

Deletion of an XML Schema

You can delete a registered XML schema using procedure DBMS XMLSCHEMA.deleteSchema.

This does the following, by default:

- 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).
- 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 is specified and there are instances in other tables that are also dependent on the XML schema.

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.
- 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 XML SCHEMAS.
- 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 XML SCHEMAS.

See Also: Oracle Database PL/SQL Packages and Types Reference

Example 17-5 illustrates the use of DELETE_CASCADE_FORCE.

If an XML schema was registered for use with binary XML, it is not removed from DBA_XML_SCHEMAS when you delete it using option DELETE_RESTRICT (the default value) or DELETE_CASCADE. Therefore, 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_XML_SCHEMAS 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_XML_SCHEMAS. 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.

Example 17-5 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;
/
```

Listing All Registered XML Schemas

An example lists all XML schemas that are registered with Oracle XML DB.

Example 17-6 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 17-6 Data Dictionary Table for Registered Schemas

DESCRIBE DBA_XML_SCHEMAS

Name	Null?	Туре
OWNER		VARCHAR2 (30)
SCHEMA_URL		VARCHAR2 (700)
LOCAL		VARCHAR2(3)
SCHEMA		<pre>XMLTYPE(XMLSchema "http://xmlns.oracle.com/xdb/XDBSchema.xsd"</pre>
		Element "schema")
INT_OBJNAME		VARCHAR2(4000)
QUAL_SCHEMA_	URL	VARCHAR2 (767)
HIER_TYPE		VARCHAR2 (11)
BINARY		VARCHAR2(3)



SCHEMA_ID HIDDEN	RAW(16) VARCHAR2(3)
SELECT OWNE	CR, LOCAL, SCHEMA_URL FROM DBA_XML_SCHEMAS;
OWNER LOC	C 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://www.w3.org/2001/xml.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?	Туре
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.

Creation of XMLType Tables and Columns Based on XML Schemas

You can create XMLType tables and columns that are constrained to a global element defined by an XML schema. After an XMLType column has been constrained to a particular element and a particular schema, it can only contain documents that are compliant with the schema definition of that element.

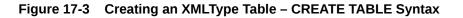
You constrain an XMLType table column to a particular element and XML schema by adding appropriate XMLSCHEMA and ELEMENT clauses to the CREATE TABLE operation.

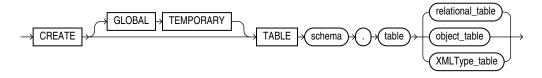
Figures Figure 17-3 through Figure 17-6 show the syntax for creating an XMLType table.



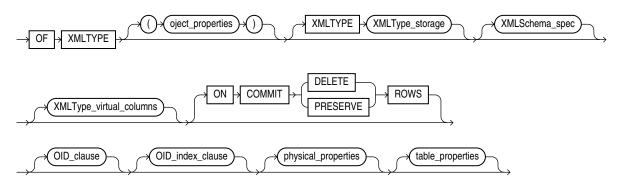
See Also: Oracle Database SQL Language Reference for the complete description of CREATE TABLE, including syntax elements such as object_properties. Note: To create an XMLType table in a different database schema from your own, you must have not only privilege CREATE ANY TABLE but also privilege CREATE 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.

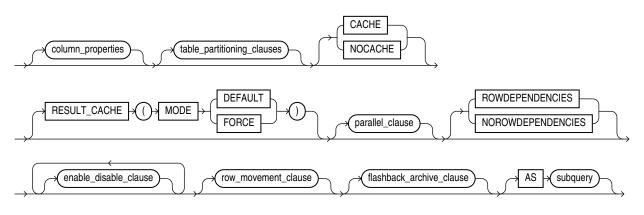




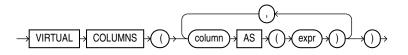












Note:

- Clause XMLType_virtual_columns can be used only for XMLType data that is stored as *binary XML*. In particular, if you use it for data that is stored object-relationally, and if you use a partitioning clause, then an error is raised.
- 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.

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 Query and Update of XML Data.

Example 17-7 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.

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
- Stored as binary XML, using a single binary-XML column

Example 17-7 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";
```

Specification of 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.



Related Topics

- Creating Virtual Columns on XMLType Data Stored as Binary XML You can create virtual columns only for XMLType data that is stored as binary XML. Such columns are useful for partitioning or constraining the data.
- 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 insert this data into a pre-existing database table, or you can query it using SQL — in a join expression, for example.

 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.

Specification of 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.

Besides 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.

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 documents must conform to then you can use that schema only to create XMLType tables and columns that are stored as binary XML.

Object-Relational Storage of XML Schema-Based Data

After you register an XML schema you can create an object-relational XMLType table or column for documents that conform to that schema. You can optionally specify object-relational storage options for the table or column.

Related Topics

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.

Choice of XMLType Storage and Indexing

Important design choices for your application include what XMLType storage model to use and which indexing approaches to use.

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 documents must conform to then you can use that 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.



The converse is also true: If you use object-relational storage for the registered XML schema, then you can use only that XML schema to create XMLType tables and columns that are stored as binary XML.

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 nonschema-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 Copy-Based Schema Evolution). 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, 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 17-2.

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.

Object-Relational Storage of XML Schema-Based Data

After you register an XML schema you can create an object-relational XMLType table or column for documents that conform to that schema. You can optionally specify object-relational storage options for the table or column.

Suppose that you have registered a purchase-order XML schema, identified by URL http:// xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd. You then create an objectrelational XMLType table, purchaseorder_as_table, to store instances that conform to element PurchaseOrder of the XML schema, as in Example 17-8.

This automatically creates hidden columns that correspond to the database 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.

Suppose that XML schema purchaseOrder.xsd defines element LineItems as a child of element PurchaseOrder, and that LineItems is a collection of LineItem elements.

With object-relational storage, collections are mapped to SQL varray values. An XML **collection** is any element that is defined by the XML schema with 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 17-9 creates table purchaseorder_as_table differently from Example 17-8. It specifies additional storage options:

- The LineItems collection varray is stored as a LOB, not as a table.
- Tablespace USERS is used for storing element Notes.
- The table is compressed for online transaction processing (OLTP).

Note:

In releases prior to Oracle Database 11gR2, 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 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 17-10 illustrates this.

See Also:

Oracle Database SQL Language Reference for information about using STORE ALL VARRAYS AS LOBS

Example 17-8 Creating an Object-Relational XMLType Table with Default Storage

```
CREATE TABLE purchaseorder_as_table OF XMLType
    ELEMENT
    "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd#PurchaseOrder";
```

Example 17-9 Specifying Object-Relational 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;
```



Example 17-10 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;
```

Ways to Identify XML Schema Instance Documents

Before an XML document can be inserted into an XML Schema-based XMLType table or column, the associated XML schema must be identified. You can do this when you create the table or column, or you can use XMLSchema-instance to explicitly add the required schema identification to the XML instance document.

To explicitly identify the XML schema when creating the XMLType table or column, you can pass the name of the XML schema to the XMLType constructor, or you can invoke XMLType method createSchemaBasedXML().

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 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, attribute noNamespaceSchemaLocation is used to identify the XML schema. The attribute value is the schema location hint. This is the unique identifier that is passed to PL/SQL procedure DBMS_XMLSCHEMA.registerSchema when the XML schema is registered with the database.

XML Schema and Multiple Namespaces
 When an 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.



Attributes noNamespaceSchemaLocation and schemaLocation

If the target XML schema does not declare a target namespace, attribute noNamespaceSchemaLocation is used to identify the XML schema. The attribute value is the schema location hint. This is the unique identifier that is 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">
    <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">
```

XML Schema and Multiple Namespaces

When an 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 XML instance document also needs to include a noNamespaceSchemaLocation attribute that provides the schema location hint for the primary XML schema.

XML Schema Data Types Are Mapped to Oracle XML DB Storage

Data that conforms to an XML schema uses 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 object-relational 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.

Figure 17-7 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 value in a single XMLType column, or using multiple object-relational columns.



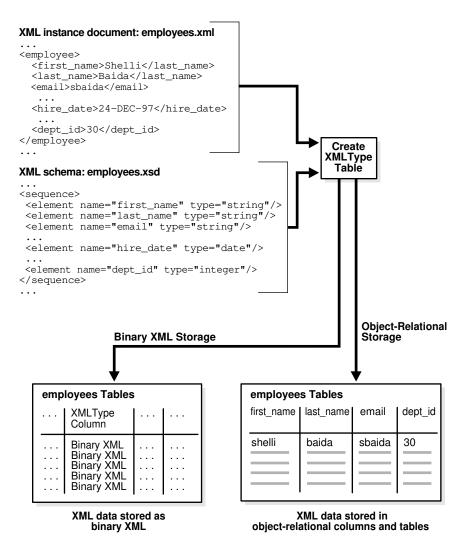


Figure 17-7 How Oracle XML DB Maps XML Schema-Based XMLType Tables

Related Topics

 Use DBMS_XMLSCHEMA to Map XML Schema Data Types to SQL Data Types You use PL/SQL package DBMS_XMLSCHEMA to map data types for XML Schema attributes and elements to SQL data types.

18

XML Schema Storage and Query: Object-Relational Storage

Advanced techniques for XML Schema-based data include using object-relational storage; annotating XML schemas; mapping Schema data types to SQL; using complexType extensions and restrictions; creating, specifying relational constraints on, and partitioning XML Schema-based data, storing XMLType data out of line, working with complex or large schemas, and debugging schema registration.

Object-Relational Storage of XML Documents

Object-relational storage of XML documents is based on decomposing the document content 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 schema.

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.

- Use DBMS_XMLSCHEMA to Map XML Schema Data Types to SQL Data Types You use PL/SQL package DBMS_XMLSCHEMA to map data types for XML Schema attributes and elements to SQL data types.
- complexType Extensions and Restrictions in Oracle XML DB In XML Schema, complexType values are declared based on complexContent and simpleContent. Oracle XML DB defines various extensions and restrictions to complexType.
- Creating XML Schema-Based XMLType Columns and Tables After an XML schema has been registered with Oracle XML DB, you can reference it when you define XMLType tables or columns.
- Overview of Partitioning XMLType Tables and Columns Stored Object-Relationally When you partition an object-relational XMLType table or a table with an XMLType column that is stored object-relationally and you use list, range, or hash partitioning, any ordered collection tables (OCTs) or out-of-line tables within the data are automatically partitioned accordingly, by default.
- Specification of Relational Constraints on XMLType Tables and Columns For XMLType data stored object-relationally, you can specify typical relational constraints for elements and attributes that occur only once in an XML document.
- Out-Of-Line Storage of XMLType Data By default, when XMLType data is stored object-relationally a child element is mapped to an embedded SQL object attribute. Sometimes better performance can be obtained by storing some XMLType data out of line. Use XML schema annotation xdb:SQLInline to do this.
- Considerations for Working with Complex or Large XML Schemas XML schemas can be complex. Examples of complex schemas include those that are recursive and those that contain circular or cyclical references. Working with complex or



large XML schemas can be challenging and requires taking certain considerations into account.

 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 event 31098 before invoking PL/SQL procedure DBMS XMLSCHEMA.registerSchema.

See Also:

- XML Schema Storage and Query: Basic for basic information about using XML Schema with Oracle XML DB
- XPath Rewrite for Object-Relational Storage for information about the optimization of XPath expressions in Oracle XML DB
- XML Schema Evolution for information about updating an XML schema after you have registered it with Oracle XML DB
- XML Schema Part 0: Primer Second Edition for an introduction to XML Schema

Object-Relational Storage of XML Documents

Object-relational storage of XML documents is based on decomposing the document content 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 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 that is compliant with the XML schema to be decomposed and stored in the database as a set of objects, without any loss of information. When an XML 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 it 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.

How Collections Are Stored for Object-Relational XMLType Storage

You can store an ordered collection as a varray in an ordered collection table (OCT), which can be a heap-based table. You can store the actual data out of line by using varray entries that are REFs to the data.

• SQL Types Created during XML Schema Registration for Object-Relational Storage Use TRUE as the value of parameter GENTYPES when you register an XML schema for use with XML data stored object-relationally (TRUE is the default value). Oracle XML DB then creates the appropriate SQL object types that enable object-relational storage of conforming XML documents.

- Default Tables Created during XML Schema Registration
 You can create default tables as part of XML schema registration. Default tables are most useful when documents are inserted using APIs and protocols such as FTP and HTTP(S), which do not provide any table specification.
- Do Not Use Internal Constructs Generated during XML Schema Registration
 In general, the SQL constructs (data types, nested tables, and tables associated with outof-line storage) that are automatically generated during XML schema registration are internal to Oracle XML DB. Oracle recommends that you do not use them in your code.
- Generated Names are Case Sensitive The names of any SQL tables, objects, and attributes generated by XML schema registration are *case sensitive*.
- SYS_XDBPD\$ and DOM Fidelity for Object-Relational Storage
 In order to provide DOM fidelity for XML data that is stored object-relationally, Oracle
 XML DB records all information that cannot be stored in any of the other object attributes
 as instance-level metadata using the system-defined binary object attribute SYS_XDBPD\$
 (positional descriptor, or PD).

How Collections Are Stored for Object-Relational XMLType Storage

You can store an ordered collection as a varray in an ordered collection table (OCT), which can be a heap-based table. You can store the actual data out of line by using varray entries that are REFs to the data.

When you register an XML schema for XMLType 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**.

You can store data in an ordered collection as a **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**). Oracle XML DB stores a collection as a *heap-based* OCT.

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 REFS in the collection table (or the 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.

Related Topics

Setting Annotation Attribute xdb:SQLInline to false for Out-Of-Line Storage

Set XML schema annotation xdb:SQLInline to false to store an XML fragment out of line. The element is mapped to a SQL object type with an embedded REF attribute, which points to another XMLType instance that is stored out of line and that corresponds to the XML fragment.

Overview of Partitioning XMLType Tables and Columns Stored Object-Relationally

When you partition an object-relational XMLType table or a table with an XMLType column that is stored object-relationally and you use list, range, or hash partitioning, any ordered collection tables (OCTs) or out-of-line tables within the data are automatically partitioned accordingly, by default.



See Also:

Object-Relational Storage of XML Documents for information about collection storage when you create XMLType tables and columns manually using object-relational storage

SQL Types Created during XML Schema Registration for Object-Relational Storage

Use TRUE as the value of parameter GENTYPES when you register an XML schema for use with XML data stored object-relationally (TRUE is the default value). Oracle XML DB then creates the appropriate SQL object types that enable object-relational 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 18-1 shows the SQL object types that are created automatically when XML schema purchaseOrder.xsd is registered with Oracle XML DB.

Note:

By default, the names of the SQL object types and attributes are system-generated. This is the case in Example 18-1. 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).

Note:

Starting with Oracle Database 12c Release 2 (12.2.0.1), if you register an XML schema for object-relational storage for an *application common user* then you *must* annotate each complex type in the schema with xdb:SQLType, to name the SQL data type. Otherwise, an error is raised.

Example 18-1 SQL Object Types for Storing XMLType Tables



```
VARCHAR2 (10 CHAR)
User
CostCenterVARCHAR2 (4 CHAR)ShippingInstructionsShippingInstructionsTyp1659_TSpecialInstructionsVARCHAR2 (2048 CHAR)LineItemsLineItemsType1666_T
Notes
                          VARCHAR2 (4000 CHAR)
DESCRIBE "LineItemsType1666 T"
"LineItemsType1666_T" is NOT FINAL
      Null? Type
Name
_____
                            -----
SYS_XDBPD$ XDB.XDB$RAW_LIST_T
LineItem
                         LineItem1667 COLL
DESCRIBE "LineItem1667 COLL"
"LineItem1667 COLL" VARRAY(2147483647) OF LineItemType1665_T
"LineItemType1665 T" is NOT FINAL
      Null? Type
Name
----- ---- -----
                                  _____
SYS_XDBPD$XDB.XDB$RAW_LIST_TItemNumberNUMBER(38)DescriptionVARCHAR2(256 CHAR)
ItemNumber
Description
                        PartType1664 T
Part
```

Default Tables Created during XML Schema Registration

DESCRIBE "PurchaseOrder1669 TAB"

You can create default tables as part of XML schema registration. Default tables are most useful when documents are inserted using APIs and protocols such as FTP and HTTP(S), which do not provide any table specification.

In such cases, the XML instance is inserted into the default table.

Example 18-2 describes the default purchase-order table.

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.

Example 18-2 Default Table for Global Element PurchaseOrder

```
Name Null? Type

TABLE of

SYS.XMLTYPE(

XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"

Element "PurchaseOrder")

STORAGE OBJECT-RELATIONAL TYPE "PurchaseOrderType1668_T"
```



Do Not Use Internal Constructs Generated during XML Schema Registration

In general, the SQL constructs (data types, nested tables, and tables associated with out-ofline storage) that are automatically generated during XML schema registration are *internal* to Oracle XML DB. Oracle recommends that you do *not* use them in your code.

More precisely, generated SQL data types, nested tables, and tables associated with out-ofline storage are based on specific internal XML schema-to-object type mappings that are subject to change and redefinition by Oracle at any time. In general:

- Do not use any generated SQL data types.
- Do not access or modify any generated nested tables or out-of-line tables.

You can, however, modify the storage options, such as partitioning, of generated tables, and you can create indexes and constraints on generated tables. You can also freely use any XML schema annotations provided by Oracle XML DB, including annotations that name generated constructs.

Generated Names are Case Sensitive

The names of any SQL tables, objects, and attributes generated by XML schema registration are *case sensitive*.

For instance, in Example 18-2, the name of table PurchaseOrder1669_TAB is derived from the name of element PurchaseOrder, so it too is mixed case. You must therefore refer to this table 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.

SYS_XDBPD\$ and DOM Fidelity for Object-Relational Storage

In order to provide DOM fidelity for XML data that is stored object-relationally, Oracle XML DB records all information that cannot be stored in any of the other object attributes as instance-level metadata using the system-defined binary object attribute SYS_XDBPD\$ (positional descriptor, or PD).

With object-relational storage of XML data, the elements and attributes declared in an XML schema are mapped to separate attributes of 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

In order to provide DOM fidelity for XML data stored object-relationally, Oracle XML DB uses a separate mechanism to keep track of this information: it is recorded as 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.

Related Topics

- You Can Override the SQLType Value in an XML Schema When Declaring Attributes You can explicitly specify a SQLType value in an XML schema, as an annotation. The SQL data type that you specify is used for XML schema validation, overriding the default SQL data types.
- Override of 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 its sub-elements and attributes. You can override this mapping by explicitly specifying a value for attribute SQLType in the input XML schema.

See Also:

DOM Fidelity for information about DOM fidelity and binary XML storage of XML data

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.

You can add such annotations manually by editing the XML schema document or, for the most common annotations, by invoking annotation-specific PL/SQL subprograms. See *Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLSCHEMA_ANNOTATE".

If you edit an XML schema manually using the Altova XMLSpy editor then you can take advantage of the *Oracle* tab in the editor for adding and editing Oracle-specific annotations. See Figure 17-2.



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
 You can annotate an XML schema to customize the names of object-relational tables,
 objects, and object attributes or to allow XPath rewrite when XQuery-expression
 arguments target recursive XML data.
- XML Schema Annotation Example A sample XML schema illustrates some of the most important Oracle XML DB annotations.
- Annotating an XML Schema Using DBMS_XMLSCHEMA_ANNOTATE
 PL/SQL package DBMS_XMLSCHEMA_ANNOTATE provides subprograms to annotate an XML schema. Using these subprograms can often be more convenient and less error prone than manually editing the XML schema.
- Available Oracle XML DB XML Schema Annotations The Oracle XML DB annotations that you can specify in element and attribute declarations are described, along with the RL (SQL subgrages in package DBMs, VML SQUEMA, ANNOTATE

are described, along with the PL/SQL subprograms in package DBMS_XMLSCHEMA_ANNOTATE that you can use to manipulate them.

• XML Schema Annotation Guidelines for Object-Relational Storage

For XMLType data stored object-relationally, careful planning is called for, to optimize performance. Similar considerations are in order as for relational data: entity-relationship models, 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.

Querying a Registered XML Schema to Obtain Annotations

You can query database views USER_XML_SCHEMAS and ALL_XML_SCHEMAS to obtain a registered XML schema with all of its annotations. The registered version of an XML schema contains a full set of Oracle XML DB annotations. These annotations were supplied by a user or set by default during XML schema registration.

Common Uses of XML Schema Annotations

•

You can annotate an XML schema to customize the names of object-relational tables, objects, and object attributes or to allow XPath rewrite when XQuery-expression arguments target recursive XML data.

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 object-relational storage of XMLType data are easy to recognize and compliant with any application-naming standards. Set parameter GENTYPES or GENTABLES to TRUE for this (TRUE is the default value for each of these parameters).
- 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 object-relational 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 TRUE. 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, NVARCHAR2, or NCLOB as the value of a SQLType annotation.

Note:

Starting with Oracle Database 12c Release 2 (12.2.0.1), if you register an XML schema for object-relational storage for an *application common user* then you *must* annotate each complex type in the schema with xdb:SQLType, to name the SQL data type. Otherwise, an error is raised.

- 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

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 object-relational 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

A sample XML schema illustrates some of the most important Oracle XML DB annotations.

The XML schema in Example 18-3 is similar to the one in Example A-2, but it also defines a Notes element and its type, NotesType.

- The schema element includes the declaration of the xdb namespace.
- 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 <code>PurchaseOrderType</code> includes a <code>SQLType</code> annotation that specifies that the generated SQL object type is named <code>purchaseorder_t</code>. 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 annotation is 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 18-4 shows some of the tables and objects that are created when the annotated XML schema of Example 18-3 is registered.

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.

Example 18-3 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">
  <xs:element name="PurchaseOrder" type="po:PurchaseOrderType"</pre>
              xdb:defaultTable="PURCHASEORDER"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER T">
    <xs:sequence>
      <xs:element name="Reference" type="po:ReferenceType" minOccurs="1"</pre>
                  xdb:SQLName="REFERENCE"/>
      <xs:element name="Actions" type="po:ActionsType"</pre>
                  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"</pre>
```



```
xdb:SQLName="EMAIL"/>
      <xs:element name="CostCenter" type="po:CostCenterType"/>
      <xs:element name="ShippingInstructions"</pre>
                  type="po:ShippingInstructionsType"/>
      <xs:element name="SpecialInstructions" type="po:SpecialInstructionsType"/>
      <xs:element name="LineItems" type="po:LineItemsType"</pre>
                  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"</pre>
                  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:simpleType name="NotesType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <rs:maxLength value="32767"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Example 18-4 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 tation/purchaseOrder.xsd	PurchaseOrder

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 N		
Name	Null?	Туре
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 F Name	INAL Null?	
SYS XDBPD\$		XDB.XDB\$RAW_LIST_T
LINEITEM VARRAY		LINEITEM V
_ DESCRIBE lineitem_v		_
LINEITEM_V VARRAY(21	474836	47) OF LINEITEM_T
LINEITEM_T is NOT FI	NAL	
Name	Null?	Туре
SYS XDBPD\$		XDB.XDB\$RAW LIST T
SYS_XDBPD\$ ITEMNUMBER		XDB.XDB\$RAW_LIST_T NUMBER(38)

PART_T

DESCRIBE part_t

PART

PART T is NOT FINAL

2 rows selected.

Annotating an XML Schema Using DBMS_XMLSCHEMA_ANNOTATE

PL/SQL package DBMS_XMLSCHEMA_ANNOTATE provides subprograms to annotate an XML schema. Using these subprograms can often be more convenient and less error prone than manually editing the XML schema.

In particular, you can use the PL/SQL subprograms in a script, which you can run at any time or multiple times, as needed. This can be especially useful if you are using a large XML schema or a standard or other third-party XML schema that you do not want to modify manually.

There are specific PL/SQL subprograms for each Oracle annotation. For example, you use PL/SQL procedure setDefaultTable to add a xdb:defaultTable annotation, and removeDefaultTable to remove a xdb:defaultTable annotation.

Each annotation subprogram has the following as its parameters:

- The XML schema to be annotated. This parameter is IN OUT.
- The name of the global element where the annotation is to be added or removed.
- The annotation (XML attribute) value.
- A Boolean flag indicating whether any corresponding existing annotation is to be overwritten. By default, it is overwritten.

If the element to be annotated is not a global element then you provide the local element name as an additional parameter. The global and local names together identify the target element. The element with the local name must be a descendent of the element with the global name.

If you use SQL*Plus, you can use PL/SQL procedure DBMS_XMLSCHEMA_ANNOTATE.printWarnings to enable and disable printing of SQL*Plus warnings during the use of other DBMS_XMLSCHEMA_ANNOTATE subprograms. By default, no warnings are printed. An example of a warning is an inability to annotate the XML schema because there is no element with the name you provided to the annotation subprogram.

Example 18-5 uses subprograms in PL/SQL package DBMS_XMLSCHEMA_ANNOTATE to produce the annotated XML schema shown in Example 18-3.



See Also:

Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XMLSCHEMA_ANNOTATE"

Example 18-5 Using DBMS_XMLSCHEMA_ANNOTATE

```
CREATE TABLE annotation tab (id NUMBER, inp XMLType, out XMLType);
INSERT INTO annotation tab VALUES (1, ... unannotated XML schema...);
DECLARE
 schema XMLType;
BEGIN
 SELECT t.inp INTO schema FROM annotation tab t WHERE t.id = 1;
  DBMS XMLSCHEMA ANNOTATE.setDefaultTable(schema, 'PurchaseOrder', 'PURCHASEORDER');
  DBMS XMLSCHEMA ANNOTATE.setSQLType(schema, 'PurchaseOrderType', 'PURCHASEORDER_T');
  DBMS XMLSCHEMA ANNOTATE.setSQLName (schema, 'complexType', 'PurchaseOrderType', 'element', 'Reference',
                                     'REFERENCE');
  DBMS XMLSCHEMA ANNOTATE.setSQLName(schema, 'complexType', 'PurchaseOrderType', 'element', 'Actions',
                                     'ACTIONS COLLECTION');
  DBMS_XMLSCHEMA_ANNOTATE.setSQLName(schema, 'complexType', 'PurchaseOrderType', 'element', 'User', 'EMAIL');
  DBMS XMLSCHEMA ANNOTATE.setSQLName(schema, 'complexType', 'PurchaseOrderType', 'element', 'LineItems',
                                     'LINEITEM_COLLECTION');
  DBMS_XMLSCHEMA_ANNOTATE.setSQLType(schema, 'complexType', 'PurchaseOrderType', 'element', 'Notes', 'CLOB');
  DBMS XMLSCHEMA ANNOTATE.setSQLType (schema, 'LineItemsType', 'LINEITEMS T');
  DBMS_XMLSCHEMA_ANNOTATE.setSQLCollType(schema, 'complexType', 'LineItemsType', 'LineItem', 'LINEITEM V');
  DBMS XMLSCHEMA ANNOTATE.setSQLName(schema, 'complexType', 'LineItemsType', 'element', 'LineItem',
                                     'LINEITEM VARRAY');
  DBMS_XMLSCHEMA_ANNOTATE.setSQLType(schema, 'LineItemType', 'LINEITEM_T');
  DBMS XMLSCHEMA ANNOTATE.setSQLType(schema, 'PartType', 'PART T');
  DBMS XMLSCHEMA ANNOTATE.disableMaintainDom(schema, 'PartType');
UPDATE annotation tab t SET t.out = schema WHERE t.id = 1;
```

```
END;
/
```

Available Oracle XML DB XML Schema Annotations

The Oracle XML DB annotations that you can specify in element and attribute declarations are described, along with the PL/SQL subprograms in package DBMS_XMLSCHEMA_ANNOTATE that you can use to manipulate them.

All annotations except those that have the prefix csx are applicable to XML schemas registered for object-relational storage.

The following annotations apply to XML schemas that are registered for binary XML storage:

- xdb:defaultTable
- xdb:tableProps

See Also:

Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XMLSCHEMA_ANNOTATE"



Table 18-1 Annotations in Elements

Attribute and PL/SQL	Values	Default	Description
xdb:columnProps	storage	NULL	Specifies the COLUMN storage clause that is inserted into the default CREATE TABLE statement. It is useful
No applicable PL/SQL.	clause		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
PL/SQL:			useful in cases where the XML data is inserted from APIs and protocols, such as FTP and HTTP(S), where
setDefaultTable removeDefaultTable enableDefaultTableCreation disableDefaultTableCreation			the table name is not specified. Applicable to object- relational storage and 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
PL/SQL:			all comments, processing instructions, namespace declarations, and so on are retained, in addition to the ordering of elements.
disableMaintainDOM			If false, then the output is not guaranteed to have the same DOM action as the input.
xdb:SQLCollType	Any SQL collection	-	Name of the SQL collection type that corresponds to this XML element. The XML element must be specified
PL/SQL:	type	element name	with maxOccurs > 1.
setSQLCollType removeSQLCollType			
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).
PL/SQL:			If false, then a REF value is stored (or a collection of
setOutOfLine removeOutOfLine			REF values, if maxOccurs > 1). This attribute is forced 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 this XML element.
PL/SQL:			
setSQLName removeSQLName			
xdb:SQLType	Any SQL data type ¹ ,	-	Name of the SQL type corresponding to this XML element declaration.
	except	element name	
PL/SQL:	NCHAR,		



Table 18-1	(Cont.)	Annotations	in	Elements
------------	---------	-------------	----	----------

Attribute and PL/SQL	Values	Default	Description
xdb:tableProps	Any table storage	NULL	Specifies the TABLE storage clause that is appended to the default CREATE TABLE statement. This is
PL/SQL:	clause		meaningful mainly for global and out-of-line elements. Applicable to object-relational storage and binary XML
setTableProps removeTableProps			storage.

¹ See Use DBMS_XMLSCHEMA to Map XML Schema Data Types to SQL Data Types.



Table 18-2 Annotations in Elements Declaring Global complexType Elements

Attribute	Values	Default	Description
xdb:maintainDOM	true false	true	If true, then instances of this element are stored so that they retain DOM fidelity on
PL/SQL:			output. This implies that all comments, processing instructions, namespace
enableMaintainDom disableMaintainDom			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:SQLType	Any SQL data type ¹ <i>except</i> NCHAR,	Name generated from element name	Name of the SQL type that corresponds to this XML element declaration.
PL/SQL:	NVARCHAR2, and NCLOB		
setSQLType removeSQLType			

¹ See Use DBMS_XMLSCHEMA to Map XML Schema Data Types to SQL Data Types.

XML Schema Annotation Guidelines for Object-Relational Storage

For XMLType data stored object-relationally, careful planning is called for, to optimize performance. Similar considerations are in order as for relational data: entity-relationship models, 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.



- Avoid Creation of Unnecessary Tables for Unused Top-Level Elements
 - 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 associated 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 your queries. This 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 instead, especially for tables that you are sure to be interested in.

Turn Off DOM Fidelity If Not Needed

By default, XML schema registration generates tables that maintain DOM fidelity. 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.

• 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 object-relational XMLType data needs additional properties specified, such as partition, tablespace, or compression, use annotation xdb:tableProps or xdb:columnProps. You can do this to add primary keys or constraints, for example.

• Store Large Collections Out of Line

If you have large collections then you might need to use annotations xdb:defaultTable and xdb:SQLInline to specify that collection elements be stored out of line.

Related Topics

• XPath Rewrite for Object-Relational Storage

For XMLType data stored object-relationally, queries involving XPath expression arguments to various SQL functions can often be automatically rewritten to queries against the underlying SQL tables, which are highly optimized.

See Also:

Table 18-1

Avoid Creation of Unnecessary Tables for Unused Top-Level Elements

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 associated tables by adding annotation xdb:defaultTable = "" to the element in the XML schema. An empty value for this attribute prevents default-table creation.

By default, XML schema registration creates a top-level table for each top-level element defined in the schema. Some such elements might be used at top level in XML instances that conform to the schema. For example, elements in an XML schema might be top-level in order to be used as a REF target.



You can use PL/SQL procedure DBMS_XMLSCHEMA_ANNOTATE.disableDefaultTableCreation to add an empty xdb:defaultTable attribute to each top-level element that has no xdb:defaultTable attribute.

Note:

Any top-level XML schema element that is used as the root element of any instance documents must have a non-empty xdb:defaultTable attribute.

See Also:

Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XMLSCHEMA_ANNOTATE" for information about PL/SQL procedure disableDefaultTableCreation.

Provide Your Own Names for Default Tables

For tuning purposes, you examine execution plan output for your queries. This 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 instead, especially for tables that you are sure to be interested in.

You do that using annotation xdb:defaultTable.

Related Topics

Default Tables Created during XML Schema Registration

You can create default tables as part of XML schema registration. Default tables are most useful when documents are inserted using APIs and protocols such as FTP and HTTP(S), which do not provide any table specification.

Turn Off DOM Fidelity If Not Needed

By default, XML schema registration generates tables that maintain DOM fidelity. 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.

Related Topics

DOM Fidelity

DOM fidelity means that all information in an XML document is preserved except whitespace that is insignificant. You can use DOM fidelity to ensure the accuracy and integrity of XML documents stored in Oracle XML DB.

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 object-relational XMLType data needs additional properties specified, such as partition, tablespace, or compression, use annotation xdb:tableProps or xdb:columnProps. You can do this to add primary keys or constraints, for example.

For example, to achieve table compression for online transaction processing (OLTP), you would add COMPRESS FOR OLTP using a tableProps attribute.

See Also:

Example 17-9 for an example of specifying Advanced Row Compression when creating XMLType tables and columns manually

Store Large Collections Out of Line

If you have large collections then you might need to use annotations xdb:defaultTable and xdb:SQLInline to specify that collection elements be stored out of line.

The maximum number of elements and attributes defined by a complexType is 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. If you have large collections, then you might run up against this limit of 4096 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.

Note:

For each inheritance hierarchy or substitution group in an XML schema, a table is created whose columns cover the content models of that hierarchy or substitution group. This too can cause the 4096-column limit to be reached.



Related Topics

ORA-01792 and ORA-04031: Issues with Large XML Schemas

Errors ORA-01792 and ORA-04031 can be raised when you work with large or complex XML schemas. You can encounter them when you register an XML schema or you create a table that is based on a global element defined by an XML schema.

Setting Annotation Attribute xdb:SQLInline to false for Out-Of-Line Storage

Set XML schema annotation xdb:SQLInline to false to store an XML fragment out of line. The element is mapped to a SQL object type with an embedded REF attribute, which points to another XMLType instance that is stored out of line and that corresponds to the XML fragment.

Querying a Registered XML Schema to Obtain Annotations

You can query database views USER_XML_SCHEMAS and ALL_XML_SCHEMAS to obtain a registered XML schema with all of its annotations. The registered version of an XML schema contains a full set of Oracle XML DB annotations. These annotations were supplied by a user or set by default during XML schema registration.

Example 18-6 illustrates this. It returns the XML schema as an XMLType instance.

As shown in Example 17-3 and Example 17-4, the location of the registered XML schema depends on whether it is local or global. If you want to project specific annotation information to relational columns, you can query RESOURCE_VIEW. Example 18-7 illustrates this. It obtains the set of global complexType definitions declared by an XML schema for object-relational storage of XMLType data, and the corresponding SQL object types and DOM fidelity values.

Example 18-6 Querying View USER_XML_SCHEMAS for a Registered XML Schema

```
SELECT SCHEMA FROM USER_XML_SCHEMAS
WHERE SCHEMA URL = 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd';
```

Example 18-7 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;
                          SQL_TYPE NAME
                                                     DOM FIDELITY
XMLSCHEMA TYPE NAME
```



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.

 You Can Apply Annotations from One XML Schema to Another Sometimes you need to apply the annotations from one XML schema to another XML schema. A typical use case is applying the annotations from an older version of a schema to a new version. You can get and set annotations using PL/SQL subprograms getSchemaAnnotations and setSchemaAnnotations, respectively.

You Can Apply Annotations from One XML Schema to Another

Sometimes you need to apply the annotations from one XML schema to another XML schema. A typical use case is applying the annotations from an older version of a schema to a new version. You can get and set annotations using PL/SQL subprograms getSchemaAnnotations and setSchemaAnnotations, respectively.

PL/SQL function getSchemaAnnotations returns all of the annotations from an XML schema. PL/SQL procedure setSchemaAnnotations sets annotations. These subprograms are in PL/SQL package DBMS XMLSCHEMA ANNOTATE.

See Also:

Oracle Database PL/SQL Packages and Types Reference, chapter "DBMS_XMLSCHEMA_ANNOTATE" for information about PL/SQL subprograms getSchemaAnnotations and setSchemaAnnotations.

Use DBMS_XMLSCHEMA to Map XML Schema Data Types to SQL Data Types

You 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 An example illustrates mapping XML Schema data types to SQL data types.

 XML Schema Attribute Data Types Mapped to SQL An XML attribute declaration can specify its XML Schema data type in terms of a primitive type, a local simpleType, a global simpleType, or a reference to a global attribute (ref=".."). The SQL data type and its associated information are derived from the base XML Schema type.

• XML Schema Element Data Types Mapped to SQL

An XML element declaration can specify its XML Schema data type using a primitive type, a local or global simpleType, a local or global complexType, or a reference to a global element (ref=".."). The SQL data type and its associated information are derived from the base XML Schema type.

- How XML Schema simpleType Is Mapped to SQL
 XML simpleType is mapped to SQL object types in various ways, depending on how the simpleType is defined.
- How XML Schema complexType Is Mapped to SQL
 XML complexType is mapped to SQL object types in various ways, depending on how the complexType is defined.

Example of Mapping XML Schema Data Types to SQL

An example illustrates mapping XML Schema data types to SQL data types.

Example 18-8 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 18-8 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">
 <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"</pre>
                 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"</pre>
               xdb:SQLName="DESCRIPTION"/>
    <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"</pre>
                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:simpleTvpe>
      <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>
```

XML Schema Attribute Data Types Mapped to SQL

An XML attribute declaration can specify its XML Schema data type in terms of a primitive type, a local simpleType, a global simpleType, or a reference to a global attribute (ref="..").

The SQL data type and its associated information are derived from the base XML Schema type.

An attribute declaration can specify its XML Schema data type in terms of any 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, any associated information (length, precision), and the memory mapping information are derived from the simpleType on which the attribute is based.

 You Can Override the SQLType Value in an XML Schema When Declaring Attributes You can explicitly specify a SQLType value in an XML schema, as an annotation. The SQL data type that you specify is used for XML schema validation, overriding the default SQL data types.

You Can Override the SQLType Value in an XML Schema When Declaring Attributes

You can explicitly specify a SQLType value in an XML schema, as an annotation. The SQL data type that you specify is used for XML schema validation, overriding the default SQL data types.

Only the following specific forms of such SQL data-type overrides are allowed:

- If the default SQL data type is STRING then you can override it with CHAR, VARCHAR, or CLOB.
- If the default SQL data type is RAW then you can override it with RAW or BLOB.

XML Schema Element Data Types Mapped to SQL

An XML element declaration can specify its XML Schema data type using a primitive type, a local or global simpleType, a local or global complexType, or a reference to a global element (ref=".."). The SQL data type and its associated information are derived from the base XML Schema type.

An element declaration can specify its XML Schema data type in terms of any of the following:

- Any of the ways for specifying type for an attribute declaration. See XML Schema Attribute Data Types Mapped to SQL.
- 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.
- Override of 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 its sub-elements and attributes. You can override this mapping by explicitly specifying a value for attribute SQLType in the input XML schema.



Override of 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 its 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.

How XML Schema simpleType Is Mapped to SQL

XML simpleType is mapped to SQL object types in various ways, depending on how the simpleType is defined.

Figure 18-1 illustrates one such mapping, XML string type to SQL VARCHAR2 or CLOB.



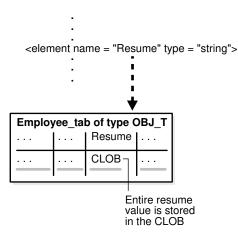


Figure 18-1 simpleType Mapping: XML Strings to SQL VARCHAR2 or CLOB

Table 18-3 through Table 18-6 present the default mapping of XML Schema simpleType to SQL, as specified in the XML Schema definition.

For example:

- An 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 18-3	XML Schema String Data Types Mapped 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)	CHAR, CLOB

Table 18-4 XML Schema Binary Data Types (hexBinary/base64Binary) Mapped 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



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
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

T I I I I I I I I I I		
Table 18-5	Default Mapping of Numeric XML S	Schema Primitive Types to SQL

Table 18-6 XML Schema Date and Time Data Types Mapped 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 18-7 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

XML Schema Primitive or Derived Type	Default SQL Data Type	Compatible SQL Data Types
Name(string)	VARCHAR2(4000)	CLOB, CHAR
NCName(string)	VARCHAR2(4000)	CLOB, CHAR
ID	VARCHAR2(4000)	CLOB, CHAR
IDREF	VARCHAR2(4000)	CLOB, CHAR
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, NVARCHAR2, and NCLOB SQLType Values Are Not Supported for SQLType Oracle XML DB does *not* support NCHAR, NVARCHAR2, and NCLOB as values for attribute SQLType: You cannot specify that an XML element or attribute is to be of type NCHAR, NVARCHAR2, or NCLOB. Also, if you provide your own data type, do not use any of these data types.
- simpleType: How XML Strings Are Mapped to SQL VARCHAR2 Versus CLOB If an XML schema specifies a data type as a string with maxLength less than 4000, it is mapped to a VARCHAR2 object attribute of the specified length. If maxLength is not specified in the schema then the XML Schema data type can only be mapped to a LOB.
- How XML Schema Time Zones Are Mapped to SQL
 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.

NCHAR, NVARCHAR2, and NCLOB SQLType Values Are Not Supported for SQLType

Oracle XML DB does *not* support NCHAR, NVARCHAR2, and NCLOB as values for attribute SQLType: You cannot specify that an XML element or attribute is to be of type NCHAR, NVARCHAR2, or NCLOB. Also, if you provide your own data type, do not use any of these data types.

Related Topics

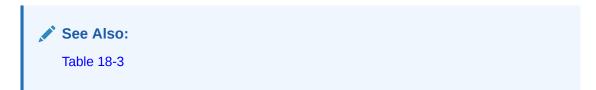
 Oracle XML DB Restrictions The restrictions associated with Oracle XML DB are listed here.



simpleType: How XML Strings Are Mapped to SQL VARCHAR2 Versus CLOB

If an XML schema specifies a data type as a string with maxLength less than 4000, it is mapped to a VARCHAR2 object attribute of the specified length. If maxLength is not specified in the schema then the XML Schema data type 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.



How XML Schema Time Zones Are Mapped to SQL

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.

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 it maps all other date types to SQL data type DATE.

SQL data types TIMESTAMP and DATE do not permit a time-zone indicator. For this reason, if your application needs time-zone information then you must use attribute SQLType to specify SQL data type TIMESTAMP WITH TIME ZONE. 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"/>
```

• Use Trailing Z to Indicate UTC Time Zone

You can specify an XML Schema time-zone component as z, to indicate UTC time zone. When a value with a trailing z is stored as SQL TIMESTAMP WITH TIME ZONE, the time zone is actually stored as +00:00. The retrieved value contains the trailing +00:00 and not the original z.



Use Trailing Z to Indicate UTC Time Zone

You can specify an XML Schema time-zone component as z, to indicate UTC time zone. When a value with a trailing z is stored as SQL TIMESTAMP WITH TIME ZONE, the time zone is actually stored as +00:00. The retrieved value contains the trailing +00:00 and not the original z.

For example, if the value in an input XML document is 1973-02-12T13:44:32**z** then the output is 1973-02-12T13:44:32.00000+00:00.

How XML Schema complexType Is Mapped to SQL

XML complexType is mapped to SQL object types in various ways, depending on how the complexType is defined.

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 is a varray value that is an ordered collections table (OCT).

 Attribute Specification in a complexType XML Schema Declaration When an element is based on a global complexType, attribute SQLType must be specified for the complexType declaration. You can optionally include the same SQLType attribute

Attribute Specification in a complexType XML Schema Declaration

within the element declaration.

When an element is based on a global complexType, attribute SQLType must be specified for the complexType declaration. You can optionally include the same SQLType attribute 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:

```
xdb:SQLType="NUMBER"/>
</xs:complexType name="PartType" xdb:SQLType="PART_T">
<xs:complexType name="PartType" xdb:SQLType="PART_T">
<xs:complexType name="Id" xdb:SQLName="PART_NUMBER" xdb:SQLType="VARCHAR2">
<xs:attribute ame="Id" xdb:SQLName="VARCHAR2">
<xs:attributename="Id" xdb:SQLName="Id" xdb:SQLName="VARCHAR2">
<xs:attributename="Id" xdb:SQLName="Id" xdb:SQLName="QUANTITY"/>
</xs:attributename="Quantity" type="moneyType" xdb:SQLName="QUANTITY"/>
<xs:attributename="UnitPrice" type="quantityType" xdb:SQLName="UNITPRICE"/>
</xs:complexType>
```

complexType Extensions and Restrictions in Oracle XML DB

In XML Schema, complexType values are declared based on complexContent and simpleContent. Oracle XML DB defines various extensions and restrictions to complexType.

- simpleContent is declared as an extension of simpleType.
- complexContent is declared as one of the following:
 - Base type
 - complexType extension
 - complexType restriction
- complexType Declarations in XML Schema: Handling Inheritance For complexType, Oracle XML DB handles inheritance in an XML schema differently for types that extend and types that restrict other complex types
- How a complexType Based on simpleContent Is Mapped to an Object Type
 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, which
 corresponds to the body value. The data type of the body attribute is based on a
 simpleType that defines the body type.
- How any and anyAttribute Declarations Are Mapped to Object Type Attributes
 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.

complexType Declarations in XML Schema: Handling Inheritance

For complexType, Oracle XML DB handles inheritance in an XML schema differently for types that extend and types that restrict other complex types

- 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 subcomplex 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 18-9 shows the registration of an XML schema that defines a base <code>complexType</code> Address and two extensions <code>USAddress</code> and <code>IntlAddress</code>.

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.

Example 18-10 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.

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 18-10, 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),
"country" VARCHAR2(4000)) NOT FINAL;
CREATE TYPE usaddr_t UNDER addr_t;
```

Example 18-9 XML Schema Inheritance: complexContent as an Extension of complexTypes

```
DECLARE
  doc VARCHAR2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"</pre>
                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"</pre>
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;
```

Example 18-10 Inheritance in XML Schema: Restrictions in complexTypes

```
DECLARE
  doc varchar2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"</pre>
                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"</pre>
                       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"</pre>
                           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;
```



How a complexType Based on simpleContent Is Mapped to an Object Type

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, which corresponds to the body value. The data type of the body attribute is based on a simpleType that defines the body type.

For the XML schema of Example 18-11, Oracle XML DB generates the following type:

CREATE TYPE obj_t AS OBJECT(SYS_XDBPD\$ XDB.XDB\$RAW_LIST_T, SYS XDBBODY\$ VARCHAR2(4000));

Example 18-11 XML Schema complexType: Mapping complexType to simpleContent

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"</pre>
             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;
```

How any and anyAttribute Declarations Are Mapped to Object Type Attributes

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.



Note:

Starting with Oracle Database 12c Release 2 (12.2.0.1), when an XML schema is registered for object-relational XMLType storage by the common user of a multitenant container database (CDB) or by an application common user, you must annotate the complex type with xdb:SQLType to specify the corresponding SQL type to use. Otherwise, an error is raised.

The code in Example 18-12 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.

For the XML schema of Example 18-12, 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));
```

Example 18-12 XML Schema: Mapping complexType to any/anyAttribute

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"</pre>
             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;
```

Creating XML Schema-Based XMLType Columns and Tables

After an XML schema has been registered with Oracle XML DB, you can reference it when you define XMLType tables or columns.

If you specify no storage model when creating an XMLType table or column for XML Schemabased 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 *objectrelational* storage is used.



Example 18-13 shows how to manually create table purchaseorder, the default table for PurchaseOrder elements.

The CREATE TABLE statement of Example 18-13 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.

The XML schema referenced Example 18-13 specifies that table purchaseorder 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 document is stored as a row in table purchaseorder.

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.

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.

XML schema registration automatically generates ordered collection tables (OCTs) for collections. 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 18-13 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 18-14.

A SQL*Plus DESCRIBE statement can be used to view information about an XMLType table, as shown in Example 18-15.

The output of the DESCRIBE statement of Example 18-15 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

Example 18-13 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$)));
```

Example 18-14 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$))));
```

Example 18-15 Using DESCRIBE with an XML Schema-Based XMLType Table

Related Topics

Local and Global XML Schemas An XML schema can be registered as local (visible only to its owner, by default) or global (visible to all database users, by default).

Overview of Partitioning XMLType Tables and Columns Stored Object-Relationally

When you partition an object-relational XMLType table or a table with an XMLType column that is stored object-relationally and you use 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 11g Release 1 (11.1).
- Equipartitioning of XMLType data that is stored out of line is not available in releases prior to Oracle Database 11g 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'

- Examples of Partitioning XMLType Data Stored Object-Relationally You can specify partitioning information for an object-relational XMLType base table during either the XML schema registration or the table creation. Examples here illustrate this.
- Partition Maintenance for XMLType Data Stored Object-Relationally 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.

See Also:

Oracle Database SQL Language Reference for information about creating tables with partitions using keywords GLOBAL and LOCAL

Examples of Partitioning XMLType Data Stored Object-Relationally

You can specify partitioning information for an object-relational XMLType base table during either the XML schema registration or the table creation. Examples here illustrate this.

- During XML schema registration, using XML Schema annotation xdb:tableProps
- During table creation using CREATE TABLE

Example 18-16 and Example 18-17 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 18-16 shows element PurchaseOrder from the purchase-order XML schema, annotated to partition the base table and its child table of line items.

Example 18-17 specifies the same partitioning as in Example 18-16, but it does so during the creation of the base table purchaseorder.

Example 18-16 and Example 18-17 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:

- Example A-2
- 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

Example 18-16 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 18-17 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)));

Partition Maintenance for XMLType Data Stored Object-Relationally

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 MOVE 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 <code>copy_indexes => 0</code> and <code>copy_constraints => false</code> for PL/SQL procedure <code>DBMS_REDEFINITION.copy_table_dependents</code>, 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

Specification of Relational Constraints on XMLType Tables and Columns

For XMLType data stored object-relationally, you can specify typical relational constraints for elements and attributes that occur only once in an XML document.

Example 18-18 defines uniqueness and foreign-key constraints on XMLType table purchaseorder in standard database schema OE.

For XMLType 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 18-18 is similar to Example 3-10, which defines a uniqueness constraint on a binary XML table. But in addition, Example 18-18 defines a foreign-key constraint that requires element User of each OE.purchaseorder document to be the e-mail address of an employee that is in table employees of standard database schema HR.

Just as for Example 3-10, the uniqueness constraint reference_is_unique of Example 18-18 ensures the uniqueness of element Reference across all documents stored in the table. The foreign key constraint user_is_valid ensures that the value of element User corresponds to a value in column email of table HR.employees.

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

See Also: Enforcing Referential Integrity Using SQL Constraints, and Example 3-10 in particular Enforcing XML Data Integrity Using the Database for information about defining contraints for XMLType data stored as binary XML

Example 18-18 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
```

Adding Unique Constraints to the Parent Element of an Attribute

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. You might, for example, want to create a unique key based on an attribute of an element that repeats itself (a collection).



Related Topics

Adding Unique Constraints to the Parent Element of an Attribute

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. You might, for example, want to create a unique key based on an attribute of an element that repeats itself (a collection).

Adding Unique Constraints to the Parent Element of an Attribute

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. You might, for example, want to create a unique key based on an attribute of an element that repeats itself (a collection).

Example 18-19 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.

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.

Example 18-19 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"</pre>
                                     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>',
            => FALSE,
   LOCAL
  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
```

Out-Of-Line Storage of XMLType Data

By default, when XMLType data is stored object-relationally a child element is mapped to an embedded SQL object attribute. Sometimes better performance can be obtained by storing some XMLType data out of line. Use XML schema annotation xdb:SQLInline to do this.

- Setting Annotation Attribute xdb:SQLInline to false for Out-Of-Line Storage Set XML schema annotation xdb:SQLInline to false to store an XML fragment out of line. The element is mapped to a SQL object type with an embedded REF attribute, which points to another XMLType instance that is stored out of line and that corresponds to the XML fragment.
- Storing Collections in Out-Of-Line Tables
 You can store collection items out of line. Instead of a single REF column, the parent
 element contains a varray of REF values that point to the collection members.



Setting Annotation Attribute xdb:SQLInline to false for Out-Of-Line Storage

Set XML schema annotation xdb:SQLInline to false to store an XML fragment out of line. The element is mapped to a SQL object type with an embedded REF attribute, which points to another XMLType instance that is stored out of line and that corresponds to the XML fragment.

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 18-2 illustrates the mapping of complexType to SQL for out-of-line storage.

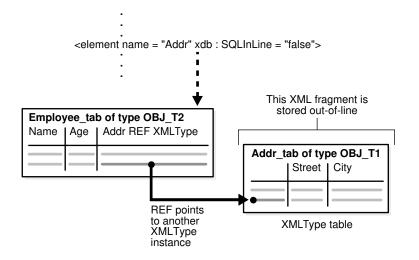


Figure 18-2 Mapping complexType to SQL for Out-Of-Line Storage

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.

In Example 18-20, 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 18-21.

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 18-22 illustrates querying table addr_tab directly to obtain the distinct city information for all employees.

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.

Example 18-20 Setting SQLInline to False for Out-Of-Line Storage

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"</pre>
             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"</pre>
                     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"</pre>
                 xdb:defaultTable="EMP TAB"/>
     </schema>';
BEGIN
  DBMS XMLSCHEMA.registerSchema(
    SCHEMAURL => 'emp.xsd',
SCHEMADOC => doc,
    ENABLE HIERARCHY => DBMS XMLSCHEMA.ENABLE HIERARCHY NONE);
END;
```

Example 18-21 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 Null? Type Name _____ TABLE of SYS.XMLTYPE(XMLSchema "emp.xsd" Element "Addr") STORAGE Object-relational TYPE "ADDR T" DESCRIBE emp t emp t is NOT FINAL Null? Type Name SYS XDBPD\$ XDB.XDB\$RAW LIST T

Name Age		VARCHAR2(4000 CHAR) NUMBER
Addr DESCRIBE addr_t Name	Null?	REF OF XMLTYPE
SYS_XDBPD\$ Street City		XDB.XDB\$RAW_LIST_T VARCHAR2(4000 CHAR) VARCHAR2(4000 CHAR)

Example 18-22 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



Storing Collections in Out-Of-Line Tables

You can store collection items out of line. 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 18-23.

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 18-20. 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 18-24.

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 18-26 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 <code>emp_tab_reflist</code> and <code>emp_tab</code>.

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 18-27.

Example 18-23 Storing a Collection Out of Line

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"</pre>
             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"</pre>
                    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>
```



Example 18-24 Generated Out-Of-Line Collection Type

Example 18-25 Renaming an Intermediate Table of REF Values

Example 18-26 XPath Rewrite for an Out-Of-Line Collection



		Eve Fong	E Street					
		George Hu	G Street					
		Iris Jones	I Street					
		Karl Luomo	K Street					
		Marina Namur	M Street					
		Omar Pinano	0 Street					
		Quincy Roberts	Q Street					
		8 rows selected.						
	4	TABLE ACCESS FULL	EMP TAB REFLIST	I	32	640	2	(0) 00:00:01
	5	TABLE ACCESS BY INDEX ROWID	EMP_TAB	Ì	1	29	1	(0) 00:00:01
*	6	INDEX UNIQUE SCAN	SYS_C005567	1	1	1	0	(0) 00:00:01

Example 18-27 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 18-26 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	1			
*	7		INDEX UNIQUE SCAN	SYS_C005567	1	1	0	(0) 00:00:01

Considerations for Working with Complex or Large XML Schemas

XML schemas can be complex. Examples of complex schemas include those that are recursive and those that contain circular or cyclical references. Working with complex or large XML schemas can be challenging and requires taking certain considerations into account.

Circular and Cyclical Dependencies Among XML Schemas

The W3C XML Schema Recommendation lets complexTypes and global elements contain recursive references. This kind of structure allows for instance documents where the element in question can appear an infinite number of times in a recursive hierarchy.

- Support for Recursive Schemas A REF to a recursive structure in an out-of-line table can make it difficult to rewrite XPath queries, because it is not known at compile time how deep the structure is. To enable XPath rewrite, a DOCID column points back to the root document in any recursive structure.
- XML Fragments Can Be Mapped to Large Objects (LOBs)
 You can specify the SQL data type to use for a complex element as being CLOB or BLOB.
- ORA-01792 and ORA-04031: Issues with Large XML Schemas Errors ORA-01792 and ORA-04031 can be raised when you work with large or complex XML schemas. You can encounter them when you register an XML schema or you create a table that is based on a global element defined by an XML schema.

1

• Considerations for Loading and Retrieving Large Documents with Collections Oracle XML DB configuration file xdbconfig.xml has parameters that control the amount of memory used by the loading operation: xdbcore-loadableunit-size and xdbcorexobmem-bound.

Circular and Cyclical Dependencies Among XML Schemas

The W3C XML Schema Recommendation lets complexTypes and global elements contain recursive references. This kind of structure allows for instance documents where the element in question can appear an infinite number of times in a recursive hierarchy.

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.

Example 18-28 An XML Schema with Circular Dependency

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"</pre>
           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"</pre>
                  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 18-28 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 involve circular schema dependencies. 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.
- complexType Declarations in 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 where the cycle would be completed. Part of the data is stored out of line, but it is retrieved as part of the parent XML document.
- 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.



For Circular XML Schema Dependencies Set Parameter GENTABLES to TRUE

Oracle XML DB supports XML schemas that involve circular schema dependencies. 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 in 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 where the cycle would be completed. Part of the data is stored out of line, but it is 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 18-3 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 18-29 creates a cycle of length two:

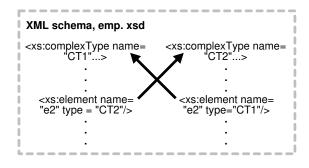
SQL types do not allow cycles in type definitions. However, they do support **weak cycles**, that is, cycles involving REF (reference) object attributes. Cyclic XML schema definitions are mapped to SQL object types in such a way that cycles are avoided by forcing SQLInline = "false" at the appropriate points. This creates a weak SQL cycle.

For the XML schema of Example 18-29, 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 18-3 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 18-30, type SectionT does this.

For the XML schema of Example 18-30, Oracle XML DB generates the following types:

```
CREATE TYPE body_coll AS VARRAY(32767) OF VARCHAR2(32767<sup>3</sup>);
CREATE TYPE section_t AS OBJECT (SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
"title" VARCHAR2(32767<sup>3</sup>),
"body" BODY_COLL,
"section" XDB.XDB$REF LIST T) NOT FINAL;
```

Note:

In Example 18-30, 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.

Figure 18-4 illustrates schematically how a complexType can reference itself.

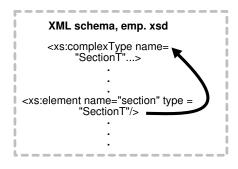


Figure 18-4 Self-Referencing Complex Type within an XML Schema

Language Reference.

¹ This value of 32767 assumes that the value of initialization parameter MAX_STRING_SIZE is EXTENDED. See Oracle Database SQL

```
DECLARE
  doc VARCHAR2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"</pre>
                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;
```

Example 18-29 XML Schema: Cycling Between complexTypes

Example 18-30 XML Schema: Cycling Between complexTypes, Self-Reference

```
DECLARE
  doc VARCHAR2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"</pre>
                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"</pre>
                         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;
```

Related Topics

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.



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.

This is illustrated in Figure 18-5.

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.

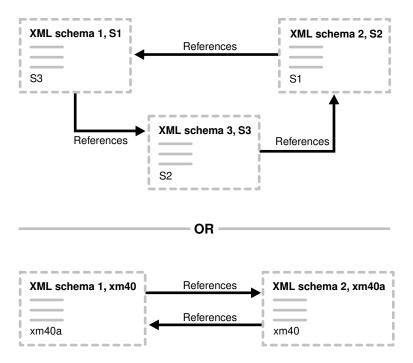


Figure 18-5 Cyclical References Between XML Schemas

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 18-31 fails, if xm40a.xsd does not exist.

XML schema xm40.xsd can, however, be created if you specify option FORCE => TRUE, as in Example 18-32:

However, an attempt to use XML schema xm40.xsd, as in Example 18-33, fails.

If you register xm40a.xsd using the FORCE option, as in Example 18-34, then both XML schemas can be used, as shown by the CREATE TABLE statements.

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.

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.

Example 18-31 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"</pre>
                        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>',
        => TRUE,
 LOCAL
 GENTYPES => TRUE,
 GENTABLES => TRUE);
END;
/
```

Example 18-32 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;
/
```

Example 18-33 Trying to Create a Table Using a Cyclic XML Schema

CREATE TABLE foo OF XMLType XMLSCHEMA "xm40.xsd" ELEMENT "Emp";



Example 18-34 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"</pre>
                        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";
```

Support for Recursive Schemas

A REF to a recursive structure in an out-of-line table can make it difficult to rewrite XPath queries, because it is not known at compile time how deep the structure is. To enable XPath rewrite, a DOCID column points back to the root document in any recursive structure.

This enables some XPath queries to use the out-of-line tables directly and join back using this column.

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.



Example 18-35 Recursive XML Schema

```
<schema targetNamespace="AbcNS" xmlns="http://www.w3.org/2001/XMLSchema"</pre>
          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>
```

defaultTable Shared Among Common Out-Of-Line Elements

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 in the same table as the sub-elements.

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 against that table, using the DOCID.

DOCID Column Creation Disabling
 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.

defaultTable Shared Among Common Out-Of-Line Elements

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 in the same table as the sub-elements.

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 18-36 shows an XML schema with an out-of-line table that is stored in ABCSECTIONTAB.

Both of the out-of-line AbcSection elements in Example 18-36 share the same default table, ABCSECTIONTAB.

However, Example 18-37 illustrates *invalid* default table sharing: recursive elements (XyZSection) do not share the same out-of-line table.



The following query cannot be rewritten.

```
SELECT XMLQuery('//XyzSection' PASSING OBJECT_VALUE RETURNING CONTENT)
FROM xyzcode;
```

Example 18-36 Out-of-line Table

```
<schema targetNamespace="AbcNS" xmlns="http://www.w3.org/2001/XMLSchema"</pre>
           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"</pre>
                 xdb:defaultTable="ABCSECTIONTAB"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Example 18-37 Invalid Default Table Sharing

```
<schema targetNamespace="XyzNS" xmlns="http://www.w3.org/2001/XMLSchema"</pre>
        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"</pre>
                 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"</pre>
                 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>
```

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 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 18-36:

```
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.

DOCID Column Creation Disabling

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.

OPTIONS is an input parameter of data type PLS_INTEGER. Its default value is 0, meaning that no options are used. To inhibit the generation of column DOCID, set parameter OPTIONS to DBMS XMLSCHEMA.REGISTER NODOCID (which is 1).



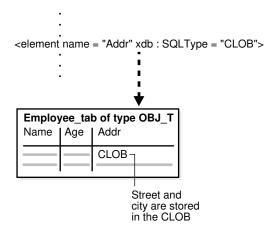
XML Fragments Can Be Mapped to Large Objects (LOBs)

You can specify the SQL data type to use for a complex element as being CLOB or BLOB.

In Figure 18-6, for example, an entire XML fragment is stored in a LOB attribute.

In Example 18-38, the XML schema defines element Addr using the annotation SQLType = "CLOB":

Figure 18-6 Mapping complexType XML Fragments to CLOB Instances



Example 18-38 Oracle XML DB XML Schema: Mapping complexType XML Fragments to LOBs

```
DECLARE
doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"</pre>
```



```
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);
```

ORA-01792 and ORA-04031: Issues with Large XML Schemas

Errors ORA-01792 and ORA-04031 can be raised when you work with large or complex XML schemas. You can encounter them 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 4096
- ORA-04031: unable to allocate *string* bytes of shared memory ("*string*", "*string*", "*string*")

These errors are raised when you try 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.

They are raised only when creating an XMLType table or column that uses object-relational storage. 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 4096 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 4096-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 <code>complexType</code> elements as <code>CLOB</code> values, rather than as objects. This is achieved by annotating the <code>complexType</code> or the usage of the <code>complexType</code> with <code>SQLType</code> = "CLOB".

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

Considerations for Loading and Retrieving Large Documents with Collections

Oracle XML DB configuration file xdbconfig.xml has parameters that control the amount of memory used by the loading operation: xdbcore-loadableunit-size and xdbcore-xobmembound.

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_REPOS.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, the 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:

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 a smaller number of disk accesses, but it requires more memory. This is controlled by configuration parameter xdbcore-loadableunit-size, whose default value is 16 KB. You can indicate the amount of memory to be given to a document by setting parameter xdbcore-xobmem-bound, which defaults to 1 MB. The values of 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 about how to use the memory optimally.



When a 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 Configuration Parameters xdbcore-loadableunit-size and xdbcore-xobmembound

Use PGA size and trial and error to determine the best values for configuration parameters xdbcore-loadableunit-size and xdbcore-xobmem-bound.

Guidelines for Configuration Parameters xdbcore-loadableunit-size and xdbcorexobmem-bound

Use PGA size and trial and error to determine the best values for configuration parameters xdbcore-loadableunit-size and xdbcore-xobmem-bound.

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, the value of xdbcore-loadableunit-size should be as close as possible to the value of xobcore-xobmem-bound.

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 reduce the value of xdbcore-xobmem-bound and set xdbcoreloadableunot-size to half of that value. Continue this way until the documents load successfully.

If the load operation 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 parameter values for further performance improvements.

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 event 31098 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. The log is written to the user session trace file, typically found in ORACLE_BASE/ diag/rdbms/ORACLE_SID/ORACLE_SID/Udump. This trace output can be a useful aid in diagnosing problems during XML schema registration.

19

XPath Rewrite for Object-Relational Storage

For XMLType data stored object-relationally, queries involving XPath expression arguments to various SQL functions can often be automatically rewritten to queries against the underlying SQL tables, which are highly optimized.

Overview of XPath Rewrite for Object-Relational Storage

Oracle XML DB can often optimize queries that use XPath expressions — for example, queries involving SQL functions such as XMLQuery, XMLTable, and XMLExists, 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.

- Common XPath Expressions that Are Rewritten The most common XPath expressions that are rewritten during XPath rewrite are described.
- XPath Rewrite for Out-Of-Line Tables
 XPath expressions that involve elements stored out of line can be automatically rewritten.
 The rewritten guery involves a join with the out-of-line table.
- Guidelines for Using Execution Plans to Analyze and Optimize XPath Queries
 Guidelines are presented for using execution plans to analyze query execution in order to
 (a) determine whether XPath rewrite occurs and (b) optimize query execution by using
 secondary indexes. These guidelines apply only to XMLType data that is stored object relationally.

Related Topics

• Performance Tuning for XQuery

A SQL query that involves XQuery expressions can often be automatically rewritten (optimized) in one or more ways. This optimization is referred to as **XML query rewrite** or optimization. When this happens, the XQuery expression is, in effect, evaluated directly against the XML document without constructing a DOM in memory.

Overview of XPath Rewrite for Object-Relational Storage

Oracle XML DB can often optimize queries that use XPath expressions — for example, queries involving SQL functions such as XMLQuery, XMLTable, and XMLExists, 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 or when an XMLType view is built on relational data.
- When you use an XMLIndex index.

The first case, rewriting queries that use object-relational XML data or XMLType views, is covered here. The XMLType views can be XML schema-based or not. Object-relational storage of XMLType data is always XML schema-based. Examples in this chapter are related to XML schema-based tables.

When XPath rewrite is possible for object-relational XML data, the database optimizer can 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, so Oracle XML DB executes XQuery-based and XPath-based queries at near-relational speed.

In certain cases, XPath rewrite is not possible. This typically 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, which is generally more costly, especially if the number of documents to be processed is large.

Example 19-1 illustrates XPath rewrite for a simple query that uses an XPath expression.

Example 19-1 XPath Rewrite

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';
```

Related Topics

- Relational Views over XML Data Relational database views over XML data provide conventional, relational access to XML content.
- XMLType Views You can create XMLType views over relational and object-relational data.
- Indexing XMLType Data Stored Object-Relationally You can effectively index XMLType data that is stored object-relationally by creating B-tree indexes on the underlying database columns that correspond to XML nodes.
- XMLIndex

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 A-2. Without such annotations, this example would use p."XMLDATA". "Requestor", not p."XMLDATA". ".REQUESTOR".

• XML Schema Annotation Guidelines for Object-Relational Storage

For XMLType data stored object-relationally, careful planning is called for, to optimize performance. Similar considerations are in order as for relational data: entity-relationship models, 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.

Common XPath Expressions that Are Rewritten

The most common XPath expressions that are rewritten during XPath rewrite are described.

Table 19-1 presents the descriptions

Table 19-1 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):	Involves traversals over object type attributes only, where the attributes are simple scalar or object types				
/PurchaseOrder/@Reference	themselves.				
/PurchaseOrder/Requestor					
Collection traversal expressions:	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.				
/PurchaseOrder/LineItems/LineItem/Part/@Id					
Predicates:	Predicates in the XPath are rewritten into SQL				
[Requestor = "Sarah J. Bell"]	predicates.				
List index (positional predicate):	Indexes are rewritten to access the nth item in a collection.				
LineItem[1]					
Wildcard traversals:	If the wildcard can be translated to one or more simple				
/PurchaseOrder/*/Part/@Id	XPath expressions, then it is rewritten.				
Descendant axis (XML schema-based data only), without recursion:	Similar to a wildcard expression. The descendant axis is rewritten if it can be mapped to one or more simple				
/PurchaseOrder//Part/@Id	XPath expressions.				
Descendant axis (XML schema-based data only), with <i>recursion</i> : /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 for information about rewrite of XQuery expressions



XPath Rewrite for Out-Of-Line Tables

XPath expressions that involve elements stored out of line can be automatically rewritten. The rewritten query involves a join with the out-of-line table.

Example 19-2 shows such a query. The XQuery expression 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 19-3. An explain-plan fragment for the same query as in Example 19-2 shows that the city index is picked up.

Note:

When gathering statistics for the optimizer on an XMLType table that is stored objectrelationally, 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.

Example 19-2 XPath Rewrite for an Out-Of-Line Table

Eve Fong George Hu Iris Jones Karl Luomo Marina Namur Omar Pinano Quincy Roberts

8 rows selected.



Example 19-3 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 INDE	K 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

Guidelines for Using Execution Plans to Analyze and Optimize XPath Queries

Guidelines are presented for using execution plans to analyze query execution in order to (a) determine whether XPath rewrite occurs and (b) optimize query execution by using secondary indexes. These guidelines apply only to XMLType data that is stored object-relationally.

Use these guidelines together, taking all that apply into consideration.

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.

You can use PL/SQL procedure DBMS_XMLSTORAGE_MANAGE.XPath2TabColMapping to find the names of the underlying tables and columns that correspond to a given XPath expression.

- Guideline: Look for underlying tables versus XML functions in execution plans
- The execution plan of a query that is rewritten refers to the names of the object-relational tables and columns that underlie the queried XMLType data. These names can be meaningful to you if they are derived from XML element or attribute names or if XML Schema annotation xdb:defaultTable was used.
- Guideline: Name the object-relational 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 You can sometimes improve the performance of a query that is rewritten to include a SQL predicate, by creating an index that applies to the column targeted by the predicate.
- Guideline: Create indexes on ordered collection tables If a collection is stored as an ordered collection table (OCT) or as an XMLType instance, then you can directly access members of the collection. Each member becomes a table row, so you can access it directly with SQL. You can often improve performance by indexing such collection members.
- 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.

Related Topics

• XPath Rewrite for Out-Of-Line Tables XPath expressions that involve elements stored out of line can be automatically rewritten. The rewritten query involves a join with the out-of-line table.



See Also:

Oracle Database PL/SQL Packages and Types Reference for information about procedure XPath2TabColMapping

Guideline: Look for underlying tables versus XML functions in execution plans

The execution plan of a query that is rewritten refers to the names of the object-relational tables and columns that underlie the queried XMLType data. These names can be meaningful to you if they are derived from XML element or attribute names or if XML Schema annotation xdb:defaultTable was used.

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 generally 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 19-4 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.

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.

Example 19-4 Execution Plan Generated When XPath Rewrite Does Not Occur

```
Predicate Information (identified by operation id):
```

Guideline: Name the object-relational 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.



For collection tables, there is no corresponding XML schema annotation. To give user-friendly names to your collection tables you must first register the XML schema. Then you can use PL/SQL procedure DBMS_XMLSTORAGE_MANAGE.renameCollectionTable to rename the tables that were created during registration, which have system-generated names.

🖍 See Also:

Oracle Database PL/SQL Packages and Types Reference for information about procedure renameCollectionTable

Guideline: Create an index on a column targeted by a predicate

You can sometimes improve the performance of a query that is rewritten to include a SQL predicate, by creating an index that applies to the column targeted by the 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 19-1 illustrates XPath rewrite for a query that includes a WHERE clause. Example 19-5 shows the predicate information from an execution plan for this query.

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 <code>extractValue</code> shortcut to create the index, by specifying an XPath expression that targets the purchase-order requestor data.

You can obtain the names of the underlying table and columns that correspond to a given XPath expression using procedure DBMS_XMLSTORAGE_MANAGE.XPath2TabColMapping. Example 19-6 illustrates this for the XPath expression /PurchaseOrder/Requestor used in the WHERE clause of Example 19-1.

If you provide an XPath expression that contains a wildcard or a descendent axis then multiple tables and columns might be selected. In that case procedure XPath2TabColMapping returns multiple <Mapping> elements, one for each table-column pair.

You can then use the table and column names retrieved this way in a CREATE INDEX statement to create an index that corresponds to the XPath expression. Example 19-7 shows three equivalent ways to create a B-tree index on the predicate-targeted column.

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 19-8 illustrates this.

Example 19-9 shows an execution plan that indicates that the index is picked up.

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 19-10, which targets an XPath expression, thus has the same effect as Example 19-8, which targets the corresponding object-relational column.

See Also:

- Indexing Non-Repeating Text Nodes or Attribute Values for information about using the shortcut of XMLCast applied to XMLQuery and the extractValue shortcut to index singleton data
- Oracle Database PL/SQL Packages and Types Reference for information about procedure XPath2TabColMapping

Example 19-5 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)
```

Example 19-6 Using DBMS_XMLSTORAGE_MANAGE.XPATH2TABCOLMAPPING

SELECT DBMS XMLSTORAGE MANAGE.XPath2TabColMapping(USER,

Example 19-7 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'));

Example 19-8 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 19-9	Execution Plan Showing that Index Is Picked Up
--------------	--

Id Operation Name Ro	Rows	Bytes Cost	(%CPU) Time
0 SELECT STATEMENT * 1 TABLE ACCESS BY INDEX ROWID PURCHASEORDER * 2 INDEX RANGE SCAN REQUESTOR INDEX	1	524	2 (0) 00:00:01 2 (0) 00:00:01 1 (0) 00:00:01

```
Predicate Information (identified by operation id):
```

Example 19-10 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)));
```

Guideline: Create indexes on ordered collection tables

If a collection is stored as an ordered collection table (OCT) or as an XMLType instance, then you can directly access members of the collection. Each member becomes a table row, 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 <code>NESTED_TABLE_ID</code>.

Example 19-11 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 19-11 does not use the purchaseorder table from sample database schema OE. It uses a purchaseorder table that uses an ordered collection table (OCT) named lineitem_table for the collection element LineItem.

The execution plan shows a full scan of ordered collection table <code>lineitem_table</code>. This could be acceptable if there were only a few hundred documents in the <code>purchaseorder</code> 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 19-12. This index provides direct access to those purchase-order documents that have LineItem elements that reference the required part.

Example 19-11 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		Row	s		Bytes		Cost	(%CPU)	Time	
0	Ι	SELECT STATEMENT					21		2352	I	20	(10)	00:00:01	
* 1		HASH JOIN RIGHT SEMI	1		1		21		2352	L	20	(10)	00:00:01	
2		JOIN FILTER CREATE	1	:BF0000	1		22		880	L	14	(8)	00:00:01	
* 3		TABLE ACCESS FULL	1	LINEITEM TABLE	1		22		880	L	14	(8)	00:00:01	
4		JOIN FILTER USE	1	:BF0000 -	1	1	32		9504	L	5	(0)	00:00:01	
* 5		TABLE ACCESS FULL		PURCHASEORDER		1	32		9504	L	5	(0)	00:00:01	

Predicate Information (identified by operation id):

1 - access("NESTED TABLE ID"="PURCHASEORDER"."SYS NC0003400035\$")

3 - filter("SYS NC00011\$"='717951002372')

5 - filter(SYS OP BLOOM FILTER(:BF0000, "PURCHASEORDER", "SYS NC0003400035\$"))

Example 19-12 Creating an Index for Direct Access to an Ordered Collection Table

CREATE INDEX lineitem_part_index ON lineitem_table 1 (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.



Related Topics

• Diagnosis of 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.



20 XML Schema Evolution

You can use XML schema evolution to update your XML schema after you have registered it with Oracle XML DB.

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.

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.

Copy-Based Schema Evolution

You perform copy-based XML schema evolution using PL/SQL procedure DBMS_XMLSCHEMA.copyEvolve. This backs up existing instance documents to temporary XMLType tables, 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-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.

Related Topics

• XML Schema Storage and Query: Basic

XML Schema is a standard for describing the content and structure of XML documents. You can register, update, and delete an XML schema used with Oracle XML DB. You can define storage structures to use for your XML schema-based data and map XML Schema data types to SQL data types.

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 In-Place XML Schema Evolution.

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.

Copy-Based Schema Evolution

You perform copy-based XML schema evolution using PL/SQL procedure DBMS_XMLSCHEMA.copyEvolve. This backs up existing instance documents to temporary XMLType tables, 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.

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 An evolved version of a purchase-order XML schema is shown. It is used in examples that illustrate the use of copy-based XML schema evolution.
- COPYEVOLVE Parameters and Errors
 The parameters of PL/SQL procedure DBMS_XMLSCHEMA.copyEvolve are described, as are
 the errors associated with this procedure.
- Limitations of Procedure COPYEVOLVE The use of PL/SQL procedure DBMS_XMLSCHEMA.copyEvolve involves certain limitations.
- Guidelines for Using Procedure COPYEVOLVE
 General guidelines for using PL/SQL procedure DBMS_XMLSCHEMA.copyEvolveare
 presented, as well as guidelines that are specific to particular contexts.
- Update of Existing XML Instance Documents Using an XSLT Stylesheet
 After you modify a registered XML schema, you must update any existing XML instance
 documents that use the schema. You do this by applying an XSLT stylesheet to each of the
 instance documents. The stylesheet represents the difference between the old and new
 XML schemas.



Examples of Using Procedure COPYEVOLVE

Several examples are presented of using PL/SQL procedureDBMS_XMLSCHEMA.copyEvolve to update an XML schema. (Be sure to back up all registered XML schemas and XML documents that reference them, before using the procedure.)

Scenario for Copy-Based Evolution

An evolved version of a purchase-order XML schema is shown. It is used in examples that illustrate the use of copy-based XML schema evolution.

Example 20-1 shows a *partial* listing of a revised version of the purchase-order XML schema of Example A-2. See Example A-3 for the *complete* revised schema listing. Text that is in **bold** here is new or different from that in the original schema.

```
Example 20-1 Revised Purchase-Order XML Schema
```

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"</pre>
           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"</pre>
                  xdb:SQLName="BILLING ADDRESS"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"</pre>
                 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"</pre>
                  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"</pre>
                  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"</pre>
                  xdb:SQLType="NUMBER"/>
  </xs:complexType>
```



```
<xs:complexType name="PartType" xdb:SQLType="PART T">
   <rs:simpleContent>
      <xs:extension base="UPCCodeType">
        <xs:attribute name="Description" type="DescriptionType" use="required"</pre>
                      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"/>
   \langle xs:all \rangle
 </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"</pre>
                    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"/>
      <rs:element name="City" type="CityType"/>
      <xs:choice>
        <xs:sequence>
         <xs:element name="State" type="StateType"/>
         <xs:element name="ZipCode" type="ZipCodeType"/>
```

ORACLE

```
</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:simpleType</pre>
</xs:simpleType name="UPCCodeType">
<xs:simpleType</pre>
</xs:simpleType name="UPCCodeType">
<xs:simpleType</pre>
</xs:simpleType name="UPCCodeType">
<xs:simpleType</pre>
</xs:simpleType>
</xs:simpleType>
</xs:simpleType>
</xs:simpleType>
</xs:simpleType>
</xs:simpleType>
```

COPYEVOLVE Parameters and Errors

The parameters of PL/SQL procedure DBMS_XMLSCHEMA.copyEvolve are described, as are the errors associated with this procedure.

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 20-1 describes the individual parameters. Table 20-2 describes the errors associated with the procedure.

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.		

Table 20-1 Parameters of Procedure DBMS_XMLSCHEMA.COPYEVOLVE



Parameter	Description		
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 20-1 (Cont.) Parameters of Procedure DBMS_XMLSCHEMA.COPYEVOLVE

Table 20-2 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_nam e>" column '<column_name>'</column_name></table_nam </owner_name></schema_url>	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.
30943 XML Schema ' <schema_url>' is dependent on XML schema '<schema_url>'</schema_url></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_nam e>" column '<column_name>'</column_name></table_nam </owner_name></schema_url>	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>'</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

The use of PL/SQL procedure DBMS XMLSCHEMA.copyEvolve involves certain limitations.

- 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.
- 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 <code>Employee</code> has been registered under URL s1.xsd. It is not possible to evolve this XML schema, because table <code>tabl</code> with column <code>o</code> with <code>XMLType</code> attribute <code>x</code> is dependent on the XML schema. Although <code>copyEvolve</code> does not handle <code>XMLType</code> object attributes, it does raise an error in such cases.

Guidelines for Using Procedure COPYEVOLVE

General guidelines for using PL/SQL procedure DBMS_XMLSCHEMA.copyEvolveare presented, as well as guidelines that are specific to particular contexts.

The following general guideline applies to using 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;

 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 stylesheet 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 stylesheet.
- Top-Level Element Name Changes

PL/SQL procedure DBMS_XMLSCHEMA.copyEvolve assumes that top-level elements have not been dropped in new schemas and that their names have not been changed. If there are such changes then call procedure copyEvolve with parameter generateTables set to FALSE and parameter preserveOldDocs set to TRUE, so that backup document copies are available.

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 such columns are needed then set copyEvolve 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 to be evolved, as well as its dependents, are not used by any
 concurrent session during the XML schema evolution process.
- Rollback When Procedure DBMS_XMLSCHEMA.COPYEVOLVE Raises an Error Procedure DBMS_XMLSCHEMA.copyEvolve either completely succeeds or it raises an error. If it raises an error then it tries to roll back as much of the operation as possible.
- Failed Rollback From Insufficient Privileges In certain cases you cannot roll back a 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.
- Privileges Needed for XML Schema Evolution There are several database privileges that you might need, in order to perform copy-based XML schema evolution.

Top-Level Element Name Changes

PL/SQL procedure DBMS_XMLSCHEMA.copyEvolve assumes that top-level elements have not been dropped in new schemas and that their names have not been changed. If there are such changes then call procedure copyEvolve with parameter generateTables set to FALSE and parameter preserveOldDocs set to TRUE, so that backup document copies are available.

With those parameter values, 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 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 such columns are needed then set copyEvolve 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 to be evolved, as well as 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 it raises an error. If it raises an error then it tries 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 a 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

There are several database privileges that you might need, in order to perform copy-based XML schema evolution.

Copy-based XML schema evolution can involve dropping or creating SQL data types, so 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
- READ 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.

Update of Existing XML Instance Documents Using an XSLT Stylesheet

After you modify a registered XML schema, you must update any existing XML instance documents that use the schema. You do this by applying an XSLT stylesheet to each of the instance documents. The stylesheet represents the difference between the old and new XML schemas.

Example 20-2 shows an XSLT stylesheet, 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 20-2 evolvePurchaseOrder.xsl: XSLT Stylesheet 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

Several examples are presented of using PL/SQL procedureDBMS_XMLSCHEMA.copyEvolve to update an XML schema. (Be sure to back up all registered XML schemas and XML documents that reference them, before using the procedure.)

Example 20-3 loads a revised XML schema and evolution XSL stylesheet into Oracle XML DB Repository.

Example 20-4 shows how to use procedure DBMS_XMLSCHEMA.copyEvolve to evolve the XML schema purchaseOrder.xsd to revisedPurchaseOrder.xsd using the XSLT stylesheet evolvePurchaseOrder.xsl.

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 20-3.

Name	Туре	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.

Table 20-3 XML Schema Evolution: XMLType Table Temporary Table Columns

For XMLType columns, the temporary table has the columns shown in Table 20-4.

Name	Туре	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 20-5 shows the mapping table columns.



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_name></owner_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.

Table 20-5 Procedure COPYEVOLVE Mapping Table

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.

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

Example 20-3 Loading Revised XML Schema and XSLT Stylesheet

```
DECLARE
 res BOOLEAN;
BEGIN
 res := DBMS XDB REPOS.createResource(
                                               -- Load revised XML schema
          '/source/schemas/poSource/revisedPurchaseOrder.xsd',
          bfilename('XMLDIR', 'revisedPurchaseOrder.xsd'),
          nls charset id('AL32UTF8'));
 res := DBMS XDB REPOS.createResource(
                                               -- Load revised XSL stylesheet
           '/source/schemas/poSource/evolvePurchaseOrder.xsl',
          bfilename('XMLDIR', 'evolvePurchaseOrder.xsl'),
          nls charset id('AL32UTF8'));
```

```
END;/
```

Example 20-4 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()),
```

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>
```

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, 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. 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.
- Supported Operations for In-Place XML Schema Evolution
 Some of the operations that are supported for in-place schema evolution are described.
 Some of these are not permitted in specific contexts, which are specified.
- Guidelines for Using In-Place XML Schema Evolution Guidelines that apply to in-place XML-schema evolution are presented. (Be sure to back up your data before performing in-place schema evolution.)



• inPlaceEvolve Parameters

The parameters of PL/SQL procedure DBMS_XMLSCHEMA.inPlaceEvolve are described, as are the errors associated with this procedure.

• The diffXML Parameter Document The value of parameter diffXML of procedure DBMS_XMLSCHEMA.inPlaceEvolve is 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).

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. 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.

For the complete list of changes supported by in-place evolution, see Supported Operations for In-Place XML Schema Evolution.

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.

In addition to this general backward-compatibility restriction, there are some other restrictions for in-place evolution.

- Backward-Compatibility Restrictions Restrictions on in-place XML schema evolution are described that ensure backward compatibility of an evolved schema, so that any possible instance documents that satisfy the old XML schema also satisfy the new one.
- Other Restrictions on In-Place Evolution
 Some restrictions on in-place XML schema evolution are necessary for reasons other than backward compatibility of the evolved XML schema.

Backward-Compatibility Restrictions

Restrictions on in-place XML schema evolution are described that ensure backward compatibility of an evolved schema, so that any possible instance documents that satisfy the old XML schema also satisfy the new one.

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.

Reorder of XML Schema Constructs

You cannot use in-place evolution to reorder XML schema elements in a way that affects the DOM fidelity of XML instance documents. For example, you cannot change the order of elements within a <sequence> element in a complex type definition.

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 maxOccurs greater than one to maxOccurs equal to one. In-place



evolution thus cannot delete an element from a complex type if the deletion requires that a collection be evolved to a non-collection.

Model Changes within a complexType Element

A model is a group, choice, sequence, or all element. Within a complexType element you cannot use in-place evolution to either add a new model or replace an existing model with a model of another type (for example, replace a choice element with a sequence element).

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 20-5, complex type ShippingInstructionsType is split into two complex types, Person-Name and Contact-Info, and the ShippingInstructionsType complex type is deleted.

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.

Example 20-5 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"</pre>
                               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>
```

Reorder of XML Schema Constructs

You cannot use in-place evolution to reorder XML schema elements in a way that affects the DOM fidelity of XML 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 maxOccurs greater than one to maxOccurs equal to one. In-place evolution thus cannot delete an element from a complex type if the deletion requires that a collection be evolved to a non-collection.

Model Changes within a complexType Element

A model is a group, choice, sequence, or all element. Within a complexType element you cannot use in-place evolution to either add a new model or replace an existing model with a model of another type (for example, replace a choice element with a sequence element).

You can, however, add a global group element, that is, add a group element outside of a complexType element.

Other Restrictions on In-Place Evolution

Some restrictions on in-place XML schema evolution 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

For object-relational XMLType data, you cannot use in-place evolution to change a noncollection object type to a collection object type. An example would be adding an element to a complex type if the element is already present in the type (or in a type related through inheritance).

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

For object-relational XMLType data, you cannot use in-place evolution to change a noncollection object type to a collection object type. An example would be adding an element to a complex type if the element is already present in the type (or in a type related through inheritance).

Supported Operations for In-Place XML Schema Evolution

Some of the operations that are supported for in-place schema evolution are described. Some of these are not permitted in specific contexts, which are specified.

 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:

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:

- 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:

- 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:

- 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

Guidelines that apply to in-place XML-schema evolution are presented. (Be sure to back up your data before performing in-place 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

The parameters of PL/SQL procedure DBMS_XMLSCHEMA.inPlaceEvolve are described, as are the errors associated with this procedure.

This is the signature of procedure DBMS XMLSCHEMA.inPlaceEvolve:

procedure inPlaceEvolve(schemaURL IN VARCHAR2, diffXML IN XMLType, flags IN NUMBER);

Table 20-6 describes the individual parameters.

Table 20-6	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 The diffXML Parameter Document.



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 performe 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, <i>except</i> 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).

Table 20-6 (Cont.) Parameters of Procedure DBMS_XMLSCHEMA.INPLACEEVOLVE

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.

The diffXML Parameter Document

The value of parameter diffXML of procedure DBMS_XMLSCHEMA.inPlaceEvolve is 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.

diffXML Operations and Examples

Operations that can be specified in the diffXML document supplied to procedure DBMS_XMLSCHEMA.inPlaceEvolve are described. An example XML document that conforms to the xdiff XML schema is shown.

Related Topics

xdiff.xsd: XML Schema for Comparing Schemas for In-Place Evolution
 A full listing is presented of xdiff.xsd, the Oracle XML DB-supplied XML schema to which
 the document specified as the diffXML parameter to procedure
 DBMS XMLSCHEMA.inPlaceEvolve must conform.

See Also:

- 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 commandline utility xmldiff
- Oracle Database SQL Language Reference for information on SQL function XMLDiff

diffXML Operations and Examples

Operations that can be specified in the diffXML document supplied to procedure DBMS_XMLSCHEMA.inPlaceEvolve are described. An example XML document that conforms to the xdiff XML schema is shown.

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 20-6 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 20-6 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>
```

Part VI Oracle XML DB Repository

Oracle XML DB Repository lets you version your data, implement and manage security, and use APIs to access and manipulate repository resources.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

Access to Oracle XML DB Repository Data

There are several ways to access and manipulate data in Oracle XML DB Repository, including using standard protocols such as FTP and HTTP(S)/WebDAV; Oracle XML DB resource Application Program Interfaces (APIs); and the repository views <code>RESOURCE_VIEW</code> and <code>PATH_VIEW</code>.

Configuration of Oracle XML DB Repository

Overall configuration of Oracle XML DB Repository applies to all repository resources. It does not include configuring parameters for handling events or managing XLink and XInclude processing. You use resource configuration files to configure resources.

• Use of XLink and XInclude with Oracle XML DB

You can use XLink and XInclude with resources in Oracle XML DB Repository. But the use of XLink is *deprecated*.

Repository Access Using RESOURCE_VIEW and PATH_VIEW

Predefined public views RESOURCE_VIEW and PATH_VIEW provide access to Oracle XML DB repository data. You can use Oracle SQL functions under_path and equals_path to query resources based on their path names, and functions path and depth to return resource path names and depths.

Resource Versions Oracle XML DB Repository resources can be versioned. A record is kept of all changes to a resource that is under version control.

PL/SQL Access to Oracle XML DB Repository

PL/SQL packages DBMS_XDB_CONFIG and DBMS_XDB_REPOS together provide the Oracle XML DB resource application program interface (API) for PL/SQL. You use the former to configure Oracle XML DB and its repository. You use the latter to perform other, non-configuration operations on the repository.

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. You can create, set, and modify access control lists (ACLs).

Repository Access Using Protocols You can access Oracle XML DB Repository data using protocols FTP and HTTP(S)/ WebDAV.



User-Defined Repository Metadata

You can create your own metadata to associate with XML data stored in Oracle XML DB Repository.

- Oracle XML DB Repository Events The use of repository events to trigger application actions is deprecated in Oracle Database 21c (21.3). There is no replacement.
- Guidelines for Oracle XML DB Applications in Java
 Design guidelines are presented for writing Oracle XML DB applications in Java. This
 includes guidelines for writing and configuring Java servlets for Oracle XML DB.
- Data Access Using URIs

You can generate and store URIs in the database and use them to retrieve the database data they target. There are three kinds of URIs you can use this way: DBUris, XDBUris, and HTTPUris.

Native Oracle XML DB Web Services
 Your applications can access Oracle Database using native Oracle XML DB Web services.



Access to Oracle XML DB Repository Data

There are several ways to access and manipulate data in Oracle XML DB Repository, including using standard protocols such as FTP and HTTP(S)/WebDAV; Oracle XML DB resource Application Program Interfaces (APIs); and the repository views <code>RESOURCE_VIEW</code> and <code>PATH_VIEW</code>.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

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. Although the repository can manage any kind of content, it provides specialized capabilities and optimizations related to managing resources with XML content.

Repository Terminology and Supplied Resources

Oracle XML DB Repository can be thought of as a file system of database objects rather than files. It is a hierarchical set of database objects, across all XML and database schemas, that are mapped to path names.

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.

• 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.

Query-Based Access to Repository Resources

PL/SQL package DBMS_XDB_REPOS provides subprograms that act on Oracle XML DB Repository resources. This API is based on the public views RESOURCE_VIEW and PATH_VIEW, which enable SQL access to repository data through protocols such as FTP and HTTP(S)/WebDAV.

- Servlet Access to Repository Resources Oracle XML DB implements Java Servlet API, version 2.2.
- Operations on Repository Resources
 You can operate on data stored in Oracle XML DB Repository resources using Java, PL/ SQL, and Internet protocols. The most common operations are described, along with the required database permissions to use them.
- Accessing the Content of Repository Resources Using SQL In SQL you can access the content of a document in Oracle XML DB Repository using PL/SQL constructor XDBURIType or using RESOURCE_VIEW and the corresponding resource document.



- Access to the Content of XML Schema-Based Documents
 You can access the content of an XML Schema-based document in the same way as for a
 non-schema-based document: use the corresponding resource document. Or you can
 access it as a row in the default table that was defined when the XML schema was
 registered with Oracle XML DB.
- Update of the Content of Repository Documents You can update the content of documents stored in Oracle XML DB Repository using Internet protocols or SQL.
- Querying Resources in RESOURCE_VIEW and PATH_VIEW Examples here illustrate folder-restricted queries of the repository using RESOURCE_VIEW and PATH VIEW together with Oracle SQL functions equals path and under path.
- Oracle XML DB Hierarchical Repository Index
 Oracle XML DB uses a hierarchical index for Oracle XML DB Repository, to optimize the performance of path-based and folder-restricted queries of the repository. It is implemented as an Oracle domain index.

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. Although the repository can manage any kind of content, it provides specialized capabilities and optimizations related to managing resources with XML content.

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.

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.



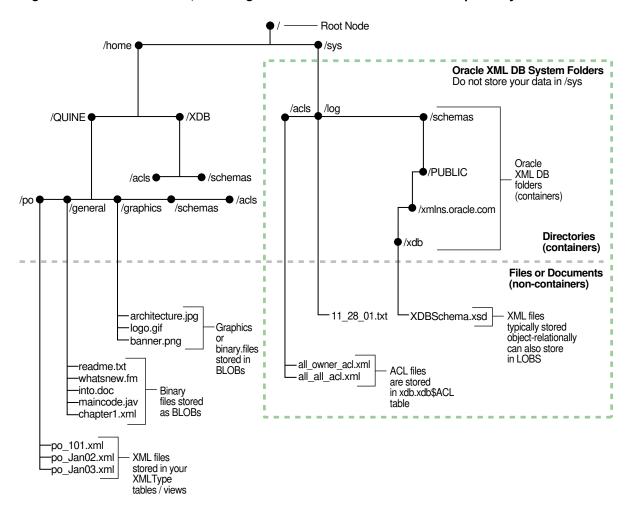


Figure 21-1 A Folder Tree, Showing Hierarchical Structures in the Repository

Caution:

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.

Your applications can access content in Oracle XML DB Repository using standard connectaccess protocols such as FTP, HTTP(S), and WebDAV, in addition to languages SQL, PL/SQL, Java, and C. Oracle XML DB adds native support to Oracle Database for these protocols, which 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 the repository.

The repository gives you direct access to XML content stored in Oracle Database, as if it were stored in a file system. You can set access control privileges on repository files and folders.

These features are available because the repository is modeled on **WebDAV**, an IETF standard that defines a set of extensions to the HTTP protocol. WebDAV 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. A resource has not only content but also associated metadata.

The following topics cover how to access data in Oracle XML DB Repository folders using the standard protocols. They discuss APIs that you can use to access the repository object hierarchy using Java, SQL, and PL/SQL.

Oracle XML DB Provides Name-Level Locking

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 multiple folders. To provide high levels of concurrency, the repository uses name-level locking rather than folder-level locking

- Two Ways to Access Oracle XML DB Repository Resources
 You can access and manipulate Oracle XML DB Repository resources using SQL with special views or by navigating paths using a hierarchical index.
- Database Schema (User Account) XDB and Oracle XML DB Repository Database schema (user account) XDB owns XMLType table XDB\$RESOURCE, which contains all of the resources (files and folders) in Oracle XML DB Repository. It also contains all of the metadata for managing the repository.

Related Topics

Repository Access Using RESOURCE_VIEW and PATH_VIEW

Predefined public views RESOURCE_VIEW and PATH_VIEW provide access to Oracle XML DB repository data. You can use Oracle SQL functions under_path and equals_path to query resources based on their path names, and functions path and depth to return resource path names and depths.

PL/SQL Access to Oracle XML DB Repository

PL/SQL packages DBMS_XDB_CONFIG and DBMS_XDB_REPOS together provide the Oracle XML DB resource application program interface (API) for PL/SQL. You use the former to configure Oracle XML DB and its repository. You use the latter to perform other, non-configuration operations on the repository.

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. You can create, set, and modify access control lists (ACLs).

 Repository Access Using Protocols You can access Oracle XML DB Repository data using protocols FTP and HTTP(S)/ WebDAV.

Oracle XML DB Provides Name-Level Locking

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 multiple folders. To provide high levels of concurrency, the repository uses name-level locking rather than folder-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 systems 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.

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.

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*.

Two Ways to Access Oracle XML DB Repository Resources

You can access and manipulate Oracle XML DB Repository resources using SQL with special views or by navigating paths using a hierarchical index.

- SQL access. 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.
- 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.

See Also:

- Oracle XML DB Repository Access for guidance on selecting an access method
- Table 21-3 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.

Database Schema (User Account) XDB and Oracle XML DB Repository

Database schema (user account) XDB owns XMLType table XDB\$RESOURCE, which contains all of the resources (files and folders) in Oracle XML DB Repository. It also contains all of the *metadata* for managing the repository.

Database schema XDB is created during Oracle XML DB installation. The primary table in this schema is an XMLType table called XDB\$RESOURCE, which contains one row for each resource (file or folder) in Oracle XML DB 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. By default, the XDB schema is dictionary protected, which means that other users cannot use system privileges to modify or tamper with its data. The new functionality also improves the performance of the database as a whole.

The tables owned by database schema (user) XDB are *internal*. Oracle recommends the following:

- Create a *dedicated tablespace* for use only by user XDB, which means also for Oracle XML DB Repository. Ensure that the tablespace is *not* read-only.
- Do not directly manipulate any tables or data owned by user XDB. For example, do not compress or uncompress them.

Use only the PL/SQL subprograms and database views provided by Oracle XML DB to carry out operations on any tables or data owned by user XDB.

• Never unlock user XDB, under any circumstance.

See Also:

- Package DBMS_XDB_ADMIN, for information about creating a dedicated tablespace for user XDB and the repository
- Oracle Database Security Guide

Repository Terminology and Supplied Resources

Oracle XML DB Repository can be thought of as a file system of database objects rather than files. It is a hierarchical set of database objects, across all XML and database schemas, that are mapped to path names.

The repository is a connected, directed, acyclic¹ 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.

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.



Repository Terminology

Some terms that apply to Oracle XML DB Repository include resource, resource name, resource content, folder or directory, path name, path components, link name, access control list (ACL), and XDBBinary element. Some of these terms have common synonyms in other contexts.

Predefined Repository Files and Folders
 Certain files and folders are predefined for Oracle XML DB Repository. You can create additional ones for your own use.

Repository Terminology

Some terms that apply to Oracle XML DB Repository include resource, resource name, resource content, folder or directory, path name, path components, link name, access control list (ACL), and XDBBinary element. Some of these terms have common synonyms in other contexts.

 resource – Any object or node in the repository hierarchy. A resource is identified by a Uniform Resource Locator (URL), which includes the path name and resource name of the object.

See Also:

- Oracle XML DB Repository: Overview
- Oracle XML DB Repository Resources
- 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 (\), 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.
- access control list (ACL) An ordered list of rules that specify access privileges for principals (users or roles) to one or more repository resources.

See Also:

Repository Access Control

 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.



Many terms used by Oracle XML DB have common synonyms in other contexts, as shown in Table 21-1.

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	

Table 21-1 Synonyms for Oracle XML DB Repository Terms

Predefined Repository Files and Folders

Certain files and folders are predefined for Oracle XML DB Repository. You can create additional ones for your own use.

These are the predefined Oracle XML DB Repository files and folders:

```
/dbfs<sup>2</sup>
/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/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/stats.xsd
```

² Repository folder /dbfs gives you protocol access to your DBFS content. See Oracle Database SecureFiles and Large Objects Developer's Guide for information about DBFS. /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.

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.
- How Documents are Stored in Oracle XML DB Repository

When an XML document that is based on a registered XML schema is loaded into the repository, the document is parsed and decomposed according to the schema into a set of SQL objects; and a corresponding resource document is created to provide repository access for the source XML document.

Repository Data Access Control

You can control access to the resources in Oracle XML DB Repository by using access control lists (ACLs), which are composed of access control entries (ACEs). An ACE grants or denies a set of privileges to a specific principal.

Repository 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 which is similar to the directory mechanisms that are used by operating-system file systems. You need certain privileges to resolve a path name.

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.

Related Topics

XDBResource.xsd: XML Schema for Oracle XML DB Resources

A full listing is presented of the Oracle XML DB-supplied XML schema XDBResource.xsd, which is used to represent Oracle XML DB resources.

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 generated tables are assigned by Oracle XML DB. They can be obtained by finding the xdb:defaultTable attribute in your XML schema document (or in the default XML schema document).

How Object-Relational Storage Is Defined for Repository Resources
 You can define object-relational storage for repository resources by subclassing or by storing data that conforms to a registered XML schema.

Oracle ASM Virtual Folder

The contents of the Oracle Automatic Storage Management (Oracle ASM) virtual folder, /sys/asm are Oracle ASM files and folders that are managed automatically by Oracle ASM.

Names of Generated Tables

The names of generated tables are assigned by Oracle XML DB. They 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.

Related Topics

• Default Tables Created during XML Schema Registration

You can create default tables as part of XML schema registration. Default tables are most useful when documents are inserted using APIs and protocols such as FTP and HTTP(S), which do not provide any table specification.

How Object-Relational Storage Is Defined for Repository Resources

You can define object-relational storage for repository resources by subclassing or by storing data that conforms to a registered XML schema.

Applications that need to define object-relational storage for repository resources can do so in either 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.

Related Topics

XML Schema Storage and Query: Basic

XML Schema is a standard for describing the content and structure of XML documents. You can register, update, and delete an XML schema used with Oracle XML DB. You can define storage structures to use for your XML schema-based data and map XML Schema data types to SQL data types.

Oracle ASM Virtual Folder

The contents of the Oracle Automatic Storage Management (Oracle ASM) virtual folder, /sys/asm are Oracle ASM files and folders that are managed automatically by Oracle ASM.



Related Topics

Access to 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, database mirroring, and I/O balancing. Repository access extends to Oracle ASM files, in the *virtual* repository folder /sys/asm. This access is reserved for database administrators (DBAs). It is *not* intended for developers.

See Also:

Oracle Automatic Storage Management Administrator's Guide

How Documents are Stored in Oracle XML DB Repository

When an XML document that is based on a registered XML schema is loaded into the repository, the document is parsed and decomposed according to the schema into a set of SQL objects; and a corresponding resource document is created to provide repository access for the source XML document.

Oracle XML DB provides special handling for XML documents. The rules for storing the contents of an XML Schema-based XML document are defined by its 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 then 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:

- 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 Schemabased 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.



Repository Data Access Control

You can control access to the resources in Oracle XML DB Repository by using access control lists (ACLs), which are composed of access control entries (ACEs). An ACE grants or denies a set of privileges to a specific principal.

A 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 which 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_REPOS.setACL. For more details on Oracle XML DB resource security, see 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.

Repository 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 which is similar to the directory mechanisms that are used by operating-system file systems. You need certain privileges to resolve a path name.

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 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.

Repository Links and Document Links

In addition to containing resources, a folder resource can contain links to other resources (files or folders). These repository links represent hierarchical repository relationships. By contrast, document links are arbitrary links among XML documents that are not necessarily repository resources.

Hard Links and Weak Links

Links that target repository resources can be hard or weak. Hard and weak links have different dependencies with respect to the resources that they target. Hard links cannot target ancestor folders; weak links can. You can query the repository path view, PATH_VIEW, to determine the type of a repository link.

Creating a Weak Link with No Knowledge of Folder Hierarchy

Weak links represent a mapping on top of the repository structure, which is determined by hard links. You can create a weak link to a resource using its OID rather than its path. You can use weak links to access a resource without having access to the folders containing it.

How and When to Prevent Multiple Hard Links

You can restrict the creation of hard links, disallowing multiple hard links to folders or files (or both). 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.

Repository Links and Document Links

In addition to containing resources, a folder resource can contain links to other resources (files or folders). These repository links represent hierarchical repository relationships. By contrast, document links are arbitrary links among XML documents that are not necessarily repository resources.

Repository links are sometimes called **folder links**. They 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 (but XLink support is *deprecated*). 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:

Use of 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 or weak. Hard and weak links have different dependencies with respect to the resources that they target. Hard links cannot target ancestor folders; weak links can. You can query the repository path view, PATH_VIEW, to determine the type of a repository link.

Both **hard links** 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.

💉 See Also:

Oracle Database PL/SQL Packages and Types Reference for information on PL/SQL function getLink



Example 21-1 Querying PATH_VIEW to Determine Link Type

Related Topics

- Deleting Repository Resources: Examples
 Examples here illustrate how to delete Oracle XML DB Repository resources and paths.
- Query-Based Access to Repository Resources
 PL/SQL package DBMS_XDB_REPOS provides subprograms that act on Oracle XML DB
 Repository resources. This API is based on the public views RESOURCE_VIEW and
 PATH_VIEW, which enable SQL access to repository data through protocols such as FTP and HTTP(S)/WebDAV.

Creating a Weak Link with No Knowledge of Folder Hierarchy

Weak links represent a mapping on top of the repository structure, which is determined by hard links. You can create a weak link to a resource using its OID rather than its path. You can use weak links to access a resource without having access to the folders containing it.

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).

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 21-2 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_REPOS.createOIDPath(resoid);
DBMS_OUTPUT.put_line(oidpath);
END;
```

Example 21-3 Creating a Weak Link Using an OID Path

```
CALL DBMS_XDB_REPOS.link(/sys/oid/1BDCB46477B59C20E040578CCE0623D3
'/home/CURRY', 'quinePurchaseOrder.xml',
DBMS_XDB_REPOS.LINK_TYPE_WEAK);
```

How and When to Prevent Multiple Hard Links

You can restrict the creation of hard links, disallowing multiple hard links to folders or files (or both). 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.

Related Topics

Configuration of 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.

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 repository files and folders (resources) the same way you access other files and folders in the file system, as shown in Figure 21-2.
- You can access repository data using HTTP(S)/WebDAV from a Web browser, as shown in Figure 21-3 and Figure 21-4.



Figure 21-3 shows a browser visiting URL http://xdbdemo:8080/. The server it is connected to is xdbdemo, and its HTTP port number is 8080.

Figure 21-4 shows a browser using HTTP to visit an XML document (an XSL stylesheet) stored in the database. The URL is http://localhost:8080/home/SCOTT/poSource/xsl purchaseOrder.xsl.

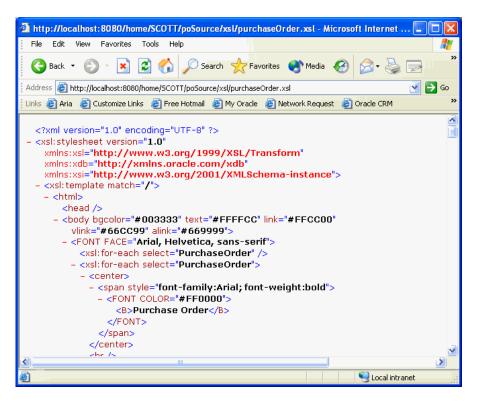
Back ▼ → > È		Internet Address	• @G
lers × Desktop Angel Ang	Name 🛆	Internet Address	1 60
Average Averag			Type
My Documents		,	
	 ALLEN-2001101709512122PDT.xml ALLEN-2001101709512145PDT.xml BLAKE-2001101709512145PDT.xml BLAKE-2001101709512151PDT.xml FORD-2001101709512138PDT.xml FORD-2001101709512138PDT.xml KING-2001101709512138PDT.xml KING-2001101709512138PDT.xml MARTIN-2001101709512138PDT.xml MARTIN-2001101709512138PDT.xml MARTIN-2001101709512138PDT.xml MARTIN-200110170951213PDT.xml MARTIN-2001101709512138PDT.xml MARTIN-200110170951214PDT.xml MARTIN-200110170951214PDT.xml MARTIN-200110170951214PDT.xml MARTIN-200110170951214PDT.xml MARTIN-200110170951214PDT.xml SCOTT-200110170951214PDT.xml SCOTT-2001101709512124PDT.xml SMITH-200110170951214PDT.xml WARD-200110170951214PDT.xml 	http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/ALLEN-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/ALLEN-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/BLAKE-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/BLAKE-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/BCRD-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/BCRD-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/BCRD-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/BCRD-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/BCRD-20011017. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-200110. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-200110. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-200110. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-200110. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-200110. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-200110. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-200110. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-200110. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARTIN-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARE-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARE-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARE-2001101. http://mdrake-sun:8080/home/SCOTT/purchaseOrders/1999/Apr/MARE-2001	 XML File
⊕ sys ⊕- @ spannala-sun			

Figure 21-2 Oracle XML DB Folders in Windows Explorer

🖉 http://xdbdemo:80	380 - Microsoft Internet Explorer		<u>_ ×</u>
File Edit View Favorite	es Tools Help		
🗢 Back 🔹 🤿 🖉 🕼	📸 🛛 🐼 Search 🛛 🐼 Favorites 🛞 Media 🏼 🅉 🗍	1- 4 💽 - E	. E <u>Q</u>
Address 🔊 http://xdbdemo:	8080/	-	🖓 Go 🛛 Links 🌺
			<u> </u>
Index of /			
Name	Last modified	Size	
home/	Sat, 16 Feb 2002 17:58:53 GMT	_	
public/	Thu, 14 Feb 2002 17:09:24 GMT	-	
<u>sys/</u>	Thu, 14 Feb 2002 17:09:24 GMT	-	
<u>xdbconfig.xml</u>	Thu, 14 Feb 2002 17:10:08 GMT	0	
			_
🛃 Done		📔 🔤 🔂 Local	intranet //

Figure 21-3 Accessing Repository Data Using HTTP(S)/WebDAV and a Web Browser

Figure 21-4 Path-Based Access Using HTTP and a URL



Access to 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.

Access to 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, database mirroring, and I/O balancing. Repository
access extends to Oracle ASM files, in the virtual repository folder /sys/asm. This access
is reserved for database administrators (DBAs). It is not intended for developers.

Access to 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 certain scenarios.
- Overview of Protocol Access to Oracle XML DB Protocol access to Oracle XML DB involves connecting, authenticating a user, parsing the request, and perhaps invoking a Java servlet.
- Retrieval of Oracle XML DB Resources When a protocol indicates that a resource is to be retrieved, the path name to the resource is resolved and the resource is fetched, by streaming it as either XML data or in RAW form.
- Storage of Oracle XML DB Resources
 When a protocol indicates that a resource that is an XML document is to be stored, its associated XML schema, if any, is consulted to determine the default table in which to store the document.
- 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.

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 certain 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

Overview of Protocol Access to Oracle XML DB

Protocol access to Oracle XML DB involves connecting, authenticating a user, parsing the request, and perhaps invoking a Java servlet.



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.
- 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.
- (HTTP only) If the requested path name maps to a servlet, then the servlet is invoked using Java Virtual Machine (JVM). The servlet code writes the response to a response stream or asks XMLType instances to do so.

Retrieval of Oracle XML DB Resources

When a protocol indicates that a resource is to be retrieved, the path name to the resource is resolved and the resource is fetched, by streaming it as either XML data or in RAW form.

Resources being fetched are streamed as XML data, except for those containing element XDBBinary, which is the XML binary data type, which have their contents streamed out in RAW form.

Storage of Oracle XML DB Resources

When a protocol indicates that a resource that is an XML document is to be stored, its associated XML schema, if any, is consulted to determine the default table in which to store the document.

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.

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.



Related Topics

 Repository Access Using Protocols You can access Oracle XML DB Repository data using protocols FTP and HTTP(S)/ WebDAV.

Access to 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, database mirroring, and I/O balancing. Repository access extends to Oracle ASM files, in the *virtual* repository folder /sys/asm. 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.

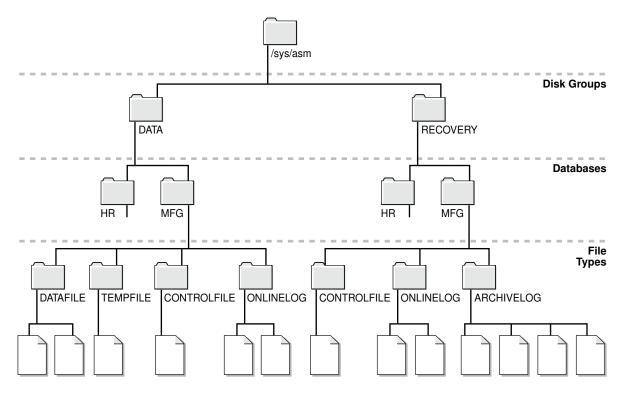
Virtual folder /sys/asm is created by default during Oracle XML DB installation. If the database is not configured to use Oracle ASM, the folder is empty and no operations are permitted on it.

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 21-5, which omits directories created for aliases, for simplicity.





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 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

PL/SQL package DBMS_XDB_REPOS provides subprograms that act on Oracle XML DB Repository resources. This API is based on the public views RESOURCE_VIEW and PATH_VIEW, which enable SQL access to repository data through protocols such as FTP and HTTP(S)/ WebDAV.

- PATH VIEW Has one row for each unique repository path
- RESOURCE VIEW Has one row for each resource

Through these views, you can access and update both the metadata and the content of documents stored in the repository. Operations on the views use underlying repository tables such as XDB\$RESOURCE.

Each view contains virtual column RES. You use column RES to access and update resource documents using SQL statements that accept a repository path notation.

View RESOURCE_VIEW contains column ANY_PATH. Column ANY_PATH contains a valid URL that the current user can pass to PL/SQL constructor XDBURIType to access the resource content. If this content is not binary data, then the resource itself also contains the content.

Table 21-2 summarizes the differences between the views.

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 views are of data type 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 code 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.

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.



Related Topics

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.

Repository Access Using RESOURCE_VIEW and PATH_VIEW

Predefined public views RESOURCE_VIEW and PATH_VIEW provide access to Oracle XML DB repository data. You can use Oracle SQL functions under_path and equals_path to query resources based on their path names, and functions path and depth to return resource path names and depths.

See Also:

- Repository Access Control
- Oracle Database Reference for more information about view PATH VIEW
- Oracle Database Reference for more information about view RESOURCE VIEW

Servlet Access to Repository Resources

Oracle XML DB implements Java Servlet API, version 2.2.

Support is limited by these restrictions:

- 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 XSLT stylesheet 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.

Related Topics

Guidelines for Oracle XML DB Applications in Java

Design guidelines are presented for writing Oracle XML DB applications in Java. This includes guidelines for writing and configuring Java servlets for Oracle XML DB.

Operations on Repository Resources

You can operate on data stored in Oracle XML DB Repository resources using Java, PL/SQL, and Internet protocols. The most common operations are described, along with the required database permissions to use them.

You can access repository data in any of these ways:

- 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

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.

Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required
Create		HTTP:PUT;	DAV::bind on
resource	<pre>DBMS_XDB_REPOS.createResource('/public/T1/testcase.txt', 'ORIGINAL text'); INSERT INTO RESOURCE_VIEW (ANY_PATH, RES) SELECT '/public/T1/copy1.txt', RES FROM RESOURCE_VIEW WHERE equals_path(RES,</pre>	FTP: PUT	parent folder
Update		HTTP: PUT;	xdb:write-
resource contents	<pre>UPDATE RESOURCE_VIEW SET RES = XMLQuery('declare default element namespace "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :) copy \$i := \$p1 modify (for \$j in \$i/Resource/Contents/text return replace value of node \$j with \$p2) return \$i' PASSING RES AS "p1", 'NEW text' AS "p2" RETURNING CONTENT) WHERE equals_path(RES, '/public/T1/copy1.txt') = 1</pre>	FTP: PUT	content ON resource
Update resource properties	<pre>UPDATE RESOURCE_VIEW SET RES = XMLQuery('declare default element namespace "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :) copy \$i := \$p1 modify (for \$j in \$i/Resource/DisplayName return replace value of node \$j with \$p2) return \$i' PASSING RES AS "p1", 'NewName1.txt' AS "p2" RETURNING CONTENT) WHERE equals_path(RES, '/public/T1/copy1.txt') = 1;</pre>	WebDAV: PROPPATCH;	DAV::write- properties on resource

Table 21-3 Accessing Oracle XML DB Repository: API Options



Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required
Update resource ACL	<pre>EXEC DBMS_XDB_REPOS.setACL('/public/T1/copy1.txt', '/sys/acls/all_owner_acl.xml');</pre>	not applicable	DAV::write- acl ON resource
Unlink resource (delete if last link)	<pre>EXEC DBMS_XDB_REPOS.deleteResource() Or DELETE FROM RESOURCE_VIEW WHERE equals_path(RES, path) > 0</pre>	HTTP: DELETE; FTP:delete	DAV::unbind on parent folder xdb:unlink- from on resource
Forcibly remove all links to resource	<pre>DBMS_XDB_REPOS.deleteResource() or DELETE FROM PATH_VIEW WHERE XMLCast(XMLQuery('declare namespace n1= 'http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :) //n1:DisplayName' PASSING RES RETURNING CONTENT) AS VARCHAR2(256)) = 'My resource'</pre>	FTP: quote rm_rf resource	DAV::unbind on all parent folders xdb:unlink- from ON resource
Move resource	<pre>UPDATE PATH_VIEW SET path = '/public/T1/copy2.txt' WHERE equals_path(RES,</pre>	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

Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required
Copy resource	INSERT INTO PATH_VIEW (path, RES, link) SELECT '/public/T1/copy3.txt', RES, link FROM PATH_VIEW	WebDAV: COPY;	Copy to new: DAV::bind on target parent folder
	<pre>WHERE equals_path(RES, '/public/T1/copy2.txt')</pre>		DAV::read on resource
			Copy to existing (replacement):
			DAV::read on resource DAV::write- properties and DAV::write- content on existing target resource
Create hard link to existing resource	<pre>EXEC DBMS_XDB_REPOS.link('/public/T1/copy3.txt',</pre>	not applicable	DAV::bind on parent folder xdb:link-to on resource
Create weak link to existing resource	<pre>D EXEC DBMS_XDB_REPOS.link('/public/T1/copy3.txt', '/public/T1', 'myweaklink', DBMS_XDB_REPOS.LINK_TYPE_WEAK);</pre>	not applicable	DAV::bind on parent folder xdb:link-to on resource
Change owner of resource	<pre>UPDATE RESOURCE_VIEW SET RES = XMLQuery('copy \$i := \$p1 modify (for \$j in \$i/Resource/Owner return replace value of node \$j with \$p2) return \$i' PASSING RES AS "p1", 'U2' AS "p2" RETURNING CONTENT)</pre>	not applicable	DAV::take- ownership on resource

Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required
Get binary or text representat	SELECT XDBURIType(path).getBLOB() FROM DUAL;	HTTP:GET; FTP:get	xdb:read- contents ON resource
ion of resource contents	<pre>SELECT XMLQuery('declare default element namespace "http://xmlns.oracle.com/xdb/XDBResource.xsd";(: :) \$r/Resource/Contents' PASSING RES AS "r" RETURNING CONTENT) FROM RESOURCE_VIEW WHERE equals_path(RES, '/public/T1/copy2.text') = 1;</pre>		
Get XMLType representat ion of resource	<pre>SELECT XDBURIType('/public/T1/res.xml').getXML FROM DUAL;</pre>	not applicable	xdb:read- contents ON resource
contents	<pre>SELECT XMLQuery('declare default element namespace "http://xmlns.oracle.com/xdb/XDBResource.xsd";(: :) \$r/Resource/Contents/*' PASSING RES AS "r" RETURNING CONTENT) FROM RESOURCE_VIEW WHERE equals_path(RES, '/public/T1/res.xml') = 1;</pre>		
Get resource properties	<pre>SELECT XMLCast(XMLQuery('declare default element namespace "http://xmlns.oracle.com/xdb/XDBResource.xsd";(: :) \$r/Resource/LastModifier' PASSING RES AS "r" RETURNING CONTENT) AS VARCHAR2(128))FROM RESOURCE_VIEW WHERE equals_path(RES, '/public/T1/res.xml') = 1;</pre>	<pre>WebDAV: PROPFIND (depth = 0);</pre>	xdb:read- properties on resource
List resources in folder	<pre>SELECT PATH FROM PATH_VIEW WHERE under_path(res, '/public/T1') = 1;</pre>	WebDAV: PROPFIND (depth = 0);	xdb:read- contents ON folder
Create folder	Call DBMS_XDB_REPOS.createFolder('/public/T2');	WebDAV: MKCOL; FTP:mkdir	DAV::bind on parent folder
Unlink empty folder	DBMS_XDB_REPOS.deleteResource('/public/T2')	HTTP: DELETE; FTP:rmdir	DAV::unbind on parent folder xdb:unlink- from on resource



_			_
Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required
Forcibly delete folder and all links to it	DBMS_XDB_REPOS.deleteResource('/public/T2', DBMS_XDB.DELETE_RECURSIVE_FORCE);	not applicable	DAV::unbind on all parent folders xdb:unlink- from on folder resource
Get resource with a row lock	SELECT FROM RESOURCE_VIEW FOR UPDATE;	not applicable	<pre>xdb:read- properties and xdb:read- contents ON resource</pre>
Add WebDAV lock on resource	<pre>EXEC DBMS_XDB_REPOS.LockResource('/public/T1/res.xml', TRUE, TRUE);</pre>	WebDAV: LOCK; FTP: quote lock	DAV::write- properties on resource
Remove WebDAV lock	<pre>DECLARE BEGIN DBMS_XDB_REPOS.GetLockToken('/public/T1/res.xml',</pre>	WebDAV: UNLOCK; FTP: quote unlock	DAV::write- properties and DAV::unlock on resource
Check out file resource	<pre>EXEC DBMS_XDB_VERSION.checkOut('/public/T1/res.xml');</pre>	not applicable	DAV::write- properties on resource
Check in file resource	<pre>EXEC DBMS_XDB_VERSION.checkIn('/public/T1/res.xml');</pre>	not applicable	DAV::write- properties on resource
Uncheck out file resource	EXEC DBMS_XDB_VERSION.unCheckOut('/public/T1/res.xml');	not applicable	DAV::write- properties on resource
Make file resource versioned	<pre>EXEC DBMS_XDB_VERSION.makeVersioned('/public/T1/res.xml');</pre>	not applicable	DAV::write- properties on resource



Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required
Remove an	DBMS_XEVENT.remove	not	xdb:write-
event handler	The use of repository events to trigger application actions is deprecated in Oracle Database 21c (21.3). There is no replacement.	applicable	config on resource or parent folder (depending on the context)
Commit changes	COMMIT;	Automatic commit after each request	not applicable
Rollback changes	ROLLBACK;	not applicable	not applicable

In addition to the privileges listed in Table 21-3, 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.

Related Topics

Repository Access Using RESOURCE_VIEW and PATH_VIEW

Predefined public views RESOURCE_VIEW and PATH_VIEW provide access to Oracle XML DB repository data. You can use Oracle SQL functions under_path and equals_path to query resources based on their path names, and functions path and depth to return resource path names and depths.

PL/SQL Access to Oracle XML DB Repository

PL/SQL packages DBMS_XDB_CONFIG and DBMS_XDB_REPOS together provide the Oracle XML DB resource application program interface (API) for PL/SQL. You use the former to configure Oracle XML DB and its repository. You use the latter to perform other, non-configuration operations on the repository.

 Repository Access Using Protocols You can access Oracle XML DB Repository data using protocols FTP and HTTP(S)/ WebDAV.

See Also:

- Oracle Database PL/SQL Packages and Types Reference for information about PL/SQL package DBMS_XDB_REPOS
- Oracle Database PL/SQL Packages and Types Reference for information about PL/SQL package DBMS_XDB_VERSION

Accessing the Content of Repository Resources Using SQL

In SQL you can access the content of a document in Oracle XML DB Repository using PL/SQL constructor XDBURIType or using RESOURCE VIEW and the corresponding resource document.

The easiest way is to use <code>XDBURIType</code>. You pass a URL to this constructor to specify which resource to access. The URL is assumed to start at the root of the repository. Object type <code>XDBURIType</code> provides methods <code>getBLOB()</code>, <code>getCLOB()</code>, and <code>getXML()</code>, to access the different kinds of content that can be associated with a resource.

Example 21-4 uses constructor XDBURITYPE to access the content of a text document.

The content of a document can also be accessed using RESOURCE_VIEW and the corresponding resource document. Example 21-5 does this to access the content of a text document.

The content of XML documents (XML Schema-based or non-schema-based) can also be accessed this way. Example 21-6 uses an XPath expression that includes nodes from an XML document and nodes from the corresponding resource document to access the contents of a PurchaseOrder document.

In Example 21-6, 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:

XQuery and Oracle XML DB for more information about the XMLNAMESPACES clause of XMLTable

Example 21-4 Accessing a Text Document in the Repository Using XDBURITYPE

SELECT XDBURIType('/home/QUINE/NurseryRhyme.txt').getCLOB() FROM DUAL;

1 row selected.

Example 21-5 Accessing Resource Content Using RESOURCE_VIEW

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.

Example 21-6 Accessing XML Documents Using Resource and Namespace Prefixes

3 rows selected.

Access to the Content of XML Schema-Based Documents

You can access the content of an XML Schema-based document in the same way as for a non-schema-based document: use the corresponding resource document. Or you can access it as a row in the default table that was defined when the XML schema was registered with Oracle XML DB.

In the first case, you can use RESOURCE_VIEW to query different types of XML Schema-based documents with a single SQL statement.

Accessing Resource Content Using Element XMLRef in Joins

Element XMLRef in a 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.

Accessing Resource Content Using Element XMLRef in Joins

Element XMLRef in a 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.

Examples here show queries that access resource content using joins based on the value of element XMLRef.

Example 21-7 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 21-8 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.

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.

Example 21-7 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
```

A Night to Remember The Unbearable Lightness Of Being The Wizard of Oz

3 rows selected.

Example 21-8 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.

Update of the Content of Repository Documents

You can update the content of documents stored in Oracle XML DB Repository using Internet protocols or SQL.

Update of Repository Content Using Internet Protocols

The most popular content authoring tools support HTTP, FTP, and WebDAV protocols. Given appropriate access permissions, a simple URL targeting a document to update is all you need, to access and edit content stored in Oracle XML DB Repository.



Update of Repository Content Using SQL

You can use XQuery Update 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.

Update of Repository Content Using Internet Protocols

The most popular content authoring tools support HTTP, FTP, and WebDAV protocols. Given appropriate access permissions, a simple URL targeting a document to update is all you need, to access and edit content stored in Oracle XML DB Repository.

Popular content authoring tools can use HTTP verb get to access the content of a document, given a URL to it, and they can use HTTP verb put to save the updated content.

Figure 21-6 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 21-6 Updating and Editing Content Stored in Oracle XML DB Using Microsoft Word

🖥 SBELL-2003030912333601PDT.xml - Microsoft Office Word 2003 Beta	
Eile Edit View Insert Format Iools Table Window Help Type a question for help	• ×
🗄 🗋 😂 🔚 😂 🔄 🗳 🚉 🐇 🐚 隆 🏈 🤊 • 🔍 - 🧶 🞲 📰 🔣 💷 🐺 📰 🖓 🧔 ¶ 98% 💿 🤗 💷 Read 🥊	
44 Normal + Left: ▼ Times New Roman ▼ 12 ▼ B I U 臣 吾 吾 〓 閨 篇 ▼ 臣 臣 章 律 田 ▼ 🖉 ▼ 🛓 ▼	
🛃 🛃 👗 🐚 瀺 Times New Roman 💌 12 🔹 📥 ▾ B 🖌 U (三) 三 三 三 三 章 章 4 🚼 📮	
■ · Z · · · · · · · · · · · · · · · · ·	<u>م</u> ا
("Reference(SBELL-2003030912333601PDT)Reference")	
("Actions (
(«Action((«User(SVOLLMAN)User»)	
Action (*) Actions (*)	
("Reject()Reject")	
("Requestor (Sarah J. Bell) Requestor ")	
(«User (K CHUNG)User») Saving as:	
("CostCenter (S30) CostCenter ") 11/2 (S30) CostCenter ") 03030912333601PDT.xml	
("ShippingInstructions Cancel	
(Iname Sarah J. Bell) name)	
(*address 400 Oracle Parkway Redwood Shores CA 94065 USA)address)	
("telephone (650 506 7400) telephone)	
ShippingInstructions (*)	
(*SpecialInstructions Saturday Delivery)SpecialInstructions *	
(«LineItems	•
("LineItem	Ô
= G B © Q A Night to Demonstration Description ₽	Ŧ
Page Sec At Ln Col REC TRK EXT OVR English (U.S 🛄 🖌	

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.



Update of Repository Content Using SQL

You can use XQuery Update 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.

- Updating a Document in the Repository by Updating Its Resource Document You can update the content of a document using a SQL UPDATE statement and SQL function XMLQuery with XQuery Update. An XQuery expression is passed to XMLQuery as the target of the update operation.
- Updating an XML Schema-Based Document in the Repository by Updating the Default Table

You can update XML Schema-based documents by performing an update operation directly on the default table that is used to manage the content of the document.

Updating a Document in the Repository by Updating Its Resource Document

You can update the content of a document using a SQL UPDATE statement and SQL function XMLQuery with XQuery Update. An XQuery expression is passed to XMLQuery as the target of the update operation.

Example 21-9 updates the content of a simple text document. The XQuery expression passed to XMLQuery as the target of the update operation identifies the text node as belonging to element /Resource/Contents/text.

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 21-10 updates a node in an XML document by performing a SQL UPDATE operation on the corresponding resource document. Here, XQuery Update is used to change the value of the text node associated with element User.

Example 21-9 Updating a Text Document Using UPDATE and XQuery Update 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);
```

Example 21-10 Updating an XML Node Using UPDATE and XQuery Update on the Resource

```
UPDATE RESOURCE VIEW
  SET RES =
   XMLQuery('declare namespace r="http://xmlns.oracle.com/xdb/XDBResource.xsd";
              copy $i := $p1 modify
                (for $j in $i/r:Resource/r:Contents/PurchaseOrder/User
                 return replace value of node $j with $p2)
              return $i'
             PASSING RES AS "p1", 'SKING' AS "p2" RETURNING CONTENT)
    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 an XML Schema-Based Document in the Repository by Updating the Default Table

You can update XML Schema-based documents by performing an 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 SQL UPDATE statement must use a join between the resource and the default table.

In general, when updating the content of XML Schema-based 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 21-11, XQuery Update is used 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 Oracle SQL function equals_path. This limits the update to the row corresponding to the resource identified by the specified path.

Example 21-11 Updating XML Schema-Based Documents in the Repository

```
UPDATE purchaseorder p
  SET p.OBJECT VALUE =
   XMLQuery('copy $i := $p1 modify
               (for $j in $i/PurchaseOrder/User
                return replace value of node $j with $p2)
             return $i'
             PASSING p.OBJECT VALUE AS "p1", 'SBELL' AS "p2" RETURNING CONTENT)
   WHERE ref(p) =
      (SELECT XMLCast(XMLQuery('declare default element namespace
                                  "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                                fn:data(/Resource/XMLRef)'
                               PASSING rv.RES RETURNING CONTENT) AS REF XMLType)
         FROM RESOURCE VIEW rv
         WHERE
           equals_path(
             rv.RES,
             '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
           = 1);
SELECT XMLCast(XMLQuery('$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

Querying Resources in RESOURCE_VIEW and PATH_VIEW

Examples here illustrate folder-restricted queries of the repository using RESOURCE_VIEW and PATH VIEW together with Oracle SQL functions equals path and under path.

Oracle XML DB provides two Oracle SQL functions, <code>equals_path</code> and <code>under_path</code>, that you can use to perform folder-restricted queries. Such queries limit SQL statements that operate on the <code>RESOURCE_VIEW</code> or <code>PATH_VIEW</code> 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.

Examples here demonstrate simple folder-restricted queries against resource documents stored in RESOURCE VIEW and PATH VIEW.

The query in Example 21-12 uses SQL function <code>equals_path</code> and <code>RESOURCE_VIEW</code> to access a resource. The resource queried is that which results from the update operation of Example 21-9: the original resource text shown in Example 21-4 and Example 21-5 has been replaced by a different nursery rhyme, "Hickory Dickory Dock..."

As Example 21-12 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 21-13 finds a path to each of the XSL stylesheets stored in Oracle XML DB Repository. It performs a search based on the DisplayName ending in .xsl.

The query in Example 21-14 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.

The query in Example 21-15 lists the contents of the folder identified by path /home/QUINE/ PurchaseOrders/2002/Apr. This is effectively a directory listing of the folder.

The query in Example 21-16 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.

The query in Example 21-17 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 21-12 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>OUINE</Owner>
  <Creator>QUINE</Creator>
  <LastModifier>OUINE</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.

Example 21-13 Determining the Path to XSLT Stylesheets 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

/home/MDSYS/epsg/sdoepsggrid2nadcon.xsl /home/MDSYS/epsg/sdoepsggrid2ntv2/xsl /source/schemas/poSource/xsl/empdept.xsl /source/schemas/poSource/xsl/purchaseOrder.xsl

4 rows selected.



Example 21-14 Counting Resources Under a Path

Example 21-15 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-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/SBELL-20021009123336331PDT.xml /home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336321PDT.xml /home/QUINE/PurchaseOrders/2002/Apr/SKING-20021009123336321PDT.xml /home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml /home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml /home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml /home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml

11 rows selected.

Example 21-16 Listing the Links Contained in a Folder

5 rows selected.

Example 21-17 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

Oracle XML DB uses a hierarchical index for Oracle XML DB Repository, to optimize the performance of path-based and folder-restricted queries of the repository. It is implemented as an Oracle domain 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 can become inefficient as the number of documents and depth of the folder hierarchy increases.

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 21-18 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 21-18 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

I	d	Operation	Na	me		Rows		Bytes		Cost	(%CPU)	Time
1	0	SELECT STATEMENT			L	17		3111		34	(6)	00:00:01
	1	NESTED LOOPS			L	17		3111	L	34	(6)	00:00:01
	2	NESTED LOOPS			L	17	T	2822		34	(6)	00:00:01
	3	NESTED LOOPS			L	466		63842		34	(6)	00:00:01
*	4	TABLE ACCESS BY INDEX ROWID	XD	B\$RESOURCE	L	1		135		3	(0)	00:00:01
*	5	DOMAIN INDEX	XD	BHI_IDX								
	6	COLLECTION ITERATOR PICKLER FETCH		_	L							
*	7	INDEX UNIQUE SCAN	XD	B PK H LINK	L	1		28	L	0	(0)	00:00:01
*	8	INDEX UNIQUE SCAN	SY	s C003900	L	1		17	L	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.

Configuration of Oracle XML DB Repository

Overall configuration of Oracle XML DB Repository applies to all repository resources. It does not include configuring parameters for handling events or managing XLink and XInclude processing. You use resource configuration files to configure resources.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

Resource Configuration Files

You configure an 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 itself a resource in Oracle XML DB Repository.

- Configuring a Resource
 Configuring an Oracle XML DB Repository resource involves creating a configuration file, adding that file to the repository as a configuration resource, and mapping the configuration resource to the resources it configures or to the entire repository.
- Common Configuration Parameters Commonly used configuration parameters are described, that is, elements in a configuration file.

Related Topics

Oracle XML DB Protocol Server Configuration Management

Oracle XML DB protocol server uses configuration parameters stored in file xdbconfig.xml to initialize its startup state and manage session level configuration. 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.

- Configuration of Repository Events In a resource configuration file, you use element event-listeners, child of element ResConfig, to configure Oracle XML DB Repository event handling.
- Configuration of Repository Resources for XLink and XInclude The resource configuration file that you use as a resource to configure XLink (*deprecated*) and XInclude processing for other resources is described.
- XDBResource.xsd: XML Schema for Oracle XML DB Resources A full listing is presented of the Oracle XML DB-supplied XML schema XDBResource.xsd, which is used to represent Oracle XML DB resources.
- XDBResConfig.xsd: XML Schema for Resource Configuration
 A full listing is presented of the Oracle XML DB-supplied XML schema used to configure
 repository resources. It is accessible in Oracle XML DB Repository at path /sys/schemas/
 PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd.



See Also:

Package DBMS_XDB_ADMIN, for information about using a dedicated tablespace for the repository

Resource Configuration Files

You configure an 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 itself a resource in Oracle XML DB Repository.

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.

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.

Related Topics

Configuration Elements defaultChildConfig and configuration

Configuration element defaultChildConfig applies to only folders. It holds configuration information 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.

Configuring a Resource

Configuring an Oracle XML DB Repository resource involves creating a configuration file, adding that file to the repository as a configuration resource, and mapping the configuration resource to the resources it configures or to the entire repository.

Follow these steps to configure an individual 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.
- 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_REPOS.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.

Related Topics

XDBResConfig.xsd: XML Schema for Resource Configuration
 A full listing is presented of the Oracle XML DB-supplied XML schema used to configure
 repository resources. It is accessible in Oracle XML DB Repository at path /sys/schemas/
 PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd.

See Also:

- Example 22-1 for an example of a simple resource configuration file
- Oracle Database PL/SQL Packages and Types Reference for information about package DBMS_RESCONFIG

Common Configuration Parameters

Commonly used configuration parameters are described, 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. You can use it to disable or enable the resource configuration file.
- Configuration Elements defaultChildConfig and configuration Configuration element defaultChildConfig applies to only folders. It holds configuration information 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.
- Configuration Element applicationData
 Element applicationData stores application-specific data. An application typically passes this data to an event handler when the handler is run. You can use any XML content you want inside element applicationData.

Configuration Element ResConfig

The top-level element of a resource configuration file is ResConfig. You can use it to disable or enable the resource configuration file.

Besides attributes namespace and schemaLocation, element ResConfig 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 Elements defaultChildConfig and configuration

Configuration element defaultChildConfig applies to only folders. It holds configuration information 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



```
<pre-condition>
        <existsNode>
            <XPath>/Resource[ContentType="xml"]</XPath>
        </existsNode>
        </pre-condition>
        <path>/cm/app_rc.xml<path>
        </configuration>
        </defaultChildConfig>
</ResConfig>
```

Configuration Element applicationData

Element applicationData stores application-specific data. An application typically passes this data to an event handler when the handler is run. You can use any XML content 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.

The use of repository events to trigger application actions is deprecated in Oracle Database 21c (21.3). There is no replacement.

Example 22-2 shows an applicationData element for use with an Oracle Spatial and Graph application.

🖍 See Also:

- Oracle Database XML Java API Reference, class XDBRepositoryEvent for information about Java function oracle.xdb.XMLType.getApplicationData
- Example 30-1 for an example of a resource configuration file for event listeners

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>
```



23

Use of XLink and XInclude with Oracle XML DB

You can use XLink and XInclude with resources in Oracle XML DB Repository. But the use of XLink is *deprecated*.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

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.

Link Types for XLink and XInclude

XLink and XInclude link types are described, as well as the relation between these and Oracle XML DB Repository links. XLink links (*deprecated*) are more general than repository links. XLink links can be simple or extended. Oracle XML DB supports only simple, not extended, XLink links.

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.

Oracle XML DB Support for XLink

You can configure Oracle XML DB Repository resources so that XLink links are ignored, or so that they are mapped to Oracle XML DB document links. However, the use of XLink with Oracle XML DB Repository is *deprecated*.

• Oracle XML DB Support for XInclude

Oracle XML DB supports XInclude 1.0 as the standard mechanism for managing compound documents. It does not support attribute <code>xpointer</code> and the inclusion of document fragments, however. Only complete documents can be included (using attribute <code>href</code>).

Use View DOCUMENT_LINKS to Examine XLink and XInclude Links

You can query the read-only public view DOCUMENT_LINKS to obtain system information about document links derived from both XLink (*deprecated*) and XInclude links. The information in this view includes the following columns, for each link:

- Configuration of Repository Resources for XLink and XInclude The resource configuration file that you use as a resource to configure XLink (*deprecated*) and XInclude processing for other resources is described.
- Manage XLink and XInclude Links Using DBMS_XDB_REPOS.processLinks You can use PL/SQL procedure DBMS_XDB_REPOS.processLinks to manually process all XLink (*deprecated*) and XInclude links in a single document or in all documents of a folder.



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.

Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

See Also:

- XML Linking Language (XLink) Version 1.0
- XML Inclusions (XInclude) Version 1.0 (Second Edition)



Link Types for XLink and XInclude

XLink and XInclude link types are described, as well as the relation between these and Oracle XML DB Repository links. XLink links (*deprecated*) are more general than repository links. XLink links can be simple or extended. Oracle XML DB supports only simple, not extended, XLink links.

• XLink and XInclude Links Model Document Relationships

XLink (*deprecated*) 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.

• XLink Link Types and XInclude Link Types XLink (*deprecated*) 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 and XInclude Links Model Document Relationships

XLink (*deprecated*) 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.

Related Topics

Hard Links and Weak Links

Links that target repository resources can be hard or weak. Hard and weak links have different dependencies with respect to the resources that they target. Hard links cannot target ancestor folders; weak links can. You can query the repository path view, <code>PATH_VIEW</code>, to determine the type of a repository link.

XLink Link Types and XInclude Link Types

XLink (*deprecated*) 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.

Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

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 contentmanagement 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.

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.

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.

Oracle XML DB Support for XLink

You can configure Oracle XML DB Repository resources so that XLink links are ignored, or so that they are mapped to Oracle XML DB document links. However, the use of XLink with Oracle XML DB Repository is *deprecated*.

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.

If you configure resources so that XLink links are mapped to Oracle XML DB document links, you 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.



Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

Related Topics

- Use View DOCUMENT_LINKS to Examine XLink and XInclude Links You can query the read-only public view DOCUMENT_LINKS to obtain system information about document links derived from both XLink (*deprecated*) and XInclude links. The information in this view includes the following columns, for each link:
- User-Defined Repository Metadata You can create your own metadata to associate with XML data stored in Oracle XML DB Repository.
- Hard Links and Weak Links

Links that target repository resources can be hard or weak. Hard and weak links have different dependencies with respect to the resources that they target. Hard links cannot target ancestor folders; weak links can. You can query the repository path view, PATH_VIEW, to determine the type of a repository link.

- Configuration of Repository Resources for XLink and XInclude The resource configuration file that you use as a resource to configure XLink (*deprecated*) and XInclude processing for other resources is described.
- XDBResConfig.xsd: XML Schema for Resource Configuration
 A full listing is presented of the Oracle XML DB-supplied XML schema used to configure
 repository resources. It is accessible in Oracle XML DB Repository at path /sys/schemas/
 PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd.

Oracle XML DB Support for XInclude

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.

• Expanding Compound-Document Inclusions You can optionally expand xi:include elements with their targets when you retrieve a compound document from Oracle XML DB Repository.

Validation of 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.



• Update of a Compound Document

You can update a compound document just as you would update any resource. This replaces the resource with a new value.

 Compound Document Versioning, Locking, and Access Control The components of a compound document are separate resources. They are versioned and locked independently, and their access is controlled independently.

Related Topics

• Use View DOCUMENT_LINKS to Examine XLink and XInclude Links You can query the read-only public view DOCUMENT_LINKS to obtain system information about document links derived from both XLink (*deprecated*) and XInclude links. The information in this view includes the following columns, for each link:

Expanding Compound-Document Inclusions

You can optionally expand xi:include elements with their targets when you retrieve a compound document from Oracle XML DB Repository.

When you retrieve a compound document 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 second argument of constructor XDBURIType:

- 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:

(The result shown here corresponds to the resource <code>bookfile.xml</code> shown in Example 23-8, together with its included resources, <code>chap1.xml</code> and <code>chap2.xml</code>.)

See Also:

- Compound Document Versioning, Locking, and Access Control for information
 about access control during expansion
- Oracle Database PL/SQL Packages and Types Reference for more information about 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>
  </Bodv>
</Chapter>
<Chapter id="2">
 <Title>Conclusion</Title>
  <Body>
   <Para>xyz xyz</Para>
   <Para>abc abc</Para>
  </Body>
</Chapter>
3 rows selected.
```

Example 23-2 Expanding Document Inclusions Using XDBURIType

Validation of 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.

Update of a Compound Document

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.

Compound Document Versioning, Locking, and Access Control

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.

Related Topics

- Expanding Compound-Document Inclusions You can optionally expand xi:include elements with their targets when you retrieve a compound document from Oracle XML DB Repository.
- Resource Versions
 Oracle XML DB Repository resources can be versioned. A record is kept of all changes to
 a resource that is under version control.
- 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. You can create, set, and modify access control lists (ACLs).

Use View DOCUMENT_LINKS to Examine XLink and XInclude Links

You can query the read-only public view DOCUMENT_LINKS to obtain system information about document links derived from both XLink (*deprecated*) 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 readproperties on it.
- The resource is a link target, and you have the privilege read-properties on it.
- Querying DOCUMENT_LINKS for XLink Information

If the folder containing a given resource has been configured to map XLink links to document hard links then you can query public view DOCUMENT_LINKS to obtain system information about the document links.

• Querying DOCUMENT_LINKS for XInclude Information You can query view DOCUMENT_LINKS to show the document links that are mapped from XInclude links.

See Also:

Oracle Database Reference for more information on public view DOCUMENT LINKS

Querying DOCUMENT_LINKS for XLink Information

If the folder containing a given resource has been configured to map XLink links to document hard links then you can query public view <code>DOCUMENT_LINKS</code> to obtain system information about the document links.

Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

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.

See Also:

Example 23-5 for an example of configuring a folder to map XLink links to hard links

Example 23-3 Querying Document Links Mapped From XLink Links



```
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 REPOS.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

Querying DOCUMENT_LINKS for XInclude Information

You can query view DOCUMENT_LINKS to show the document links that are mapped from XInclude links.

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_REPOS.crea			
'/public/hardlink	dir/book.xml',		
' <book xmlns:xi="</td><td>http://www.w3.org/2001/XInclud</td><td>e"></book>			
<xi:include hr<="" td=""><td>ef="/public/hardlinkdir/toc.xm</td><td>1"/></td><td></td></xi:include>	ef="/public/hardlinkdir/toc.xm	1"/>	
<xi:include hr<="" td=""><td>ef="/public/hardlinkdir/part1.</td><td>xml"/></td><td></td></xi:include>	ef="/public/hardlinkdir/part1.	xml"/>	
<xi:include hr<="" td=""><td>ef="/public/hardlinkdir/part2.</td><td>xml"/></td><td></td></xi:include>	ef="/public/hardlinkdir/part2.	xml"/>	
<xi:include hr<="" td=""><td>ef="/public/hardlinkdir/index.:</td><td>xml"/></td><td></td></xi:include>	ef="/public/hardlinkdir/index.:	xml"/>	
');	· - ·		
END;			
FROM DOCUMENT_LINKS dl, RE	2.ANY_PATH target, dl.LINK_TYP SOURCE_VIEW r1, RESOURCE_VIEW ; SID and dl.TARGET_ID = r2.RESI;	r2	_FORM
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 X /public/hardlinkdir/book.xml /public/hardlinkdir/index.xml Hard X

XInclude XInclude

Configuration of Repository Resources for XLink and XInclude

The resource configuration file that you use as a resource to configure XLink (*deprecated*) and XInclude processing for other resources is described.

You configure XLink and XInclude treatment for Oracle XML DB Repository resources as you would configure any other treatment of repository resources: using a resource configuration file. See Configuring a Resource for an example.

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_REPOS.processLinks, after specifying the appropriate resource configuration parameters.

Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

Configure the Treatment of Unresolved Links: Attribute UnresolvedLink

A LinkConfig element can have an UnresolvedLink attribute with a value of Error (default value) or Skip. This determines what happens if an XLink (*deprecated*) or XInclude link cannot be resolved at the time of document insertion into the repository (resource creation).

 Configure the Type of Document Links to Create: Element LinkType You use the LinkType element of a resource configuration file to specify the type of document link to be created whenever an XLink (*deprecated*) or XInclude link is encountered when a document is stored in Oracle XML DB Repository. The LinkType element has these possible values (element content):

- Configure the Path Format for Retrieval: Element PathFormat 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.
- Configure Conflict-Resolution for XInclude: Element ConflictRule You use the ConflictRule element of a resource configuration file to specify the conflictresolution 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):
- Configure the Decomposition of Documents Using XInclude: Element SectionConfig You use element SectionConfig of a resource configuration file to specify how nonschema-based XML documents are to be decomposed when they are added to Oracle XML DB Repository to create a set of resources that contain XInclude inclusion references.
- XLink and XInclude Configuration Examples Examples show how XLink (*deprecated*) and XInclude are to be related to Oracle XML DB Repository resources and links.

Related Topics

- Manage XLink and XInclude Links Using DBMS_XDB_REPOS.processLinks
 You can use PL/SQL procedure DBMS_XDB_REPOS.processLinks to manually process all
 XLink (*deprecated*) and XInclude links in a single document or in all documents of a folder.
- Configuration of Oracle XML DB Repository
 Overall configuration of Oracle XML DB Repository applies to all repository resources. It
 does not include configuring parameters for handling events or managing XLink and
 XInclude processing. You use resource configuration files to configure resources.

Configure the Treatment of Unresolved Links: Attribute UnresolvedLink

A LinkConfig element can have an UnresolvedLink attribute with a value of Error (default value) or Skip. This determines what happens if an XLink (*deprecated*) 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_REPOS.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.



Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

Related Topics

• Hard Links and Weak Links

Links that target repository resources can be hard or weak. Hard and weak links have different dependencies with respect to the resources that they target. Hard links cannot target ancestor folders; weak links can. You can query the repository path view, PATH_VIEW, to determine the type of a repository link.

Manage XLink and XInclude Links Using DBMS_XDB_REPOS.processLinks
 You can use PL/SQL procedure DBMS_XDB_REPOS.processLinks to manually process all XLink (*deprecated*) and XInclude links in a single document or in all documents of a folder.

Configure the Type of Document Links to Create: Element LinkType

You use the LinkType element of a resource configuration file to specify the type of document link to be created whenever an XLink (*deprecated*) 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.

Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

See Also:

- Example 23-5
- Example 23-6

Configure the Path Format for Retrieval: Element PathFormat

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.

Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

See Also:

- Example 23-5
- Example 23-6

Configure Conflict-Resolution for XInclude: Element ConflictRule

You use the ConflictRule element of a resource configuration file to specify the conflictresolution 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.

Related Topics

Resource Versions

Oracle XML DB Repository resources can be versioned. A record is kept of all changes to a resource that is under version control.

Configure the Decomposition of Documents Using XInclude: Element SectionConfig

You use element SectionConfig of a resource configuration file to specify how non-schemabased XML documents are to be decomposed when they are 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
- Example 23-8

XLink and XInclude Configuration Examples

Examples show how XLink (*deprecated*) and XInclude are to be related to Oracle XML DB Repository resources and links.

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 23-6 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 23-7 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.

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/chapN.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.

Note:

The use of XLink with Oracle XML DB Repository is *deprecated*, starting with Oracle Database 12c Release 2 (12.2.0.1).

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 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 Configuring XInclude Document Decomposition

```
<ResConfig>
```



```
</ResConfig>
```

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>
```

Related Topics

- Configuration of Oracle XML DB Repository
 Overall configuration of Oracle XML DB Repository applies to all repository resources. It
 does not include configuring parameters for handling events or managing XLink and
 XInclude processing. You use resource configuration files to configure resources.
- XDBResConfig.xsd: XML Schema for Resource Configuration A full listing is presented of the Oracle XML DB-supplied XML schema used to configure repository resources. It is accessible in Oracle XML DB Repository at path /sys/schemas/ PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd.
- Configure the Decomposition of Documents Using XInclude: Element SectionConfig You use element SectionConfig of a resource configuration file to specify how nonschema-based XML documents are to be decomposed when they are added to Oracle XML DB Repository to create a set of resources that contain XInclude inclusion references.

Manage XLink and XInclude Links Using DBMS_XDB_REPOS.processLinks

You can use PL/SQL procedure DBMS_XDB_REPOS.processLinks to manually process all XLink (*deprecated*) 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.

Related Topics

Configure the Treatment of Unresolved Links: Attribute UnresolvedLink

A LinkConfig element can have an UnresolvedLink attribute with a value of Error (default value) or Skip. This determines what happens if an XLink (*deprecated*) or XInclude link cannot be resolved at the time of document insertion into the repository (resource creation).

Repository Access Using RESOURCE_VIEW and PATH_VIEW

Predefined public views RESOURCE_VIEW and PATH_VIEW provide access to Oracle XML DB repository data. You can use Oracle SQL functions under_path and equals_path to query resources based on their path names, and functions path and depth to return resource path names and depths.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

- Overview of Oracle XML DB RESOURCE_VIEW and PATH_VIEW
 Public views RESOURCE_VIEW and PATH_VIEW provide a mechanism for using SQL to access data stored in Oracle XML DB Repository. You can use these views to access data stored in the repository using Internet protocols such as FTP and WebDAV or using application program interfaces (APIs).
- Oracle SQL Functions That Use RESOURCE_VIEW and PATH_VIEW Oracle SQL functions used with public views RESOURCE_VIEW and PATH_VIEW include under_path, equals_path, path, and depth.
- Accessing Repository Data Paths, Resources and Links: Examples Examples here illustrate how to access Oracle XML DB Repository paths, resources, and link properties.
- Deleting Repository Resources: Examples
 Examples here illustrate how to delete Oracle XML DB Repository resources and paths.
- Updating Repository Resources: Examples
 Examples here illustrate how to update Oracle XML DB Repository resources and paths.
- Working with Multiple Oracle XML DB Resources
 To perform an 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.
- Performance Guidelines for Oracle XML DB Repository Operations Guidelines are presented for improving the performance of repository operations such as resource creation and querying.
- 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.



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

Public views RESOURCE_VIEW and PATH_VIEW provide a mechanism for using SQL to access data stored in Oracle XML DB Repository. You can use these views to access data stored in the repository using Internet protocols such as FTP and WebDAV or using application program interfaces (APIs).

Figure 24-1 illustrates this.

RESOURCE_VIEW consists of a resource, itself an XMLType instance, that contains the name of the resource, its ACLs, and its properties, static or extensible.

- If the content of a 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 of a resource is not XML data then the RESOURCE_VIEW stores the content as a LOB.

Note:

As of Oracle Database Release 11.2.0.1.0, repository content stored in line as a LOB uses SecureFiles LOB storage. Prior to that, it used BasicFiles LOB storage.

Parent-child relationships between folders 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 data.

RESOURCE_VIEW and PATH_VIEW, along with PL/SQL package DBMS_XDB_REPOS, 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 using RESOURCE VIEW or PL/SQL package DBMS XDB REPOS.



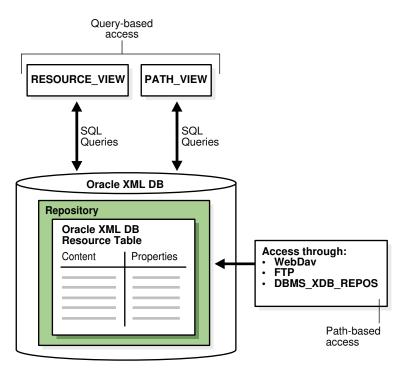


Figure 24-2 illustrates the structure of RESOURCE VIEW and PATH VIEW.

Figure 24-2 RESOURCE_VIEW and PATH_VIEW Structure

RESOURCE_VIEW Columns Resource as an XMLType Path Resource OID			PATH_ Path

PATH_VIEW Columns				
Path	Resource as an XMLType	Link as XMLType	Resource OID	

Note:

Neither RESOURCE_VIEW nor PATH_VIEW contains the *root* folder (/) resource. All other repository resources are included.

A path in the RESOURCE_VIEW is an arbitrary one of the paths that can be used to access the given 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 data type 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 <code>under_path</code> are returned. If <code>/a/b/c</code>, <code>/a/b/d</code>, and <code>/a/e</code> are all links to the same resource, then a query on <code>PATH_VIEW</code> that retrieves all of the paths under <code>/a/b</code> returns only <code>/a/b/c</code> and <code>/a/b/d</code>, not <code>/a/e</code>.

RESOURCE_VIEW Definition and Structure

Public view RESOURCE_VIEW contains one row for each resource in Oracle XML DB Repository (except for the root folder resource).

PATH_VIEW Definition and Structure
 Public view PATH_VIEW contains one row for each unique path that accesses a resource in
 Oracle XML DB Repository (except for the root folder resource). Each resource can have
 multiple paths, also called links.

• The Difference Between RESOURCE_VIEW and PATH_VIEW PATH_VIEW includes all the path names to a particular resource. RESOURCE_VIEW includes one of the possible path names to the resource. PATH_VIEW also includes the link properties. For better performance, use RESOURCE_VIEW, not PATH_VIEW, whenever possible.

Operations You Can Perform Using UNDER_PATH and EQUALS_PATH

You can use Oracle SQL functions under_path and equals_path to get a resource or its OID; create, delete, or update a resource; or list a directory that corresponds to a path name.

Related Topics

 Overview of How To Use Oracle XML DB An overview of the various ways of using Oracle XML DB is presented.

RESOURCE_VIEW Definition and Structure

Public view RESOURCE_VIEW contains one row for each resource in Oracle XML DB Repository (except for the root folder resource).

Table 24-1 describes the 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

Table 24-1 Structure of RESOURCE_VIEW

PATH_VIEW Definition and Structure

Public view PATH_VIEW contains one row for each unique path that accesses a resource in Oracle XML DB Repository (except for the root folder resource). Each resource can have multiple paths, also called **links**.

Table 24-2 describes the structure of PATH VIEW.

Table 24-2 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

The Difference Between RESOURCE_VIEW and PATH_VIEW

PATH_VIEW includes all the path names to a particular resource. RESOURCE_VIEW includes one of the possible path names to the resource. PATH_VIEW also includes the link properties. For better performance, use RESOURCE_VIEW, not PATH_VIEW, whenever possible.

Figure 24-3 illustrates this difference.

Because many Internet applications need only one URL to access a resource, RESOURCE_VIEW is widely applicable.

PATH_VIEW contains both *link* properties and resource properties, whereas the RESOURCE_VIEW contains only resource properties.

Because it handles the information for multiple paths, PATH_VIEW access can be slower. If you use RESOURCE_VIEWthen the database can take advantage of the fact that only one path is needed; the index can do less work to determine all the possible paths.

Note:

When using RESOURCE_VIEW, if you specify a path using function under_path or equals_path, the function finds the resource regardless of whether or not the specified path is the arbitrary one chosen to normally appear with that resource using RESOURCE VIEW.



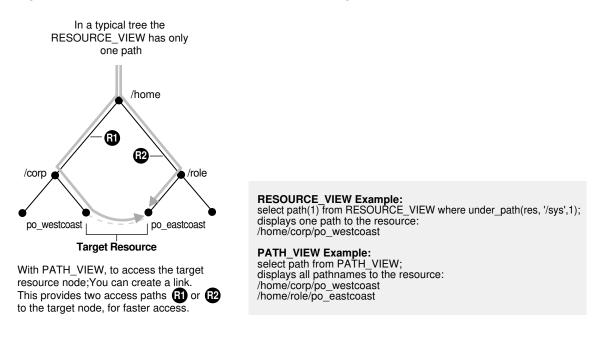


Figure 24-3 RESOURCE_VIEW and PATH_VIEW Explained

Operations You Can Perform Using UNDER_PATH and EQUALS_PATH

You can use Oracle SQL functions under_path and equals_path to get a resource or its OID; create, delete, or update a resource; or list a directory that corresponds to a path name.

- Given a path name, you can:
 - 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, you can:
 - Update resources
 - Delete resources
 - Get resources or their OID

Oracle SQL Functions That Use RESOURCE_VIEW and PATH_VIEW

Oracle SQL functions used with public views RESOURCE_VIEW and PATH_VIEW include under_path, equals_path, path, and depth.

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).



EQUALS_PATH SQL Function

You use Oracle SQL function equals_path to find a resource that has a given path name. It is functionally equivalent to under path with a depth restriction of zero.

- PATH SQL Function
 Oracle SQL function path returns the relative path name of the resource under a given
 pathname argument to function under_path or equal_path.
- DEPTH SQL Function

Oracle SQL function depth returns the folder depth of the resource under the specified starting path.

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 a 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 24-4 shows the under_path syntax.

Figure 24-4 UNDER_PATH Syntax

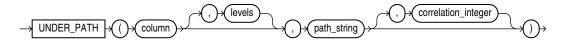


Table 24-3 details the signature of Oracle SQL function under path.

Table 24-3	UNDER	_PATH SQL	Function	Signature
------------	-------	-----------	----------	-----------

Syntax	Description
under path(resource column,	Determines whether a resource is under a specified path.
pathname);	Parameters:
<pre>under_path(resource_column, depth, pathname);</pre>	 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.
	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.
	• <i>pathname</i> – The path name to resolve.

Syntax	Description		
<pre>under_path(resource_column, pathname, correlation);</pre>	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). 		
<pre>under_path(resource_column, depth, pathname, correlation);</pre>	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.		
correration),	Parameters:		
	• resource_column – The column name or column alias of the RESOURCE column in the PATH_VIEW or RESOURCE_VIEW.		
	• <i>depth</i> – The maximum depth to search. A nonnegative integer.		
	• <i>pathname</i> – The path name to resolve.		
	 correlation – An integer that can be used to correlate under_path with related SQL functions (path and depth). 		
	For a resource to be returned, only one of the accessible paths to the resource must be under the <i>pathname</i> argument. If no such path is under argument <i>pathname</i> then a NULL value is returned.		

Table 24-3 (Cont.) UNDER_PATH SQL Function Signature

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

You use Oracle SQL function equals_path to find a resource that has a given path name. It is functionally equivalent to under path with a depth restriction of zero.

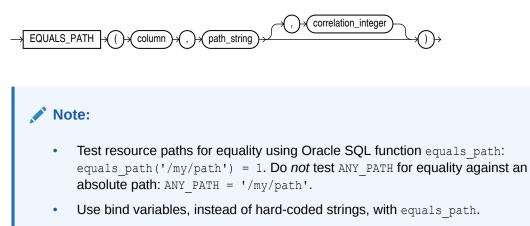
equals path(resource column, pathname);

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 24-5 illustrates the complete equals_path syntax.

Figure 24-5 EQUALS_PATH Syntax



PATH SQL Function

Oracle SQL function path returns the relative path name of the resource under a given pathname argument to function under path or equal path.

The path column in the RESOURCE_VIEW always contains the absolute path of the resource. The syntax of function path is:

path(correlation);

where:

• correlation is an integer that can be used to correlate path with under_path or equals_path.

Figure 24-6 illustrates the syntax for function path.

Figure 24-6 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 depth with path with under_path or equals_path.



Accessing Repository Data Paths, Resources and Links: Examples

Examples here illustrate how to access Oracle XML DB Repository paths, resources, and link properties.

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 24-1 uses Oracle SQL function path to retrieve the relative paths under path /a/b.

Example 24-2 uses ANY PATH to retrieve the absolute paths under path /a/b.

Example 24-3 is the same as Example 24-2, except that the test is *not*-equals (!=) instead of equals (=). The query in Example 24-3 finds *all paths in the repository* that are *not* under path /a/b.

Example 24-4 shows the relative paths that are under repository folders a/b and /a/e, respectively. The expression path(1) represents the paths that are under folder a/b, since it uses the same correlation number, 1, as the expression under_path(RES, '/a/b', 1), which specifies folder a/b. Similarly for path(2) and folder /a/e. Expression ANY_PATH returns the corresponding absolute paths.

Example 24-1 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 24-2 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 24-3 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 24-4 Determining Paths Using Multiple Correlations



4 rows selected.

Example 24-5 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 24-6 Extracting Resource Metadata Using UNDER_PATH

This produces a result similar to the following:



Example 24-7 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
<pre>schemas/OE/localhost:8080</pre>	3
schemas/PUBLIC	2
schemas/PUBLIC/www.w3.org	3
<pre>schemas/PUBLIC/xmlns.oracle.com</pre>	3

Example 24-8 Extracting Link and Resource Information from PATH_VIEW

```
$1/ns:LINK/ns:ChildName' PASSING LINK AS "1" 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 24-9 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 24-10 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 24-11 Retrieve RESID of a Given Resource

This produces a result similar to the following:

```
1 row selected.
```

Example 24-12 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 24-13 Folders Under a Given Path

This produces a result like the following:

```
ANY_PATH
/sys/acls
/sys/apps
/sys/log
/sys/schemas
```

4 rows selected.

Example 24-14 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"RET
_____
/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

Examples here illustrate how to delete Oracle XML DB Repository resources and paths.

If you delete only *leaf* resources, then you can use DELETE FROM RESOURCE_VIEW, as in Example 24-15.

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 24-16 illustrates this.

Example 24-15 Deleting Resources

DELETE FROM RESOURCE VIEW WHERE equals path(RES, '/public/myfile') = 1';

Example 24-16 Deleting Links to Resources

Suppose that '/home/myfile1' is a link to '/public/myfile':

CALL DBMS XDB REPOS.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.

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 24-17 illustrates how to delete a nonempty folder: folder example is deleted, along with its subfolder example1.

Note:

As always, take care to avoid deadlocks with concurrent transactions when operating on multiple rows.

Example 24-17 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
```

Updating Repository Resources: Examples

Examples here illustrate how to update Oracle XML DB Repository resources and paths.

Example 24-18 changes the resource at path /test/HR/example/paper.

See Also: User-Defined Repository Metadata for additional examples of updating resource metadata

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 24-19 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.

💉 See Also:

Table 21-3 for additional examples that use SQL functions that apply to RESOURCE_VIEW and PATH_VIEW

Example 24-18 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>
    <binarv>4F7261636C65206F7220554E4958</binarv>
 </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 =
   XMLQuery('copy $i := $p1 modify
               (for $j in $i/Resource/DisplayName
               return replace value of node $j with $p2)
            return $i'
            PASSING r.RES AS "p1", 'My New Paper' AS "p2"
            RETURNING CONTENT)
   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.
```

Example 24-19 Updating a Path in the PATH_VIEW



Working with Multiple Oracle XML DB Resources

To perform an 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 <code>RESOURCE_VIEW</code> and <code>PATH_VIEW</code>.

For example, you can perform the following operations:

- Update resources based on attributes see Example 24-20
- Finding resources inside a folder see Example 24-21
- Copy a set of Oracle XML DB resources see Example 24-22

The SQL DML statement in Example 24-22 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 24-20 Updating Resources Based on Attributes

```
UPDATE RESOURCE VIEW
  SET RES =
    XMLQuery('copy $i := $p1 modify
                (for $j in $i/Resource/DisplayName
                 return replace value of node $j with $p2)
               return $i'
              PASSING RES AS "p1", 'My New Paper' AS "p2"
              RETURNING CONTENT)
  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 24-21 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/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/xdblog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/xdblog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/xdblog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/xdblog.xsd
```

12 rows selected.

Example 24-22 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 Guidelines for Oracle XML DB Repository Operations

Guidelines are presented for improving the performance of repository operations such as resource creation and querying.

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_REPOS.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.

Related Topics

 Administration of Oracle XML DB Administration of Oracle XML DB includes installing, upgrading, and configuring it.

🖍 See Also:

Oracle Database PL/SQL Packages and Types Reference for information about PL/SQL function DBMS XDB REPOS.createResource

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.

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 24-24.

```
CREATE INDEX xdb$resource_ctx_i ON XDB.XDB$RESOURCE(OBJECT_VALUE)
INDEXTYPE IS CTXSYS.CONTEXT;
```

Example 24-23 Find All Resources Containing "Paper"

SELECT PATH FROM PATH VIEW WHERE contains (RES, 'Paper') > 0;

PATH /newlocation/myexample /test/myexample

2 rows selected.

Example 24-24 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.

Related Topics

- Query and Update of XML Data There are many ways for applications to query and update XML data that is in Oracle Database, both XML schema-based and non-schema-based.
- PL/SQL APIs for XMLType: References The PL/SQL Application Programming Interfaces (APIs) for XMLType are described.



Resource Versions

Oracle XML DB Repository resources can be versioned. A record is kept of all changes to a resource that is under version control.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

Overview of Oracle XML DB Repository Resource Versioning

You can create and manage different versions of a repository resource. A record is kept of all changes to a resource that is under version control. When you update a version-controlled resource, the pre-update contents are stored as a separate resource version -a snapshot for the historical record.

- Overview of PL/SQL Package DBMS_XDB_VERSION
 You use PL/SQL package DBMS_XDB_VERSION to work with Oracle XML DB Repository resource versions.
- Resource Versions and Resource IDs

A resource object ID, or **resource ID**, is a unique, constant, system-generated identifier for a 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**.

• Resource Versions 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 access control list (ACL) of a version resource cannot be changed.

Resource Versioning Examples

Examples here create a version-controlled resource; retrieve the content of a resource using its resource ID; check out a version-controlled resource (for all users); update the content of a resource; check in a resource; retrieve the content and metadata of different versions of a resource; and cancel a resource check-out.

Overview of Oracle XML DB Repository Resource Versioning

You can create and manage different versions of a repository resource. A record is kept of all changes to a resource that is under version control. When you update a version-controlled resource, the pre-update contents are stored 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 Resource Versions and Resource IDs.

Table 25-1 lists some terms used in this chapter.

Table 25-1 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 <pre>checkOut</pre> before you can modify a version-controlled resource. Use <pre>checkIn</pre> to make your changes permanent. Use <pre>unCheckOut</pre> to cancel your changes. (Use <pre>COMMIT</pre> 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 checkin 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 schemabased. 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.



Overview of PL/SQL Package DBMS_XDB_VERSION

You use PL/SQL package DBMS_XDB_VERSION to work with Oracle XML DB Repository resource versions.

Table 25-2 summarizes the main DBMS_XDB_VERSION subprograms.

Table 25-2 PL/SQL Functions and Procedures in Package DBMS_XDB_VERSION

Function or Procedure	Description
makeVersioned(pathname	Turn a resource with the given path name into a version controlled resource.
VARCHAR2) RETURN DBMS_XDB_VERSION.RESID_TYP E;	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	Check in a version-controlled resource that has been checked out.
VARCHAR2) RETURN	pathname - Path name of the checked-out resource.
DBMS_XDB_VERSION.RESID_TYP	Returns the resource id of the newly created version.
Ε;	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	Check in a checked-out resource.
VARCHAR2) RETURN	The argument is the path name of the checked-out resource.
DBMS_XDB_VERSION.RESID_TYP E;	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.
getPredecessors(pathname VARCHAR2) RETURN	Given a path name that references a version resource or a version-controlled resource, return the predecessors of the resource.
RESID_LIST_TYPE; getPredsByRESID(resid	Retrieving predecessors by resource ID, using function getPredsByRESID is more efficient than by path name, using function getPredecessors.
DBMS_XDB_VERSION.RESID_TYP E) RETURN RESID_LIST_TYPE;	The list of predecessors returned has only one element (the parent): Oracle XML DB does not support version branching.

Function or Procedure	Description
getSuccessors(pathname VARCHAR2) RETURN	Given a version resource or a version-controlled resource, return the successors of the resource.
RESID_LIST_TYPE;	Retrieving successors by resource ID, using function getSuccsByRESID is more
getSuccsByRESID(resid	efficient than by path name, using function getSuccessors.
<pre>DBMS_XDB_VERSION.RESID_TYP E) RETURN RESID_LIST_TYPE;</pre>	The list of successors returned has only one element (the parent): Oracle XML DB does not support version branching.
<pre>getResourceByRESID(resid DBMS_XDB_VERSION.RESID_TYP E) RETURN XMLType;</pre>	Given a resource ID, return the resource as an XMLType instance.

Table 25-2 (Cont.) PL/SQL Functions and Procedures in Package DBMS_XDB_VERSION

Resource Versions and Resource IDs

A resource object ID, or **resource ID**, is a unique, constant, system-generated identifier for a 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 \mathbf{R} , Oracle XML DB creates a new resource ID, \mathbf{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 \mathbf{P}) are created at check-in time. Together, the new resource ID \mathbf{P} and the snapshot it refers to thus represent the *previous*, not the current, version of the resource. Resource ID \mathbf{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.

Resource Versions 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 access control list (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

Examples here create a version-controlled resource; retrieve the content of a resource using its resource ID; check out a version-controlled resource (for all users); update the content of a resource; check in a resource; retrieve the content and metadata of different versions of a resource; and cancel a resource check-out.

- Putting a resource under version control Example 25-2
- Retrieving the content of the resource using its resource ID Example 25-3
- Checking out a version-controlled resource (for all users) Example 25-4
- Updating the content of a resource Example 25-5
- Checking in a resource Example 25-6
- Retrieving the content and metadata of different versions of a resource Example 25-7, Example 25-8, Example 25-9
- Canceling a resource check-out Example 25-10

Example 25-3 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.

The new resource is *not* version-controlled. Example 25-2 uses PL/SQL function DBMS_XDB_VERSION.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 25-2 also copies the resource ID of the new version resource to bind variable current_RESID. Example 25-3 shows how to use PL/SQL constructor XDBUritype together with PL/SQL function createOIDPath to retrieve the resource content by referencing the resource ID.

Example 25-4 checks out the version-controlled resource (and commits), so that it can be modified. Any user can modify a resource that has been checked out.

Example 25-5 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 25-5 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 25-6 checks in the resource and commits the check-in.



PL/SQL function checkIn returns the resource ID of the current version, which is the same as current_RESID. Example 25-6 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 25-6 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:

- PL/SQL constructor XDBURIType, together with PL/SQL function DBMS_XDB_REPOS.createOIDPath – Retrieve content. See Example 25-3 and Example 25-7.
- PL/SQL function DBMS_XDB_VERSION.getContentsCLOBByRESID Retrieve content. See Example 25-8.
- PL/SQL function DBMS_XDB_VERSION.getResourceByRESID Retrieve metadata. See Example 25-9.

You can use constructor XDBURITYPE with function 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 25-7 through Example 25-9 use these different methods to retrieve the two versions of the resource addressed by bind variables <code>current_RESID</code> and <code>previous_RESID</code> after check-in.

You can cancel a check-out using PL/SQL function DBMS_XDB_VERSION.unCheckOut. Example 25-10 illustrates this.

Example 25-1 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_REPOS.existsResource(:targetPath))
THEN DBMS_XDB_REPOS.deleteResource(:targetPath);
END IF;
res := DBMS_XDB_REPOS.createResource(:targetPath, 'Mary had a little lamb');
END;
```

Example 25-2 Creating a Version-Controlled Resource

```
DECLARE
  resid DBMS_XDB_VERSION.RESID_TYPE;
BEGIN
  resid := DBMS_XDB_VERSION.makeVersioned(:targetPath);
  :current_RESID := resid;
  COMMIT;
```

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.



```
END;
/
```

Example 25-3 Retrieving Resource Content by Referencing the Resource ID

```
SELECT XDBURIType(DBMS_XDB_REPOS.createOIDPath(:current_RESID)).getClob()
FROM DUAL;
```

XDBURITYPE(DBMS_XDB_REPOS.CREATEOIDPATH(:CURRENT_RESID)).GETCLOB()

```
Mary had a little lamb
```

```
1 row selected.
```

Example 25-4 Checking Out a Version-Controlled Resource

```
BEGIN
    DBMS_XDB_VERSION.checkOut(:targetPath);
    COMMIT;
END;
/
```

Example 25-5 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 =
   XMLQuery('copy $i := $p1 modify
                (for $j in $i/Resource/DisplayName
                 return replace value of node $j with $p2)
                return $i'
               PASSING
                 RES AS "p1",
                 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)) AS "p2"
               RETURNING CONTENT)
   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 25-6 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;
/
```

Example 25-7 Retrieving Resource Version Content Using XDBURITYPE and CREATEOIDPATH

```
SELECT XDBURIType(DBMS_XDB_REPOS.createOIDPath(:current_RESID)).getClob()
FROM DUAL;
```

XDBURITYPE (DBMS_XDB_REPOS.CREATEOIDPATH (:CURRENT_RESID)).GETCLOB()

Mary had a little lamb

1 row selected.

```
SELECT XDBURIType(DBMS_XDB_REPOS.createOIDPath(:previous_RESID)).getClob()
FROM DUAL;
```

XDBURITYPE(DBMS_XDB_REPOS.CREATEOIDPATH(:PREVIOUS_RESID)).GETCLOB()

Hickory dickory dock, the mouse ran up the clock

1 row selected.

Example 25-8 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 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 25-9 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.
```

Example 25-10 Canceling a Check-Out Using UNCHECKOUT

```
DECLARE
  resid DBMS_XDB_VERSION.RESID_TYPE;
BEGIN
  resid := DBMS_XDB_VERSION.unCheckOut(:targetPath);
END;
/
```

26 PL/SQL Access to Oracle XML DB Repository

PL/SQL packages DBMS_XDB_CONFIG and DBMS_XDB_REPOS together provide the Oracle XML DB resource application program interface (API) for PL/SQL. You use the former to configure Oracle XML DB and its repository. You use the latter to perform other, non-configuration operations on the repository.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

DBMS_XDB_REPOS: Access and Manage Repository Resources

You use PL/SQL package DBMS_XDB_REPOS to access and manage resources in Oracle XML DB Repository. It includes methods for managing resource 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.

- DBMS_XDB_REPOS: ACL-Based Security Management PL/SQL package DBMS_XDB_REPOS provides Oracle XML DB ACL-based security management functions and procedures.
- DBMS_XDB_CONFIG: Configuration Management PL/SQL package DBMS_XDB_CONFIG provides Oracle XML DB configuration management functions and procedures.

Related Topics

- PL/SQL APIs for XMLType: References The PL/SQL Application Programming Interfaces (APIs) for XMLType are described.
- Package DBMS_XDB_ADMIN
 PL/SQL package DBMS_XDB_ADMIN has subprograms for managing and configuring Oracle
 XML DB and Oracle XML DB Repository.

DBMS_XDB_REPOS: Access and Manage Repository Resources

You use PL/SQL package DBMS_XDB_REPOS to access and manage resources in Oracle XML DB Repository. It includes methods for managing resource 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.

See Also:

Oracle Database PL/SQL Packages and Types Reference



Table 26-1 describes the functions and procedures in package DBMS_XDB_REPOS.

Function/Procedure	Description
addResource	Insert a new file resource into the repository hierarchy, with the given string as its contents.
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.
getContentBLOB	Return the contents of a resource as a BLOB instance.
getContentVARCHAR2	Return the contents of a resource as a VARCHAR2 value.
getContentXMLRef	Return the contents of a resource as a reference to an XMLType instance.
getContentXMLType	Return the contents of a resource as an XMLType instance.
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.
getResource	Return the instance of class DBMS_XDBRESOURCE.XDBResource that is located at a given path in the repository.
getXDB_tablespace	Return the current tablespace of database schema (user account) XDB.
nasBLOBContent	Return TRUE if a given resource has BLOB content.
nasCharContent	Return TRUE if a given resource has character content.
nasXMLContent	Return TRUE if a given resource has XMLType content.
nasXMLReference	Return TRUE if a given resource has a reference to XMLType content.
isFolder	Return TRUE if a given resource is a folder.
link	Create a link to an existing resource.
lockResource	Obtain a WebDAV-style lock on a resource, given a path to the resource.
processLinks	Process all XLink (deprecated) and XInclude links in a document or folder.
ourgeResourceMetadata	Delete all user-defined metadata from a given resource.
refreshContentSize	Recompute the content size of a given resource.
renameResource	Rename a given resource.
touchResource	Change the last-modified time to the current time.
unlockResource	Unlock a resource, given its lock token and path.
updateResourceMetadata	Modify user-defined resource metadata.

Example 26-1 uses package DBMS_XDB_REPOS 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_REPOS

```
DECLARE
 retb BOOLEAN;
BEGIN
  retb := DBMS XDB REPOS.createfolder('/public/mydocs');
 retb := DBMS XDB REPOS.createresource('/public/mydocs/emp selby.xml',
                                         '<emp name>selby</emp name>');
 retb := DBMS XDB REPOS.createresource('/public/mydocs/emp david.xml',
                                         '<emp name>david</emp name>');
END;
PL/SQL procedure successfully completed.
CALL DBMS_XDB_REPOS.link('/public/mydocs/emp_selby.xml',
                         '/public/mydocs',
                         'person selby.xml');
Call completed.
CALL DBMS_XDB_REPOS.link('/public/mydocs/emp_david.xml',
                         '/public/mydocs',
                         'person_david.xml');
Call completed.
CALL DBMS XDB REPOS.deleteresource('/public/mydocs/emp selby.xml');
Call completed.
CALL DBMS XDB REPOS.deleteresource('/public/mydocs/person selby.xml');
Call completed.
CALL DBMS XDB REPOS.deleteresource('/public/mydocs/emp david.xml');
Call completed.
CALL DBMS XDB REPOS.deleteresource('/public/mydocs/person david.xml');
Call completed.
CALL DBMS XDB_REPOS.deleteresource('/public/mydocs');
Call completed.
```

DBMS_XDB_REPOS: ACL-Based Security Management

PL/SQL package DBMS_XDB_REPOS provides Oracle XML DB ACL-based security management functions and procedures.



Function/Procedure	Description
ACLCheckPrivileges	Check the access privileges granted to the current user by an ACL.
changeOwner	Change the owner of a given resource to a given user.
changePrivileges	Add an ACE to a resource ACL.
checkPrivileges	Check the access privileges granted to the current user for a resource.
getACLDocument	Return the ACL document that protects a resource, given the path name of the resource.
getPrivileges	Return all privileges granted to the current user for a resource.
setACL	Set the ACL for a resource.

Table 26-2 DBMS_XDB_REPOS: Security Management Subprograms

See Also:

Oracle Database PL/SQL Packages and Types Reference

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.

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).

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.

In Example 26-5, user oe connects and calls DBMS_XDB_REPOS.getPrivileges, which shows all of the privileges granted to user oe on resource emp selby.xml.

Example 26-2 Using DBMS_XDB_REPOS.GETACLDOCUMENT

XMLSERIALIZE(DOCUMENTDBMS XDB REPOS.GETACLDOCUMENT('/PUBLIC/MYDOCS/EMP SELBY.XML

1 row selected.

Example 26-3 Using DBMS_XDB_REPOS.SETACL

CONNECT SYSTEM Enter password: password Connected. -- Give all privileges to owner, HR. CALL DBMS XDB REPOS.setACL('/public/mydocs/emp selby.xml', '/sys/acls/all_owner_acl.xml'); Call completed. COMMIT; Commit complete. SELECT XMLSerialize (DOCUMENT DBMS XDB REPOS.getACLDocument('/public/mydocs/emp selby.xml') AS CLOB) FROM DUAL; XMLSERIALIZE (DOCUMENTDBMS XDB REPOS.GETACLDOCUMENT ('/PUBLIC/MYDOCS/EMP SELBY.XML _____ <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. Example 26-4 Using DBMS XDB REPOS.CHANGEPRIVILEGES CONNECT hr Enter password: password Connected.

```
-- 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"
```

SET SERVEROUTPUT ON

```
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 REPOS.changePrivileges('/public/mydocs/emp selby.xml', ace);
END:
PL/SQL procedure successfully completed.
COMMIT;
SELECT XMLSerialize (DOCUMENT
                   DBMS XDB REPOS.getACLDocument('/public/mydocs/emp selby.xml')
                   AS CLOB)
  FROM DUAL;
XMLSERIALIZE (DOCUMENTDBMS XDB REPOS.GETACLDOCUMENT ('/PUBLIC/MYDOCS/EMP SELBY.XML
_____
<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.
```

Example 26-5 Using DBMS_XDB_REPOS.GETPRIVILEGES

```
CONNECT oe
Enter password: password
Connected.
SELECT XMLSerialize(DOCUMENT
                  DBMS XDB REPOS.getPrivileges('/public/mydocs/emp selby.xml')
                  AS CLOB)
 FROM DUAL;
XMLSERIALIZE (DOCUMENTDBMS XDB REPOS.GETPRIVILEGES ('/PUBLIC/MYDOCS/EMP SELBY.XML'
 _____
<privilege xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.</pre>
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:take-ownership/> <dav:read-current-user-privilege-set/> </privilege>

1 row selected.

DBMS_XDB_CONFIG: Configuration Management

PL/SQL package DBMS_XDB_CONFIG provides Oracle XML DB configuration management functions and procedures.

Note:

Oracle recommends that you use the subprograms of PL/SQL package DBMS_XDB_CONFIG to set or change FTP or HTTP port numbers. Do *not* set ports by directly editing configuration file xdbconfig.xml

Subprogram	Description
addHTTPExpireMapping	Add a mapping of a URL pattern to an expiration date to table XDB\$CONFIG. The mapping controls the Expire headers for URLs that match the pattern.
addMIMEMapping	Add a MIME mapping to table XDB\$CONFIG.
addSchemaLocMapping	Add a schema-location mapping to table XDB\$CONFIG.
addServlet	Add a servlet to table XDB\$CONFIG.
addServletMapping	Add a servlet mapping to table XDB\$CONFIG.
addServletSecRole	Add a security role reference to a servlet.
addXMLExtension	Add an XML extension to table XDB\$CONFIG.
cfg_get	Return the configuration information for the current session.
cfg_refresh	Refresh the session configuration information using the current Oracle XML DB configuration file, xdbconfig.xml.
cfg_update	Update the Oracle XML DB configuration information. This writes the configuration file, xdbconfig.xml.
deleteHTTPExpireMapping	Delete all mappings of a given URL pattern to an expiration date from table XDB\$CONFIG.
deleteMIMEMapping	Delete a MIME mapping from table XDB\$CONFIG.
deleteSchemaLocMapping	Delete a schema-location mapping from table XDB\$CONFIG.

Table 26-3 DBMS_XDB_CONFIG: Configuration Management Subprograms

Subprogram	Description
deleteServlet	Delete a servlet from table XDB\$CONFIG.
deleteServletMapping	Delete a servlet mapping from table XDB\$CONFIG.
deleteServletSecRole	Delete a security role reference from a servlet.
deleteXMLExtension	Delete an XML extension from table XDB\$CONFIG.
enableDigestAuthentication	Enable digest authentication.
getFTPPort	Return the current FTP port number.
getHTTPConfigRealm	Return the HTTP configuration realm.
getHTTPPort, getHTTPSPort	Return the current HTTP(S) port number.
getListenerEndPoint	Return the parameters of a listener end point for the HTTP server.
GetRemoteHTTPPort, GetRemoteHTTPSPort	Return the number of the current remote HTTP(S) port.
setFTPPort	Set the Oracle XML DB FTP port to the specified port number.
setHTTPConfigRealm	Set the HTTP configuration realm.
setHTTPPort, setHTTPSPort	Set the Oracle XML DB HTTP(S) port to the specified port number.
setListenerEndPoint	Set the parameters of a listener end point for the HTTP server.
setListenerLocalAccess	Either (a) restrict all listener end points to listen on only the localhost interface or (b) allow all listener end points to listen on both localhost and non-localhost interfaces.
SetRemoteHTTPPort, SetRemoteHTTPSPort	Define the port number of a remote HTTP or HTTPS port, respectively. A remote port number is stored similarly to a non-remote HTTP(S) port, except they it is not specified in the configuration file, xdbconfig.xml file or in the configuration XML schema, xdbconfig.xsd.
usedPort	Return the ports used by other pluggable databases (PDBs) in the same multitenant container database (CDB). The return value is an $XMLType$ instance that lists each PDB by id number and its associated ports by type and number.

Table 26-3 (Cont.) DBMS_XDB_CONFIG: Configuration Management Subprograms

🖋 See Also:

Oracle Database PL/SQL Packages and Types Reference, Chapter "DBMS_XDB_CONFIG"

Example 26-6 uses function cfg_get to retrieve the Oracle XML DB configuration file, xdbconfig.xml.

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 26-6 Using DBMS_XDB_CONFIG.CFG_GET

CONNECT SYSTEM Enter password: password

Connected.



```
SELECT DBMS XDB CONFIG.cfg get() FROM DUAL;
DBMS_XDB_CONFIG.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>200000000</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 Using DBMS_XDB_CONFIG.CFG_UPDATE

```
DECLARE
configxml SYS.XMLType;
configxml2 SYS.XMLType;
BEGIN
-- Get the current configuration
```

```
configxml := DBMS XDB CONFIG.cfg get();
  -- Modify the configuration
SELECT XMLQuery(
  'declare default element namespace
  "http://xmlns.oracle.com/xdb/xdbconfig.xsd"; (: :)
  copy $i := $p1 modify
     (for $j in $i/xdbconfig/sysconfig/protocolconfig/httpconfig/http-port
     return replace value of node $j with $p2)
    return $i'
   PASSING CONFIGXML AS "p1", '8000' AS "p2" RETURNING CONTENT)
 INTO configxml2 FROM DUAL;
 -- Update the configuration to use the modified version
 DBMS_XDB_CONFIG.cfg_update(configxml2);
END;
/
PL/SQL procedure successfully completed.
SELECT DBMS XDB CONFIG.cfg get() FROM DUAL;
DBMS XDB CONFIG.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. You can create, set, and modify access control lists (ACLs).

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

Access Control Concepts

Access control terms and concepts are explained. Each of the access-control entity — user, role, principal, privilege, access control list (ACL), and access control entry (ACE) — is implemented declaratively as an XML document or fragment.

Database Privileges for Repository Operations

The database privileges needed for common operations on resources in Oracle XML DB Repository are described.

• Privileges

The privileges provided with Oracle Database include the standard WebDAV privileges as well as Oracle-specific privileges.

- ACLs and ACEs ACLs are used to protect resources, which in the case of Oracle Database are resources (files and folders) in Oracle XML DB Repository. ACLs are composed of ACEs.
- Overview of 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 PL/SQL package DBMS XDB REPOS.

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.

Repository Resources and Database Table Security

A uniform security mechanism for accessing REF-based repository resources is provided by enabling hierarchy on the tables used to store them. When ACL-based security is not needed for particular resources, you can optimize their access by using PL/SQL procedure DBMS XDBZ.disable hierarchy to turn off ACL checking.

 Integration Of Oracle XML DB with LDAP You can allow Lightweight Directory Access Protocol (LDAP) users to use the features of Oracle XML DB, including ACLs.



See Also:

- Repository Access Using Protocols for more information about WebDAV
- Administration of Oracle XML DB for information about configuring and administering resource security
- PL/SQL APIs for XMLType: References for information about the PL/SQL APIs you can use to manage resource security

Access Control Concepts

Access control terms and concepts are explained. Each of the access-control entity — 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_REPOS.setACL to set its controlling ACL.

Authentication and Authorization

The term **authentication** refers to verifying the *identity* of something (for example, a user, device, or application). The term **authorization** refers to verifying whether something that has been authenticated is allowed to *access* something else (for example, a database table or WebDAV resource).

• Principal: A User or Role

In the context of fine-grained database access control, a principal is ultimately a set of one or more people or applications that access information in the database. A principal can be a database user or role, or an LDAP user or group.

• 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. An aggregate privilege includes other privileges; an atomic privilege does not.

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.

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.



Authentication and Authorization

The term **authentication** refers to verifying the *identity* of something (for example, a user, device, or application). The term **authorization** refers to verifying whether something that has been authenticated is allowed to *access* something else (for example, a database table or WebDAV resource).

Principals of various kinds can be authorized to access Oracle XML DB Repository resources. Application-specific principals can be authenticated.

Related Topics

• Principal: A User or Role

In the context of fine-grained database access control, a principal is ultimately a set of one or more people or applications that access information in the database. A principal can be a database user or role, or an LDAP user or group.

Principal: A User or Role

In the context of fine-grained database access control, a principal is ultimately a set of one or more people or applications that access information in the database. A principal can be a database user or role, or an LDAP user or group.

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.
- LDAP users and groups of LDAP users. For details about using LDAP principals see Integration Of Oracle XML DB with LDAP.

When a term such as "user" or "role" is used here without qualification, it applies to all types of user or role. When it is important to distinguish the type, the qualifier "database" or "LDAP" is used.

• Database Roles and ACLs Map Privileges to Users

A database role maps privileges to users. In the context of fine-grained access control, an ACL maps privileges to users, and a role is just a set of users.

• Principal DAV::owner

You can use principal DAV::owner to refer to the owner of a given repository resource. 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.



Database Roles and ACLs Map Privileges to Users

A database role maps privileges to users. In the context of fine-grained access control, an ACL maps privileges to users, and a role is just a set of 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 to refer to the owner of a given repository resource. 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. An aggregate privilege includes other privileges; an atomic privilege does not.

- 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.

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.

Related Topics

Privileges

The privileges provided with Oracle Database include the standard WebDAV privileges as well as Oracle-specific privileges.

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.

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 REPOS.setACL.

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 globally unique identifier (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 user or database user).

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>
```

Related Topics

• ACL and ACE Evaluation

Privileges are checked before a principal is allowed to access a repository resource that is protected by 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.



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.

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

Related Topics

ACL and ACE Evaluation

Privileges are checked before a principal is allowed to access a repository resource that is protected by 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.

Database Privileges for Repository Operations

The database privileges needed for common operations on resources in Oracle XML DB Repository are described.

Table 27-1 shows the database privileges required for some common operations on resourcesin Oracle XML DB Repository. In addition to the privileges listed in column **PrivilegesRequired** you must have the resolve privilege for the folder containing the resource and for allof 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
GET	An FTP or HTTP(S) retrieval of resource R	read-properties, read-contents on ${\sf R}$
SET_ACL	Set the ACL of a resource R	DAV::write-acl on R



Operation	Description	Privileges Required
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.

Table 27-1 (Cont.) Database Privileges Needed for Operations on Oracle XML DB Resources

See Also:

Upgrade or Downgrade of an Existing Oracle XML DB Installation for information about treatment of database access privileges when upgrading

Privileges

The privileges provided with Oracle Database include the standard WebDAV privileges as well as Oracle-specific privileges.

The standard WebDAV privileges use the WebDAV namespace DAV:¹. The Oracle-specific privileges use the Oracle XML DB ACL namespace, http://xmlns.oracle.com/xdb/acl.xsd, which has the predefined prefix xdb.

Atomic Privileges

An atomic privilege does not include other privileges. The available atomic privileges for repository operations are described, and their database counterparts are listed.

Aggregate Privileges

An Aggregate privilege contains other privileges, atomic or aggregate. The available aggregate privileges for repository operations are listed together with their component atomic privileges.

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

An atomic privilege does not include other privileges. The available atomic privileges for repository operations are described, and their database counterparts are listed.

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 day, but this is only conventional. day is not a predefined prefix for Oracle XML DB.

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
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

An Aggregate privilege contains other privileges, atomic or aggregate. The available aggregate privileges for repository operations are listed together with their component atomic 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	<pre>xdb:write-acl-ref, xdb:update-acl</pre>

Table 27-3 Aggregate Privileges



Table 27-3 (Cont.) Aggregate Privileges

Aggregate Privilege	Component Atomic Privileges	
DAV::update	DAV::write-content,DAV::write-properties	
xdb:update	DAV::write-properties, DAV::write-content	

ACLs and ACEs

ACLs are used to protect resources, which in the case of Oracle Database are resources (files and folders) in Oracle XML DB Repository. ACLs are composed of 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.

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 schema (user account) 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:

- acl.xsd: XML Schema for ACLs for the ACL XML schema
- Oracle Database Security Guide



• System ACLs

The system ACLs, which are predefined and supplied with Oracle Database, are described.

ACL and ACE Evaluation

Privileges are checked before a principal is allowed to access a repository resource that is protected by 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.

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.

• Element invert: Complement the Principals in an ACE

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.

System ACLs

The system ACLs, which are predefined and supplied with Oracle Database, are described.

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.

Related Topics

Local and Global XML Schemas

An XML schema can be registered as local (visible only to its owner, by default) or global (visible to all database users, by default).



ACL and ACE Evaluation

Privileges are checked before a principal is allowed to access a repository resource that is protected by 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-trumpsgrant. 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 aclevaluation-method, in configuration file xdbconfig.xml, to either ace-order Or deny-trumpsgrant. 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 11g 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 Upgrade or Downgrade of an Existing Oracle XML DB Installation.

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.

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.

Element invert: Complement the Principals in an ACE

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 27-3, 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 27-3 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>
```

Overview of 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 PL/SQL package DBMS_XDB_REPOS.

Those ACL 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.

- Creating an ACL Using DBMS_XDB_REPOS.CREATERESOURCE An example illustrates using DBMS_XDB_REPOS.createResource to create an ACL.
- Retrieving an ACL Document, Given its Repository Path An example shows how to retrieve an ACL document, given its location in Oracle XML DB Repository.
- Setting the ACL of a Resource An example uses PL/SQL procedure DBMS_XDB_REPOS.setACL to set the ACL of a resource.
- Deleting an ACL An example uses PL/SQL procedure DBMS_XDB_REPOS.deleteResource to delete an 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/XML function XMLQuery with XQuery Update to manipulate ACLs. You must COMMIT after making any ACL changes.
- Retrieving the ACL Document that Protects a Given Resource An example illustrates how to use PL/SQL function DBMS_XDB_REPOS.getACLDocument to retrieve the ACL document that protects a given resource.
- Retrieving Privileges Granted to the Current User for a Particular Resource An example illustrates how to use PL/SQL function DBMS_XDB_REPOS.getPrivileges to retrieve privileges granted to the current user.
- Checking Whether the Current User Has Privileges on a Resource An example illustrates how to use PL/SQL function DBMS_XDB_REPOS.checkPrivileges to check whether the current user has a given set of privileges on a resource. The function returns a nonzero value if the user has the privileges.
- Checking Whether a User Has Privileges Using the ACL and Resource Owner Function DBMS_XDB_REPOS.ACLCheckPrivileges is typically used by applications that must perform ACL evaluation on their own, before allowing a user to perform an operation.
- Retrieving the Path of the ACL that Protects a Given Resource An example uses a RESOURCE_VIEW query to retrieve the path of the ACL that protects a given resource. The query uses the fact that the XMLRef and ACLOID elements of a resource form the link between an ACL and a resource.
- Retrieving the Paths of All Resources Protected by a Given ACL An example retrieves the paths of all resources protected by a given ACL.

Related Topics

 Administration of Oracle XML DB Administration of Oracle XML DB includes installing, upgrading, and configuring it.

Creating an ACL Using DBMS_XDB_REPOS.CREATERESOURCE

An example illustrates using DBMS_XDB_REPOS.createResource to create an ACL.

Example 27-4 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.

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.

Example 27-4 Creating an ACL Using CREATERESOURCE



```
'<acl description="myacl"</pre>
               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;
```

Retrieving an ACL Document, Given its Repository Path

An example shows how to retrieve an ACL document, given its location in Oracle XML DB Repository.

Example 27-5 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

An example uses PL/SQL procedure DBMS XDB REPOS.setACL to set the ACL of a resource.

Example 27-6 creates resource /TESTUSER/pol.xml and sets its ACL to /TESTUSER/acl1.xml using PL/SQL procedure DBMS_XDB_REPOS.setACL.

Example 27-6 Setting the ACL of a Resource

Deleting an ACL

An example uses PL/SQL procedure DBMS_XDB_REPOS.deleteResource to delete an ACL.

Example 27-7 deletes the ACL created in Example 27-4.

If a resource is being protected by an ACL that you want to delete, change the ACL of that resource before deleting the ACL.

Example 27-7 Deleting an ACL

CALL DBMS XDB REPOS REPOS.deleteResource('/TESTUSER/acl1.xml');

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/XML function XMLQuery with XQuery Update 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 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 Repository Access Using Protocols. You can update an ACL or an access control entry (ACE) using RESOURCE VIEW.

Example 27-8 uses SQL/XML function XMLQuery together with XQuery Update 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 27-9 uses XQuery Update to append an ACE to an existing ACL. The ACE gives privileges read-properties and read-contents to user HR.

Example 27-10 uses XQuery Update to delete an ACE from an ACL. The first ACE is deleted.

Example 27-8 Updating (Replacing) an Access Control List

```
UPDATE RESOURCE VIEW r
SET r.RES =
  XMLQuery(
     'declare namespace r="http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
     declare namespace a="http://xmlns.oracle.com/xdb/acl.xsd"; (: :)
     copy $i := $p1 modify
        (for $j in $i/r:Resource/r:Contents/a:acl
         return replace node $j with $p2)
      return $i'
     PASSING r.RES AS "p1",
             '<acl description="myacl"</pre>
                   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>' AS "p2"
    RETURNING CONTENT)
 WHERE equals path(r.RES, '/TESTUSER/acl1.xml') = 1;
```

Example 27-9 Appending ACEs to an Access Control List

```
UPDATE RESOURCE VIEW r
  SET r.RES =
    XMLQuery('declare namespace r="http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
              declare namespace a="http://xmlns.oracle.com/xdb/acl.xsd"; (: :)
              copy $i := $p1 modify
                (for $j in $i/r:Resource/r:Contents/a:acl
                 return insert nodes $p2 as last into $j)
              return $i'
             PASSING r.RES AS "p1",
                     XMLType('<ace xmlns="http://xmlns.oracle.com/xdb/acl.xsd">
                                 <grant>true</grant>
                                 <principal>HR</principal>
                                 <privilege>
                                   <read-properties/>
                                   <read-contents/>
                                 </privilege>
                              </ace>') as "p2"
             RETURNING CONTENT)
  WHERE equals path(r.RES, '/TESTUSER/acl1.xml') = 1;
```

Example 27-10 Deleting an ACE from an Access Control List

```
copy $i := $p modify delete nodes $i/r:Resource/r:Contents/a:acl/a:ace[1]
return $i'
PASSING r.RES AS "p" RETURNING CONTENT)
WHERE equals_path(r.RES, '/TESTUSER/acl1.xml') = 1;
```

Retrieving the ACL Document that Protects a Given Resource

An example illustrates how to use PL/SQL function DBMS_XDB_REPOS.getACLDocument to retrieve the ACL document that protects a given resource.



Example 27-11 Retrieving the ACL Document for a Resource

```
SELECT XMLSerialize(DOCUMENT DBMS XDB REPOS.getACLDocument('/TESTUSER/po1.xml')
                  AS CLOB)
 FROM DUAL;
XMLSERIALIZE (DOCUMENTDBMS XDB REPOS.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.
```

Retrieving Privileges Granted to the Current User for a Particular Resource

An example illustrates how to use PL/SQL function DBMS_XDB_REPOS.getPrivileges to retrieve privileges granted to the current user.

Example 27-12 Retrieving Privileges Granted to the Current User for a Particular Resource

```
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

An example illustrates how to use PL/SQL function DBMS_XDB_REPOS.checkPrivileges to check whether the current user has a given set of privileges on a resource. The function returns a nonzero value if the user has the privileges.

Example 27-13 checks to see if the access privileges read-contents and read-properties have been granted to the current user on resource /TESTUSER/pol.xml. The positive-integer return value shows that they have.

Example 27-13 Checking If a User Has a Certain Privileges on a Resource

```
SELECT DBMS XDB REPOS.checkPrivileges(
        '/TESTUSER/pol.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 REPOS.CHECKPRIVILEGES('/TESTUSER/PO1.XML',
       -----
                                               1
1 row selected.
```



Checking Whether a User Has Privileges Using the ACL and Resource Owner

Function DBMS_XDB_REPOS.ACLCheckPrivileges is typically used by applications that must perform ACL evaluation on their own, before allowing a user to perform an operation.

Example 27-14 checks whether the ACL /TESTUSER/acl1.xml grants the privileges readcontents 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 27-14 Checking User Privileges Using ACLCheckPrivileges

```
CONNECT sh
Enter password: <password>
Connected.
SELECT DBMS XDB REPOS.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 REPOS.ACLCHECKPRIVILEGES ('/TESTUSER/ACL1.XML', 'TESTUSER',
    _____
                                                           0
```

1 row selected.

Retrieving the Path of the ACL that Protects a Given Resource

An example uses a RESOURCE_VIEW query to retrieve the path of the ACL that protects a given resource. The query uses the fact that the XMLRef and ACLOID elements of a resource form the link between an ACL and a resource.

Example 27-15 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 Example 27-15, make_ref returns a REF to the row of table XDB\$ACL whose OID is the / Resource/ACLOID for the resource /TESTUSER/pol.xml. The inner query returns the ACLOID of the resource. The outer query returns the path to the corresponding ACL.



Example 27-15 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
```

Retrieving the Paths of All Resources Protected by a Given ACL

An example retrieves the paths of all resources protected by a given ACL.

Example 27-16 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.

Example 27-16 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/pol.xml
1 row selected.
```

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 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). 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

A uniform security mechanism for accessing REF-based repository resources is provided by enabling hierarchy on the tables used to store them. When ACL-based security is not needed for particular resources, you can optimize their access by using PL/SQL procedure DBMS_XDBZ.disable_hierarchy to turn off ACL checking.

Resources in Oracle XML DB Repository are of two types:

- LOB-based (the 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 (the content is XML data 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 lastmodified 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.

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 DBMS_XDBZ.disable_hierarchy 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:



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 DBMS_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

Integration Of Oracle XML DB with LDAP

You can allow Lightweight Directory Access Protocol (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 user name 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 (also known as LDAP roles) 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 Database Security Guide for information about setting up SSL authentication Example 27-17 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 27-18 shows an ACL for an LDAP group. Element <principal> contains the full distinguished name of the LDAP group.

Example 27-17 ACL Referencing an LDAP User

Example 27-18 ACL Referencing an LDAP Group

You can access Oracle XML DB Repository data using protocols FTP and HTTP(S)/WebDAV.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

Overview of Oracle XML DB Protocol Server

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. HTTPS provides *secure* access to Oracle XML DB Repository.

Oracle XML DB Protocol Server Configuration Management

Oracle XML DB protocol server uses configuration parameters stored in file xdbconfig.xml to initialize its startup state and manage session level configuration. 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.

FTP and the Oracle XML DB Protocol Server
 File Transfer Protocol (FTP), specified in RFC959, is one of the oldest and most popular protocols. It provides access to heterogeneous file systems in a uniform manner. You can use FTP to access and update data stored in Oracle XML DB Repository.

 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.

• WebDAV and Oracle XML DB

Web Distributed Authoring and Versioning (WebDAV) is an IETF standard protocol that Oracle XML DB uses 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.

Overview of Oracle XML DB Protocol Server

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. 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_REPOS to access Oracle Automatic Storage Management (Oracle ASM) files and folders in the repository virtual folder /sys/asm.

As described in Getting Started with Oracle XML DB and Access to 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.



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

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 the Oracle XML DB configuration file, xdbconfig.xml, 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.

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



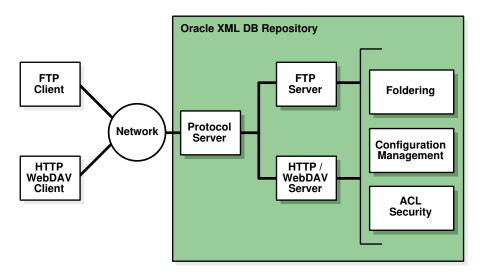


Figure 28-1 Oracle XML DB Architecture: Protocol Server

Related Topics

Guidelines for Oracle XML DB Applications in Java
 Design guidelines are presented for writing Oracle XML DB applications in Java. This
 includes guidelines for writing and configuring Java servlets for Oracle XML DB.

Oracle XML DB Protocol Server Configuration Management

Oracle XML DB protocol server uses configuration parameters stored in file xdbconfig.xml to initialize its startup state and manage session level configuration. 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.

Protocol Server Configuration Parameters

The Oracle XML DB protocol configuration parameters are described. They include those common to all protocols, those specific to FTP, and those specific to HTTP(S)/WebDAV.

- Configuring Secure HTTP (HTTPS)
 To enable the 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. The DBA must then stop and restart the database and the listener.
- Using Listener Status to Check Port Configuration You can use the TNS Listener command, lsnrctl status, to verify that HTTP(S) and FTP support has been enabled. An example illustrates this.
- Configuring Protocol Port Parameters after Database Consolidation

In a multitenant container database (CDB), protocol server port numbers distinguish the plugged-in pluggable databases (PDBs): each such database must have unique port numbers. A database administrator (DBA) must ensure that each port number used by a PDB is unique.

- Configuration and Management of Authentication Mechanisms for HTTP You configure the authentication mechanisms to allow for HTTP access to Oracle XML DB Repository by setting element authentication, a child of element httpconfig, in configuration file xdbconfig.xml.
- Oracle XML DB Repository and File-System Resources
 IETF 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, ACL based security, and an ability to store and retrieve any content.
- Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents Oracle XML DB protocol server always checks whether a document being inserted is based on an XML schema that is registered with Oracle XML DB Repository. If it is, then the XMLType storage model to use is determined by that XML schema. If it is not, then the document is stored as a BLOB.
- Event-Based Logging You can log the requests received and responses sent by a protocol server by setting event number 31098 to level 2.
- Auditing of HTTP and FTP Protocols You can use SQL statement CREATE AUDIT POLICY to audit Oracle XML DB HTTP and FTP protocol messages.

Related Topics

Configuration of 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.

Protocol Server Configuration Parameters

The Oracle XML DB protocol configuration parameters are described. They include those common to all protocols, those specific to FTP, and those specific to HTTP(S)/WebDAV.

Table 28-1 shows the parameters common to all protocols. All of their parameter names, 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

For examples of the usage of these parameters, see the configuration file, xdbconfig.xml.



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.	
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 28-1 Common Protocol Configuration Parameters



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.
ftp-port	Port on which FTP server listens. By default, this is 0, which means that FTP is <i>disabled</i> . 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 ${\tt tcp}.$
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."
host-name	Name used to access the host system. The value can be an IP address or a name that is mapped to an IP address using <i>host naming</i> (e.g., in file /etc/hosts on Linux) — see <i>Oracle Database Net Services Reference</i> . By default, the IP address returned by the operating system is used.
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 28-2 Configuration Parameters Specific to FTP

Table 28-3 Configuration Parameters Specific to HTTP(S)/WebDAV (Except Servlet)

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 <i>disabled</i> . If this parameter is empty (<http-port></http-port>), then the default value of 0 applies. An empty parameter is <i>not</i> recommended.
	This parameter <i>must</i> 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 http://www.ntminut.com ,
	This parameter is <i>optional</i> , 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></http2-port>) also raises an error.



Parameter	Description
http-protocol	Protocol over which the HTTP(S)/WebDAV server runs on port http-port. Must be either TCP or TCPS
	This parameter <i>must</i> be present. Otherwise, validation of xdbconfig.xml against XML schema xdbconfig.xsd fails. An empty parameter (<http- protocol/>) also raises an error.</http-
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 <i>not</i> recommended.)</http2-
	This parameter is <i>optional</i> , 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 the 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 <i>blocked</i> . See Anonymous Access to Oracle XML DB Repository Using HTTP.
authentication	The HTTP authentication mechanisms allowed. See Configuration and Management of Authentication Mechanisms for HTTP
expire	HTTP header that specifies the expiration date and time for a URL. See Control of URL Expiration Time.

Table 28-3 (Cont.) Configuration Parameters Specific to HTTP(S)/WebDAV (ExceptServlet)



Note:

Oracle recommends that you use the subprograms of PL/SQL package DBMS_XDB_CONFIG to set or change FTP or HTTP port numbers. Do *not* set ports by directly editing configuration file xdbconfig.xml

Related Topics

xdbconfig.xsd: XML Schema for Configuring Oracle XML DB
 A full listing is presented of file xdbconfig.xsd, which contains the XML schema used to
 configure Oracle XML DB.

See Also:

- Administration of Oracle XML DB for more information about the configuration file xdbconfig.xml
- Configuration of Mappings from Default Namespace to Schema Location for more information about the schemaLocation-mappings parameter
- Configuration of XML File Extensions for more information about the xmlextensions parameter

Configuring Secure HTTP (HTTPS)

To enable the 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. The DBA must then stop and restart the database and the listener.

Enabling the HTTP Listener to Use SSL

To configure the HTTP Listener for SSL, a database administrator (DBA) must create a wallet for the server and import a certificate; specify the wallet location to the server; disable client authentication; add an SSL_DH_anon cipher suite to SSL_CIPHER_SUITES; and create a listening end point that uses TCP/IP with SSL.

• Enabling TCPS Dispatcher To enable launching of a TCPS dispatcher during database startup, a database administrator (DBA) must edit the database pfile.

Related Topics

Configuration of 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.



Enabling the HTTP Listener to Use SSL

To configure the HTTP Listener for SSL, a database administrator (DBA) must create a wallet for the server and import a certificate; specify the wallet location to the server; disable client authentication; add an SSL_DH_anon cipher suite to SSL_CIPHER_SUITES; and create a listening end point that uses TCP/IP with SSL.

More precisely, a DBA must carry out the following steps to configure the HTTP Listener for SSL.

- 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 Enterprise User 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. Add an SSL_DH_anon cipher suite to SSL_CIPHER_SUITES Use any of these:
 - SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
 - SSL_DH_anon_WITH_RC4_128_MD5
 - SSL_DH_anon_WITH_DES_CBC_SHA

This step updates file sqlnet.ora.

5. 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 Security Guide for detailed information regarding steps 1 through 5



Enabling TCPS Dispatcher

To enable launching of a TCPS dispatcher during database startup, a database administrator (DBA) must edit the database pfile.

The following line must be added to the pfile, 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

Using Listener Status to Check Port Configuration

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

Example 28-1 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

```
_____
                             LISTENER
Alias
Version
                             TNSLSNR for 32-bit Windows: Version 11.1.0.5.0 - Beta
Start Date
                             20-JUN-2007 15:35:40
                             0 days 16 hr. 47 min. 42 sec
Uptime
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
```

Configuring Protocol Port Parameters after Database Consolidation

In a multitenant container database (CDB), protocol server port numbers distinguish the plugged-in pluggable databases (PDBs): each such database must have unique port numbers. A database administrator (DBA) must ensure that each port number used by a PDB is unique.

To avoid port conflicts and to resolve any port conflicts that might result from consolidation, a database administrator must proceed as follows:

- 1. Use PL/SQL function DBMS_XDB_CONFIG.usedPort to obtain the port numbers used by the other PDBs in the same CDB.
- Use PL/SQL subprogram DBMS_XDB_CONFIG.setFTPPort or DBMS_XDB_CONFIG.setHTTPPort, as needed, to change each port number that conflicts so that it is unique.

Note:

After a PDB plug-in or clone operation, and until Oracle XML DB has been fully initialized, the port parameter settings in the configuration file, <code>xdbconfig.xml</code>, might be *undefined*. Oracle recommends that you use <code>DBMS_XDB_CONFIG</code> subprogram getFTPPort, getHTTPPort, getHTTPSPort, setFTPPort, setHTTPPort, or setHTTPSPort to obtain or modify a port value.

Configuration and Management of Authentication Mechanisms for HTTP

You configure the authentication mechanisms to allow for HTTP access to Oracle XML DB Repository by setting element authentication, a child of element httpconfig, in configuration file xdbconfig.xml.

Starting with 12c Release 1 (12.1.0.1), Oracle Database supports not only basic authentication but also digest access authentication.

User credentials are case-sensitive. In particular, to be authenticated, a user name must exactly match the name as it was created (which by default is all uppercase).

Digest access authentication, also known as **digest authentication** provides encryption of user credentials (name, password, etc.) without the overhead of complete data encryption.

Note:

By default, when a user is created digest authentication is *not* enabled for the new user. To enable digest authentication when creating a user, use clause DIGEST ENABLE with SQL statement CREATE USER, specifying the password for the user.

You can also enable digest authentication for an existing user. To do that, use DIGEST ENABLE with SQL statement ALTER USER. This initializes the user password for digest authentication, but it does not directly enable digest authentication. Digest authentication is enabled for the user when the user next logs in with the password.

You can configure the authentication mechanism to use by setting element authentication, a child of element httpconfig, in configuration file xdbconfig.xml. Element authentication is optional. If absent then only basic authentication is used.

Element authentication has two possible child elements:

- Element allow-mechanism specifies an allowed mechanism: basic, digest, custom, digestMD5, digestSHA256, or digestSHA512. Use a separate allow-mechanism element to specify each mechanism you want to allow.
- Element digest-auth is optional. It specifies information for a digest mechanism. Its child element nonce-timeout specifies the number of seconds that a given nonce remains valid. The default value is 300 seconds.

The default value is used if there is an allow-mechanism that specifies digest but there is no digest-auth element. A digest-auth element is ignored if there is no allow-mechanism that specifies digest.

HTTP requests are accepted for each allow-mechanism specified. Authentication challenges are presented in the order of the specified allow-mechanism types. For example, if both digest and basic are present, in that order, then a digest challenge is presented before a basic challenge. Oracle recommends that you *always* put a stronger authentication before a weaker one. (Digest authentication is stronger than basic authentication.)

Nonces for Digest Authentication

With digest authentication, the server generates a nonce whenever it issues an unauthorized response. Clients include the nonce in requests to the server. The server checks nonces received from the client to see if it needs to refuse the client authentication. A client can authenticate the server the same way.

💉 See Also:

- Configuration of Oracle XML DB Using xdbconfig.xml
- Upgrade or Downgrade of an Existing Oracle XML DB Installation for installation, upgrade, and downgrade considerations
- HTTP Authentication: Basic and Digest Access Authentication, IETF RFC2617
- Oracle Database SQL Language Reference for information about SQL statement CREATE USER
- Oracle Database SQL Language Reference for information about SQL statement
 ALTER USER

Nonces for Digest Authentication

With digest authentication, the server generates a nonce whenever it issues an unauthorized response. Clients include the nonce in requests to the server. The server checks nonces received from the client to see if it needs to refuse the client authentication. A client can authenticate the server the same way.

A **nonce** is a unique string that the server generates each time it issues an HTTP 401 (unauthorized) response. Clients include the nonce in subsequent requests that they issue to the server. The server checks the nonce it receives from the client. If incorrect or if the noncetimeout period has expired, the server can immediately refuse to authenticate.



(A client can use the same mechanism to authenticate the server: it can generate its own nonce. Both client and server can use this client nonce to help prevent particular plain-text attacks.)

A new nonce is created each time the server sends a digest challenge to a client. A nonce is based on a **nonce key**. The initial nonce key is generated randomly when you install or upgrade the database.

If you use digest authentication then Oracle also recommends that you create a new nonce key periodically, to ensure the integrity of the key. You use PL/SQL procedure DBMS_XDB_ADMIN.createNonceKey to do this.

Oracle XML DB Repository and File-System Resources

IETF 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, ACL-based security, and an ability to store and retrieve any content.

The repository can store binary data input through FTP and XML schema-based documents.

See Also:

- FTP Protocol Specification, IETF RFC959
- HTTP Protocol Specification, IETF RFC2616
- WebDAV Protocol Specification, RFC2518

Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents

Oracle XML DB protocol server always checks whether a document being inserted is based on an XML schema that is registered with Oracle XML DB Repository. If it is, then the XMLType storage model to use is determined by that XML schema. If it is not, then the document is stored as a BLOB.

Event-Based Logging

You can log the requests received and responses sent by a protocol server 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"

Auditing of HTTP and FTP Protocols

You can use SQL statement CREATE AUDIT POLICY to audit Oracle XML DB HTTP and FTP protocol messages.



You can audit all or failed HTTP messages, 401 AUTH HTTP return-code messages, and all or failed FTP messages. Columns with name prefix PROTOCOL_ of data dictionary view UNIFIED AUDIT TRAIL capture the audit result.

Be aware that a unified audit policy for HTTP and FTP protocols can affect performance.

See Also: Oracle Database Security Guide

FTP and the Oracle XML DB Protocol Server

File Transfer Protocol (FTP), specified in RFC959, is one of the oldest and most popular protocols. It provides access to heterogeneous file systems in a uniform manner. You can use FTP to access and update data stored in Oracle XML DB Repository.

Oracle XML DB Protocol Server: FTP Features

File Transfer Protocol (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.

Oracle XML DB Protocol Server: FTP Features

File Transfer Protocol (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.

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), by contrast, commands and data are transferred using a single connection.

Note:

For security reasons, FTP is *disabled*, by default, for Oracle Database. 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.

- FTP Features That Are Not Supported FTP features that are not supported by Oracle XML DB include record-oriented files and operations append, allocate, account, and abort.
- Supported FTP Client Methods
 Oracle XML DB supports several FTP client methods for access to Oracle XML DB Repository.
- FTP Quote Methods
 Oracle Database supports several FTP quote methods, which provide information directly to Oracle XML DB.



- Uploading Content to Oracle XML DB Repository Using FTP An example shows the commands issued and the output generated when a standard command line FTP tool loads documents into Oracle XML DB Repository:
- 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. You can use protocols and resource APIs to access Oracle ASM files in
 repository virtual folder /sys/asm. All files in /sys/asm are binary.
- 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 non-standard, unprotected port.
- 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).
- FTP Server Session Management Oracle XML DB protocol server provides session management for FTP. 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.
- Handling Error 421. Modifying the Default Timeout Value of an FTP Session
 If you are frequently disconnected from the server and you must reconnect and traverse
 the entire directory before performing the next operation, then you might need to modify
 the default timeout value for FTP sessions. If the session is idle for more than this period, it
 is disconnected.
- 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.

Related Topics

Configuration of 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.

🖍 See Also:

FTP Protocol Specification, IETF RFC 959

FTP Features That Are Not Supported

FTP features that are not supported by Oracle XML DB include record-oriented files and operations append, allocate, account, and abort.

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

Oracle XML DB supports several FTP client methods for access to Oracle XML DB Repository.

- 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 for an example of using FTP method proxy

FTP Quote Methods

Oracle Database supports several FTP ${\tt quote}$ 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:

- * 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.

Uploading Content to Oracle XML DB Repository Using FTP

An example shows the commands issued and the output generated when a standard command line FTP tool loads documents into Oracle XML DB Repository:

The key point demonstrated by Figure 28-3 and Example 28-2 is that neither Windows Explorer nor an FTP tool is aware that it is working with Oracle XML DB. Since such tools and Oracle XML DB both support open Internet protocols they work with each other out of the box.

¹ BOM is a Unicode-standard signature that indicates the order of the stream of bytes that follows it.

Any tool that understands the WebDAV or FTP protocol can be used to create content managed by Oracle XML DB Repository. No additional software needs to be installed on the client or the mid-tier.

When the contents of folders are viewed using a tool such as Windows Explorer or FTP, the lengths of any XML Schema-based documents contained in the folder are shown as zero (0) bytes. This was designed as such for two reasons:

- It is not clear what the size of a 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.

Example 28-2 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 OUINE 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 OUINE 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 OUINE 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: -1
959 bytes received in 0.0027 seconds (349.45 Kbytes/s)
ftp> cd "..."
250 CWD Command successful
. . . .
ftp> quit
221 QUIT Goodbye.
```

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. You can use protocols and resource APIs to access Oracle ASM files in 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 28-3 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 Access to Oracle XML DB Repository Data. In Example 28-3, 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.

In Example 28-3, after connecting to and logging onto database <code>myhost</code> (first four lines), FTP methods <code>cd</code> and <code>ls</code> are used to navigate and list folders, respectively. When in folder /sys/asm/DATA/dbs, FTP command <code>get</code> is used to copy files <code>t_dbl.f</code> and <code>t_ax1.f</code> to the current folder of the local file system. Then, FTP command <code>put</code> is used to copy file <code>my_db2.f</code> 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 28-4 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 28-4 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. 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.



In Example 28-4:

- 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.

Example 28-3 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

Example 28-4 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
```

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 non-standard, unprotected port.

To use FTP on the standard port, 21, your database administrator must do the following:

 (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

 (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 (a), to execute the second command in the background. Do not use lsnrctl start to start the listener.

 Use PL/SQL procedure DBMS_XDB_CONFIG.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_CONFIG.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 28-5 shows how to make an FTP connection with the IPv6 address 2001::0db8:ffff:ffff.

See Also:

Oracle Database Net Services Reference for information about IPv6

Example 28-5 FTP Connection Using IPv6

```
ftp> open 2001::0db8:ffff:ffff 1521
Connected to 2001::0db8: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:(none)): username
331 pass required for USERNAME
Password: password-for-username
230 USERNAME logged in
ftp>
```

FTP Server Session Management

Oracle XML DB protocol server provides session management for FTP. 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.

Related Topics

Configuration of 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.

Handling Error 421. Modifying the Default Timeout Value of an FTP Session

If you are frequently disconnected from the server and you must reconnect and traverse the entire directory before performing the next operation, then you might need to modify the default

timeout value for FTP sessions. If the session is idle for more than this period, it is disconnected.

You can increase the timeout value (default = 6000 centiseconds) by modifying the configuration document as follows and then restarting the database:

Example 28-6 Modifying the Default Timeout Value of an FTP Session

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.

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 of the Oracle XML DB protocol server
 implements the IETF RFC2616 specification with the *exception* of a few optional features.

Oracle XML DB Protocol Server: HTTP(S) Features

The Oracle XML DB HTTP(S) component of the Oracle XML DB protocol server implements the IETF RFC2616 specification with the *exception* of a few optional features.

These are the optional HTTP(S) features that are not supported:

- 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
- Supported HTTP(S) Client Methods Oracle XML DB supports several HTTP(S) client methods for access to Oracle XML DB Repository.
- 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 non-standard, unprotected port.
- Use of 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 ([]).
- HTTPS: Support for Secure HTTP If properly configured, you can access Oracle XML DB Repository in a secure fashion, using HTTPS.
- Control of 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.

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*.

• Use of Java Servlets with HTTP(S)

Oracle XML DB supports the use of Java servlets. Each must each 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. The servlet name must be associated with a pattern.

Embedded PL/SQL Gateway

You can use the embedded PL/SQL gateway to implement a Web application entirely in PL/SQL. It runs in the Oracle XML DB HTTP listener.

- Transmission of Multibyte Data From a Client
 When a client sends multibyte data in a URL, RFC 2718 specifies that the client should
 use the %HH format, where HH is the hexadecimal notation of the byte value in UTF-8
 encoding.
- Characters That Are Not ASCII in URLs
 Convert non-ASCII characters that appear in URLs passed to an HTTP server to UTF-8 and escape them using the %HH format, where HH is the hexadecimal notation of the byte value.
- Character Sets for HTTP(S)
 You can control the character sets used for data that is transferred using HTTP(S).

See Also:

HTTP 1.1 Protocol Specification, IETF RFC 2616

Supported HTTP(S) Client Methods

Oracle XML DB supports several HTTP(S) client methods for access to Oracle XML DB Repository.

- 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 are in accordance with WebDAV. Servlets and Web services may support additional HTTP(S) methods, such as POST.

See Also: WebDAV Client Methods Supported by Oracle XML DB 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 non-standard, unprotected port.

To use HTTP or HTTPS on a standard port (80 for HTTP, 443 for HTTPS), your database administrator must do the following:

 (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_CONFIG.setHTTPPort 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_CONFIG.setHTTPPort(port_number);

 Force the database to reregister with the listener, using this SQL statement: SQL> ALTER SYSTEM REGISTER;

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

Use of 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:fff]:8080/
```

🖍 See Also:

Oracle Database Net Services Administrator's Guide 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) for configuration information.

Note:

Oracle recommends that you use *digest* authentication for WebDAV access to Oracle XML DB Repository. Digest authentication is supported starting with Oracle Database 12c Release 1 (12.1.0.1). If your database is installed on Microsoft Windows and you cannot use digest authentication then see WebDAV and Microsoft Windows for information about configuring *basic* authentication.



Control of 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.

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.

See Also:

Alternate Interval Syntax for the alternate syntax for mod_expires directive ExpiresDefault

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.



Note:

If user account ANONYMOUS is *locked* for a multitenant container database (CDB) then locking or unlocking ANONYMOUS for a pluggable database (PDB) plugged into that CDB has *no effect* on access by ANONYMOUS to the PDB.

See Also:

- Table 28-3 for information about parameter allow-repository-anonymousaccess
- Configuration of Oracle XML DB Using xdbconfig.xml for information about configuring Oracle XML DB parameters
- Configuration of Oracle XML DB Servlets for information about parameter
 security-role-ref

Use of Java Servlets with HTTP(S)

Oracle XML DB supports the use of Java servlets. Each must each 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. The servlet name must be associated with a pattern.

The pattern 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 against the registered patterns. If there is a match then the protocol server invokes the corresponding servlet with the appropriate initialization parameters. The Java Virtual Machine (JVM) is started, and it invokes a Java method to initialize the servlet, create response and request objects, pass these on to the servlet, and run the servlet.

Related Topics

Guidelines for Oracle XML DB Applications in Java
 Design guidelines are presented for writing Oracle XML DB applications in Java. This
 includes guidelines for writing and configuring Java servlets for Oracle XML DB.

Embedded PL/SQL Gateway

You can use the embedded PL/SQL gateway to implement a Web application entirely in PL/SQL. It runs in the Oracle XML DB HTTP listener.

With the embedded PL/SQL gateway, 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.

Note:

If you are currently using *mod_plsql*, which is a plug-in of *Oracle HTTP Server* that lets you invoke PL/SQL stored procedures using HTTP(S), Oracle recommends that you migrate to using the embedded PL/SQL gateway instead.

See Also:

- Oracle Database Development Guide for information about configuring and using the embedded PL/SQL gateway
- Administration of Oracle XML DB for information on the configuration definition of the embedded gateway in xdbconfig.xml

Transmission of Multibyte Data From a Client

When a client sends multibyte data in a URL, RFC 2718 specifies that the client should use the <code>%HH</code> format, where <code>HH</code> 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.

Related Topics

Configuration of 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.



See Also:

RFC 2616, HTTP 1.1 Protocol Specification, HTTP Protocol Specification, IETF RFC2616

Characters That Are Not ASCII in URLs

Convert non-ASCII characters that appear in URLs passed to an HTTP server to UTF-8 and escape them using 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

Related Topics

 Configuration of 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.

Character Sets for HTTP(S)

You can control the character sets used for data that is transferred using HTTP(S).

- HTTP(S) Request Character Set The character set of an HTTP(S) request body is determined using a straightforward but somewhat complex algorithm.
- HTTP(S) Response Character Set The response generated by the Oracle XML DB HTTP server is in a character set specified in the Accept-Charset field of the request.

HTTP(S) Request Character Set

The character set of an HTTP(S) request body is determined using a straightforward but somewhat complex algorithm.

- **1.** The Content-Type header is evaluated. If the Content-Type header specifies a charset value, the specified charset is used.
- 2. The MIME type of the document is evaluated as follows:
 - a. If the MIME type is "*/xml" then the character set is determined as follows:
 - i. If neither a BOM nor an encoding declaration is present then UTF-8 is used.
 - ii. If a BOM is present then UTF-16 is used.
 - iii. If an encoding declaration is present then the specified encoding is used.



- **b.** If the MIME type is text then ISO8859-1 is used.
- c. If the MIME type is neither "*/xml" nor text then 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*.

HTTP(S) Response Character Set

The response generated by the Oracle XML DB HTTP server is in a character set specified in the Accept-Charset field of the request.

Accept-Charset can specify a list of character sets. Based on the q-value, Oracle XML DB chooses one of them that does not require conversion. This might not necessarily be the character set with the highest q-value. If Oracle XML DB cannot find one that does not require conversion, then the conversion used is based on the highest q-value.

WebDAV and Oracle XML DB

Web Distributed Authoring and Versioning (WebDAV) is an IETF standard protocol that Oracle XML DB uses 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 foldering and access-control features of WebDAV. Foldering is specified by RFC2518.

- WebDAV and Microsoft Windows
 For Microsoft Windows (XP with Service Pack 2 SP2 or later Windows system), use digest
 authentication for WebDAV access to Oracle XML DB Repository, if possible. If not, you
 must make appropriate modifications to the Windows XP Registry in order to use basic
 authentication.
- Creating a WebFolder in Microsoft Windows For Use With Oracle XML DB Repository Create a WebFolder in Windows 2000 and use it with Oracle XML DB Repository.

Oracle XML DB WebDAV Features

Oracle XML DB supports the foldering and access-control features of WebDAV. Foldering is specified by RFC2518.

WebDAV is a set of extensions to the HTTP(S) protocol that allow you to share, edit, and manage your files on remote Web servers.

WebDAV Features That Are Not Supported by Oracle XML DB

Oracle XML DB supports specification RFC2518, with the *exception* of a few features. For methods COPY, MOVE and DELETE it also supports the binding of resources as described in specification RFC5842.



 WebDAV Client Methods Supported by Oracle XML DB Oracle XML DB supports several HTTP(S)/WebDAV client methods for access to Oracle XML DB Repository.

See Also: RFC 2518: WebDAV Protocol Specification, WebDAV Protocol Specification, IETF RFC2518

WebDAV Features That Are Not Supported by Oracle XML DB

Oracle XML DB supports specification RFC2518, with the *exception* of a few features. For methods COPY, MOVE and DELETE it also supports the binding of resources as described in specification RFC5842.

These are the WebDAV features from RFC2518 that Oracle XML DB does not support:

- Using the name of a write-locked null resource (a lock-null resource) as a folder name is not supported, because it is represented as a zero-length resource in the file system. This is an optional feature.
- For method LOCK, you cannot specify infinity for the depth, to simultaneously lock a resource and all of its descendents. This feature is not optional, but it is not supported by Oracle XML DB.

In addition, for methods COPY, MOVE and DELETE Oracle XML DB supports the binding of resources as described in section 2 of RFC5842, "Binding Extensions to Web Distributed Authoring and Versioning (WebDAV)". A binding is a mapping of a URI to a resource, for a given folder.

See Also:

- WebDAV Protocol Specification, IETF RFC2518, Section 7.4 for information about lock-null resources
- *WebDAV Protocol Specification*, IETF RFC2518, Section 9.2 for information about depth-infinity locks
- Binding Extensions to Web Distributed Authoring and Versioning (WebDAV), IETF RFC5842, Section 2 for information about resource bindings

WebDAV Client Methods Supported by Oracle XML DB

Oracle XML DB supports several HTTP(S)/WebDAV client methods for access to Oracle XML DB Repository.

- 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)

Related Topics

Privileges

The privileges provided with Oracle Database include the standard WebDAV privileges as well as Oracle-specific privileges.

 Adding Metadata Using WebDAV PROPPATCH An alternative to using procedure DBMS_XDB_REPOS.appendResourceMetadata to add resource metadata is to use WebDAV method PROPPATCH.

See Also:

Supported HTTP(S) Client Methods for additional supported HTTP(S) client methods

WebDAV and Microsoft Windows

For Microsoft Windows (XP with *Service Pack 2* SP2 or later Windows system), use digest authentication for WebDAV access to Oracle XML DB Repository, if possible. If not, you must make appropriate modifications to the Windows XP Registry in order to use basic authentication.

Oracle Database supports digest authentication, starting with Oracle Database 12c Release 1 (12.1.0.1).

If you must use basic authentication then set Windows Registry key BasicAuthLevel to the value 1 or 2. Value 1 means use Secure Sockets Layer (SSL), which Oracle recommends. Value 2 means do not use SSL.

Related Topics

Configuring Secure HTTP (HTTPS)

To enable the repository to use *secure* HTTP connections (HTTPS), a database administrator (DBA) must configure the database accordingly: configure parameters <code>http2-port</code> and <code>http2-protocol</code>, enable the HTTP Listener to use SSL, and enable launching of the TCPS Dispatcher. The DBA must then stop and restart the database and the listener.

See Also:

WebDAV Redirector Registry Settings



Creating a WebFolder in Microsoft Windows For Use With Oracle XML DB Repository

Create a WebFolder in Windows 2000 and use it with Oracle XML DB Repository.

Create a WebFolder in Windows 2000:

- 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 you access any Windows folder.

Figure 28-2 Creating a WebFolder in Microsoft Windows

Add Network Place Wizar	i 🛛 🗡	:
	Welcome to the Add Network Place Wizard This wizard helps you add a link to a Network Place where you can store your documents. A Network Place can be a shared folder, a Web folder on the Internet, or an FTP site. Type the location of the Network Place: http://server1:8080 Browse Here are some examples. To continue, click Next.	
	< Back Next > Cancel	10 m

 Use of WebDAV with Windows Explorer to Copy Files into Oracle XML DB Repository You can use Windows Explorer to insert a folder from a local hard drive into Oracle Database.

Use of WebDAV with Windows Explorer to Copy Files into Oracle XML DB Repository

You can use Windows Explorer to insert a folder from a local hard drive into Oracle Database.



Figure 28-3 illustrates this.

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.

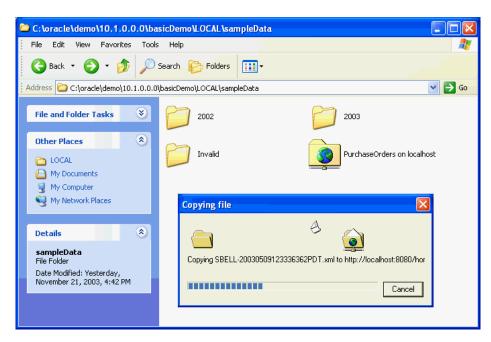


Figure 28-3 Copying Files into Oracle XML DB Repository



29

User-Defined Repository Metadata

You can create your own metadata to associate with XML data stored in Oracle XML DB Repository.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

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.

Using XML Schemas to Define Resource Metadata

Before you can add user metadata to photo resources, you must define the structure of such metadata using XML Schema. An XML schema is created and registered for each kind (technique, category) of photo resource metadata.

- Addition, Modification, and Deletion of Resource Metadata
 You can add, update, and delete user-defined resource metadata using PL/SQL
 procedures in package DBMS_XDB_REPOS, SQL DML statements INSERT, UPDATE, and
 DELETE, or WebDAV protocol method PROPPATCH.
- Querying XML Schema-Based Resource Metadata You can use metadata column RESID when querying resource metadata, to join the metadata with the associated data.
- XML Image Metadata from Binary Image Metadata Digital cameras include image metadata as part of the image files they produce.
- Adding Non-Schema-Based Resource Metadata
 You store user-defined resource metadata that is not XML Schema-based as a CLOB
 instance under the Resource element of the associated resource.
- PL/SQL Procedures Affecting Resource Metadata
 You can use PL/SQL procedures DBMS_XMLSCHEMA.registerSchema,
 DBMS_XDBZ.enable_hierarchy, DBMS_XDBZ.disable_hierarchy,
 DBMS_XDBZ.is_hierarchy_enabled, DBMS_XDB_REPOS.appendResourceMetadata,
 DBMS_XDB_REPOS.deleteResourceMetadata, DBMS_XDB_REPOS.purgeResourceMetadata,
 and DBMS_XDB_REPOS.updateResourceMetadata to perform resource metadata operations.

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

The term "metadata" is used in the context of XML in various ways, including XML Schema definitions, XML tags, and Oracle XML DB Repository resource information that supplements the resource content.

User-Defined Resource Metadata

User-defined resource metadata is itself represented as XML: it is XML data that is associated with other XML data, describing it or providing supplementary, related information.

Scenario: Metadata for a Photo Collection

A scenario used to illustrate the use of schema-based resource metadata 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.

Kinds of Metadata – Uses of the Term

The term "metadata" is used in the context of XML in various ways, including XML Schema definitions, XML tags, and Oracle XML DB Repository resource information that supplements the resource content.

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.

Related Topics

- 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.

User-Defined Resource Metadata

User-defined resource 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 as part of the resource document in the resource table, XDB.XDB\$RESOURCE.

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

A scenario used to illustrate the use of schema-based resource metadata 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.



Using XML Schemas to Define Resource Metadata

Before you can add user metadata to photo resources, you must define the structure of such metadata using XML Schema. An XML schema is created and registered for each kind (technique, category) of photo resource metadata.

See Also:

Scenario: Metadata for a Photo Collection for general information about the example user-defined metadata scenario

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.

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.

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:

By default, user metadata is stored object-relationally if it is XML schema-based and as a CLOB instance if non XML schema-based. You can store either as binary XML instead, by setting the OPTIONS parameter for XML schema registration to REGISTER BINARYXML.

Example 29-1 Registering 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"</pre>



```
xmlns:xdb="http://xmlns.oracle.com/xdb"
                                    xmlns="inamespace">
                          <xsd:element name="ImgTechMetadata"</pre>
                                       xdb:defaultTable="IMGTECHMETADATATABLE">
                            <xsd:complexType>
                              <xsd:sequence>
                                                                 type="xsd:float"/>
                                 <xsd:element name="Height"</pre>
                                 <xsd:element name="Width"</pre>
                                                                 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;
```

Example 29-2 Registering 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"</pre>
                                        xdb:defaultTable="IMGCATMETADATATABLE">
                            <xsd:complexType>
                               <xsd:sequence>
                                 <xsd:element name="Categories"</pre>
                                              type="CategoriesType"/>
                              </xsd:sequence>
                            </xsd:complexType>
                          </xsd:element>
                          <xsd:complexType name="CategoriesType">
                            <xsd:sequence>
                               <xsd:element name="Category" type="xsd:string"</pre>
                                           maxOccurs="unbounded"/>
                            </xsd:sequence>
                          </xsd:complexType>
                        </xsd:schema>',
    enableHierarchy => DBMS XMLSCHEMA.ENABLE HIERARCHY RESMETADATA);
END;
/
```

Addition, Modification, and Deletion of Resource Metadata

You can add, update, and delete user-defined resource metadata using PL/SQL procedures in package DBMS_XDB_REPOS, SQL DML statements INSERT, UPDATE, and DELETE, or WebDAV protocol method PROPPATCH.

You can add, update, and delete user-defined resource metadata in any of the following ways:

- Use PL/SQL procedures in package DBMS_XDB_REPOS:
 - 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_REPOS 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.

Adding Metadata Using APPENDRESOURCEMETADATA
 You can use procedure DBMS_XDB_REPOS.appendResourceMetadata to add user-defined

metadata to resources.

Deleting Metadata Using DELETERESOURCEMETADATA

You can use procedure DBMS_XDB_REPOS.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_REPOS.purgeResourceMetadata.

 Adding Metadata Using SQL DML An alternative to using procedure DBMS_XDB_REPOS.appendResourceMetadata to add, update, or delete resource metadata is to update the RESOURCE_VIEW directly using DML statements INSERT and UPDATE.

 Adding Metadata Using WebDAV PROPPATCH An alternative to using procedure DBMS_XDB_REPOS.appendResourceMetadata to add resource metadata is to use WebDAV method PROPPATCH.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about the procedures in PL/SQL package DBMS_XDB_REPOS

Adding Metadata Using APPENDRESOURCEMETADATA

You can use procedure DBMS_XDB_REPOS.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 29-4 adds metadata of type ImgTechMetadata to the same resource as Example 29-3, placing the photo in several user-defined content categories.



```
Example 29-3 Add Metadata to a Resource – Technical Photo Information
```

```
DECLARE
 returnbool BOOLEAN;
BEGIN
  returnbool := DBMS XDB REPOS.createResource(
                  '/public/horse with pig.jpg',
                  bfilename('MYDIR', 'horse_with_pig.jpg'));
  DBMS XDB REPOS.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 Add Metadata to a Resource – Photo Content Categories
```

```
BEGIN
 DBMS XDB REPOS.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_REPOS.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_REPOS.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

Adding Metadata Using SQL DML

An alternative to using procedure DBMS_XDB_REPOS.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 =
   XMLQuery('declare namespace r = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
              declare namespace c = "cnamespace"; (: :)
              copy $tmp := . modify insert node
               <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>
              into $tmp/r:Resource
              return $tmp'
```



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

An alternative to using procedure DBMS_XDB_REPOS.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

You can use metadata column RESID when querying resource metadata, to join the metadata with the associated data.

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:



 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

Digital cameras include image metadata as part of the image files they produce.

- 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.

Adding Non-Schema-Based Resource Metadata

You store user-defined resource metadata that is *not* XML 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 in the resource table, XDB\$RESOURCE.

The following skeleton shows the structure and position of non-schema-based resource metadata:

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_REPOS.createResource(
```

```
'/public/NurseryRhyme.txt',
           bfilename('MYDIR', 'tdadxdb-xdb repos meta-011.txt'),
           nls charset id('AL32UTF8'));
  UPDATE RESOURCE VIEW SET RES =
   XMLQuery('declare namespace r = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
             declare namespace n = "nurserynamespace"; (: :)
             copy $tmp := . modify insert node
              <n:NurseryMetadata>
                <Author>Mother Goose</Author>
              </n:NurseryMetadata>
             into $tmp/r:Resource
             return $tmp'
            PASSING RES
            RETURNING CONTENT)
   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

You can use PL/SQL procedures DBMS_XMLSCHEMA.registerSchema, DBMS_XDBZ.enable_hierarchy, DBMS_XDBZ.disable_hierarchy, DBMS_XDBZ.is_hierarchy_enabled, DBMS_XDB_REPOS.appendResourceMetadata, DBMS_XDB_REPOS.deleteResourceMetadata, DBMS_XDB_REPOS.purgeResourceMetadata, and DBMS_XDB_REPOS.updateResourceMetadata to 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_REPOS.appendResourceMetadata Add metadata to a resource.
- DBMS_XDB_REPOS.deleteResourceMetadata Delete specified metadata from a resource.
- DBMS_XDB_REPOS.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 REPOS.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

The use of repository events to trigger application actions is deprecated in Oracle Database 21c (21.3). There is no replacement.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

Repository events are events that can be used to trigger application actions. Repository events include repository changes, such as 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 a resource. The deprecation of the use of repository events includes deprecation of the DBMS_XEVENT package, and the following subprogram groups:

You can use Oracle XML DB Repository to store and access data of any kind, in the form of repository resources. You can access repository data from any application. Sometimes your application needs to perform certain actions whenever a particular repository operation occurs. You can do this using repository events, but doing so is deprecated.

- Overview of Repository Events Your application can perform specific actions when events associated with repository operations occur. For example, you might want to perform a move-to-wastebasket or another backup action whenever a resource is deleted.
- Possible Repository Events Repository operations are associated with predefined events. Except for a rendering operation, events come in pre and post pairs.
- 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.
- Repository Event Handler Considerations
 Some considerations are listed for defining handlers for Oracle XML DB Repository events.
- Configuration of Repository Events In a resource configuration file, you use element event-listeners, child of element ResConfig, to configure Oracle XML DB Repository event handling.

Related Topics

•

Configuration of Oracle XML DB Repository
 Overall configuration of Oracle XML DB Repository applies to all repository resources. It
 does not include configuring parameters for handling events or managing XLink and
 XInclude processing. You use resource configuration files to configure resources.

Overview of Repository Events

Your application can perform specific actions when events associated with repository operations occur. For example, you might want to perform a move-to-wastebasket or another backup action whenever a resource is deleted.

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.

Repository Events: Use Cases

Examples of cases where you can use repository events include moving a resource to a wastebasket and categorizing a resource based on its MIME type.

Repository Events and Database Triggers

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.

• Repository Event Listeners and Event Handlers

Each repository operation is associated with events. Your application can configure event listeners for particular resources or the entire repository. A listener can be restricted by a node-existence precondition. A listener is a set of PL/SQL or Java handlers, each of which processes a single event.

Repository Event Configuration

Repository event configuration involves defining resource configuration files and defining the order in which they are processed. The files define event listeners. Such configuration applies to events on individual resources and events for the repository as a whole.

Repository Events: Use Cases

Examples of cases where you can use repository events include moving a resource to a wastebasket and categorizing a resource based on its MIME type.

- 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

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 events. Your application can configure event listeners for particular resources or the entire repository. A listener can be restricted by a node-existence precondition. A listener is a set of PL/SQL or Java handlers, each of which processes a single event.

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**.

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

Repository event configuration involves defining resource configuration files and defining the order in which they are processed. The files define event listeners. Such configuration applies to events on individual resources and events for the repository as a whole.

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. Events for the repository as a whole can also be configured by multiple resource configuration files. Similarly, the repository also 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.



Related Topics

Configuration of Repository Events

In a resource configuration file, you use element event-listeners, child of element ResConfig, to configure Oracle XML DB Repository event handling.

See Also:

Resource Configuration Files for general information about resource configuration and resource configuration lists

Possible Repository Events

Repository operations are associated with predefined events. Except for a rendering operation, events come in pre and post pairs.

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.

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	N/A	N/A
	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.		
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.
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.

Table 30-1 Predefined Repository Events



Repository Event Type	Description	Pre Handler Execution	Post Handler Execution
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_VERSION.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.

Table 30-1 (Cont.) Predefined Repository Events

For simplicity, the documentation 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	lf 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 for information on accessing resources using APIs and protocols



Repository Event Handler Considerations

Some considerations are listed for defining 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 Database 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
- Configuration of Repository Events for information about defining repository event handlers with invoker's rights

Configuration of Repository Events

In a resource configuration file, you use element event-listeners, child of element ResConfig, to configure Oracle XML DB Repository event handling.



You configure event treatment for Oracle XML DB Repository resources as you would configure any other treatment of repository resources — see "Configuring a Resource".

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.

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.

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.

Configuration Element listener

Element listener is a child of element event-listeners, and it configures an individual repository event listener.

Repository Events Configuration Examples

Examples of configuring repository events are presented. Resource configuration files define Java and PL/SQL event listeners, with and without preconditions, respectively. An example categorizes resources according to MIME type. It includes PL/SQL code to create the resource configuration file. Examples implement listeners in Java and PL/SQL.

See Also:

Configuration of 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 for information about insufficientprivilege 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

Element listener is a child of element event-listeners, and it configures an individual repository event 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 Elements defaultChildConfig and configuration.
- 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 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

Examples of configuring repository events are presented. Resource configuration files define Java and PL/SQL event listeners, with and without preconditions, respectively. An example categorizes resources according to MIME type. It includes PL/SQL code to create the resource configuration file. Examples implement listeners in Java and PL/SQL.

Example 30-1 shows the content of a resource configuration file that defines two event 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 the default database schema.

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 <code>oracle.ifs.quota</code> defined in database schema <code>IFS</code> (the default schema for this resource configuration file). These handlers are invoked only for events on resources of type <code>ifs-file</code> in namespace <code>http://foo.xsd</code>.

See Also:

Configuration Elements defaultChildConfig and configuration 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 30-3 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.



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
 Oracle Database PL/SQL Packages and Types Reference for information about PL/SQL package DBMS_XDB_REPOS

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 30-5 demonstrates the invocation of the event handlers that are implemented in Example 30-3 or Example 30-4.

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>
      <event.s>
       <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>
      <event.s>
       <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:iteml>1234</foo:iteml>

</foo:data>

</nopplicationData>

</resConfig>
```

Example 30-2 Resource Configuration File for PL/SQL Event Listeners with No Preconditions

```
DECLARE
 b BOOLEAN := FALSE;
BEGIN
 b := DBMS XDB REPOS.createFolder('/public/resconfig');
  b := DBMS XDB REPOS.createResource(
         '/public/resconfig/appcatg-rc1.xml',
         '<ResConfig xmlns="http://xmlns.oracle.com/xdb/XDBResConfig.xsd"</pre>
                     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 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);
               VARCHAR2(1000);
     ResPath
     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);
                  := DBMS XEVENT.getName(XDBPathObj);
     ResPath
     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 REPOS.deleteResource (LinkFolder || ResDisplayName);
  END;
  PROCEDURE handlePostLinkIn (eventObject DBMS XEVENT.XDBRepositoryEvent) AS
     XDBResourceObj DBMS XDBRESOURCE.XDBResource;
     ResDisplayName VARCHAR2(100);
               VARCHAR2(1000);
     ResPath
                  VARCHAR2(1000);
     ResOwner
     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);
```

```
:= DBMS XEVENT.getName(XDBPathObj);
    ResPath
    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 REPOS.link (ResPath, LinkFolder, ResDisplayName);
   END;
END;
```

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 el)
    {
     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 REPOS.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 REPOS.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 Invoking Event Handlers

}

```
DECLARE
 ret BOOLEAN;
BEGIN
 ret := DBMS XDB REPOS.createResource('/public/res-app/res1.xml',
                                     '<name>TestForEventType-1</name>');
END;
DECLARE
 b BOOLEAN := FALSE;
 dummy data CLOB := 'AAA';
BEGIN
 b := DBMS XDB REPOS.createResource('/public/res-app/res2.gif', dummy data);
END;
DECLARE
 b BOOLEAN := FALSE;
 dummy data CLOB := 'AAA';
BEGIN
 b := DBMS XDB REPOS.createFolder('/public/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 REPOS.deleteResource('/public/res-app/res2.gif');
EXEC DBMS XDB REPOS.deleteResource('/public/res-app/res1.xml');
EXEC DBMS XDB REPOS.deleteResource('/public/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

Guidelines for Oracle XML DB Applications in Java

Design guidelines are presented for writing Oracle XML DB applications in Java. This includes guidelines for writing and configuring Java servlets for Oracle XML DB.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

- Overview of Oracle XML DB Java Applications You can use Java code either in a client or an application server, using the OCI driver for JDBC, or in the Java Virtual Machine (JVM).
- HTTP(S): Access Java Servlets or Directly Access XMLType Resources
 If a downstream client needs to work with XML in its textual representation then using
 HTTP(S) to either access 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.
- Use JDBC XMLType Support to Access Many XMLType Object Elements If a 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 Servlets to Manipulate and Write Out Data Quickly as XML Oracle XML DB servlets 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 endusers.
- Oracle XML DB Java Servlet Support Restrictions The Oracle XML DB protocol server supports FTP, HTTP 1.1, WebDAV, and Java Servlets. It supports Java Servlet version 2.2, with a few exceptions.
- Configuration of Oracle XML DB Servlets Oracle XML DB servlets are configured using file xdbconfig.xml 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 they have the same semantics.
- HTTP Request Processing for Oracle XML DB Servlets
 Oracle XML DB processing of an HTTP request is described.
- Session Pool and Oracle XML DB Servlets
 Oracle Database uses one Java virtual machine (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.
- Native XML Stream Support Java node class DOM has Oracle-specific method write(), which provides native XML stream support.



Oracle XML DB Servlet APIs

The APIs supported by Oracle XML DB servlets are described. They are defined by the Java Servlet 2.2 specification.

Oracle XML DB Servlet Example Examples show the definition of a simple Oracle XML DB servlet that prints the content of a file resource, and how to register and map that servlet.

Overview of Oracle XML DB Java Applications

You can use Java code either in a client or an application server, using the OCI driver for JDBC, or in the Java Virtual Machine (JVM).

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.

All Oracle XML DB application program interfaces (APIs) for Java are available to applications running both in the server and outside the database.

These APIs include:

- JDBC support for XMLType
- XMLType class
- Java DOM implementation

HTTP(S): Access Java Servlets or Directly Access XMLType Resources

If a downstream client needs to work with XML in its textual representation then using HTTP(S) to either access 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.



Use JDBC XMLType Support to Access Many XMLType Object Elements

If a 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 Servlets to Manipulate and Write Out Data Quickly as XML

Oracle XML DB servlets 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.

Servlets are intended for writing HTTP stored procedures in Java that can be accessed using HTTP(S). If you need to develop an entire Internet application then deploy your application servlet in Oracle Fusion Middleware and have the servlet access data in the database using either JDBC or APIs such as java.net.*.

Oracle XML DB Java Servlet Support Restrictions

The Oracle XML DB protocol server supports FTP, HTTP 1.1, WebDAV, and Java Servlets. It supports Java Servlet version 2.2, with a few exceptions.

Support for Java Servlet version 2.2. has these restrictions:

- 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.

Configuration of Oracle XML DB Servlets

Oracle XML DB servlets are configured using file xdbconfig.xml 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 they have the same semantics.

Table 31-1 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 31-1 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	N/A



XML Element Name	Defined By	Supported?	Description	Comment
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	N/A
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 <i>must</i> be distributable.
errnum	Oracle	yes	Oracle error number	See Oracle Database Error Messages Reference
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	N/A
extension	Java	yes	A filename extension used to associate with MIME types, character sets, and so on.	N/A
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
icon	Java	Yes	URL of icon to associate with a servlet	Supported for servlets
init-param	Java	Yes	Initialization parameter for a servlet	N/A
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	N/A
large-icon	Java	Yes	Large sized icon for icon display	N/A
load-on-startup	Java	Yes	Specifies if a servlet is to be loaded on startup	N/A

Table 31-1 (Cont.) XML Elements Defined for Servlet Deployment Descriptors



extension and the MIME type of the contentmime-typeJavaYesMIME type name for resource content application/octet-streamOracleErrorOracleYesSpecifies an Oracle error to associate with an error pageN/Aparam-nameJavaYesName of a parameter for a Servlet or ServletContextSupported for servletsparam-valueJavaYesValue of a parameterN/Arealm-nameJavaNoHTTP(S) realm used for authentication have for accessing a servletNot supportedrole-linkJavaYesSpecifies a role a particular user must have for accessing a servletRefers to a database role name. Make sure to capitalize by default!role-nameJavaYesA servlet name for a roleJust another name to call the database role. Used by the Servlet APIs	XML Element Name	Defined By	Supported?	Description	Comment
mime-mappingJavaYesSpecifies a mapping between filename extension and the MIME type of the contentN/Amime-typeJavaYesMIME type name for resource content application/octet-streamFor example: text/xml or application/octet-streamOracleErrorOracleYesSpecifies an Oracle error to associate with an error pageN/Aparam-nameJavaYesName of a parameter for a Servlet or ServletOntextSupported for servletsparam-valueJavaYesValue of a parameterN/Arealm-nameJavaNoHTTP(S) realm used for authenticationNot supportedrole-linkJavaYesSpecifies a role a particular user must have for accessing a servletRefers to a database role name. Make sure to capitalize by defaultrole-nameJavaYesA servlet name for a roleJust another name to call the database role. Used by the Servlet APIssecurity-roleJavaYesA reference between a servlet to useNot supported. You must manually create roles using the SQL CREATE ROLEservletJavaYesSpecifies the classname for the Java servletN/Aservlet-classJavaYesSpecifies the programming language in servlet is written.Configuration information for a servlet servlet for servlet.servlet-languageOracleYesSpecifies a filename pattern with which to associate the servlet is written.All of the mappings defined yava are supported for coutomer-defined servlets.	location	Java	Yes	Specifies the URL for an error page	
with an extension and the MIME type of the contentmime-typeJavaYesMIME type name for resource contentFor example: text/xml or application/octet-streamOracleErrorOracleYesSpecifies an Oracle error to associate with an error pageN/Aparam-nameJavaYesName of a parameter for a Servlet or ServletContextSupported for servlets ServletContextparam-valueJavaYesValue of a parameter for a Servlet on ServletContextN/Arealm-nameJavaYesValue of a parameter for a coressing a servletN/Arole-linkJavaYesSpecifies a role a particular user must have for accessing a servletRefers to a database role name Make sure to capitalize by defaultirole-nameJavaYesA servlet name for a roleJust another name to capitalize by defaultirole-nameJavaYesA servlet name for a roleJust another name to capitalize by defaultisecurity-roleJavaYesA reference between a servlet to useN/Asecurity-role-refJavaYesConfiguration information for a servletN/Aservlet-classJavaYesSpecifies the classname for the JavaN/Aservlet-languageOracleYesSpecifies the programming language in servlet is written.Scherter, or PL/ SQL_Currenty, only Java is supported for cursters.servlet-mappingJavaYesSpecifies a filename pattern with whichAll of the mappings servlets.	login-config	Java	No	Specifies a method for authentication	Not supported
Oracle ErrorOracleYesSpecifies an Oracle error to associateN/Aparam-nameJavaYesName of a parameter for a Servlet or ServletContextSupported for servletsparam-nameJavaYesValue of a parameter for a Servlet or ServletContextN/Arealm-nameJavaYesValue of a parameterN/Arealm-nameJavaYesSpecifies a role a particular user must have for accessing a servletRefers to a database role name. Make sure to capitalize by default!role-nameJavaYesA servlet name for a roleJust another name to call the database role. Used by the Servlet APIssecurity-roleJavaYesA feference between a servlet to use servletNot supported. You must manually creater roles rolesecurity-role-refJavaYesA reference between a servlet and a roleN/Aservlet-classJavaYesSpecifies the classname for the Java servletN/Aservlet-languageOracleYesSpecifies the programming language in servletN/Aservlet-mappingJavaYesSpecifies a filename pattern with with die servletN/A	mime-mapping	Java	Yes	extension and the MIME type of the	N/A
with an error pageparam-nameJavaYesName of a parameter for a Servlet or ServletContextSupported for servlets ServletContextparam-valueJavaYesValue of a parameterN/Arealm-nameJavaNoHTTP(S) realm used for authenticationNot supportedrole-linkJavaYesSpecifies a role a particular user must have for accessing a servletRefers to a database role name. Make sure to capitalize by default!role-nameJavaYesA servlet name for a roleJust another name to capitalize by default!role-nameJavaYesA servlet name for a roleJust another name to capitalize by default!security-roleJavaYesA servlet name for a servlet to useNot supported. You must APIssecurity-role-refJavaYesA reference between a servlet on a servletN/Aservlet-classJavaYesSpecifies the classname for the Java servletN/Aservlet-languageOracleYesSpecifies the programming language in which the servlet is written.Either Java, C, or PL/ SQL. Currently, only Java is supported for customer-defined servlet servletservlet-mappingJavaYesSpecifies a filename pattern with which defined by Java are supported	mime-type	Java	Yes	MIME type name for resource content	
ServletContextN/Aparam-valueJavaYesValue of a parameterN/Arealm-nameJavaNoHTTP(S) realm used for authenticationNot supportedrole-linkJavaYesSpecifies a role a particular user must have for accessing a servletRefers to a database role name. Make sure to capitalize by default!role-nameJavaYesA servlet name for a roleJust another name to call the database role. Used by the Servlet APIssecurity-roleJavaNoDefines a role for a servlet to useNot supported. You must manually create roles using the SQL CREATE ROLEsecurity-role-refJavaYesA reference between a servlet and a roleN/Aservlet-classJavaYesSpecifies the classname for the Java servletN/Aservlet-languageOracleYesSpecifies 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-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mappings defined by Java are supported	OracleError	Oracle	Yes		N/A
realm-nameJavaNoHTTP(S) realm used for authenticationNot supportedrole-linkJavaYesSpecifies a role a particular user must have for accessing a servletRefers to a database role name. Make sure to capitalize by default!role-nameJavaYesA servlet name for a roleJust another name to call the database role. Used by the Servlet APIssecurity-roleJavaNoDefines a role for a servlet to use roleNot supported. You must manually create roles using the SQL CREATE ROLEsecurity-role-refJavaYesA reference between a servlet and a roleN/AservletJavaYesSpecifies the classname for the Java servlet-classN/Aservlet-languageOracleYesSpecifies the programming language in which the servlet is written.Either Java, C, or PL/ SQL. Currently, only Java is supported for customer-defined servlet.servlet-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mappings defined by Java are supported	param-name	Java	Yes	-	Supported for servlets
role-linkJavaYesSpecifies a role a particular user must have for accessing a servletRefers to a database role name. Make sure to capitalize by default!role-nameJavaYesA servlet name for a roleJust another name to call the database role. Used by the Servlet APIssecurity-roleJavaNoDefines a role for a servlet to useNot supported. You must manually create roles using the SQL CREATE ROLEsecurity-role-refJavaYesA reference between a servlet and a roleN/AservletJavaYesConfiguration information for a servletN/Aservlet-classJavaYesSpecifies the programming language in servlet is written.Either Java, C, or PL/ SQL. Currently, only Java is supported for customer-defined servlets.servlet-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mappings defined by Java are supported	param-value	Java	Yes	Value of a parameter	N/A
have for accessing a servletrole name. Make sure to capitalize by default!role-nameJavaYesA servlet name for a roleJust another name to call the database role. Used by the Servlet APIssecurity-roleJavaNoDefines a role for a servlet to useNot supported. You must manually create roles using the SQL CREATE ROLEsecurity-role-refJavaYesA reference between a servlet and a roleN/AservletJavaYesConfiguration information for a servletN/Aservlet-classJavaYesSpecifies the classname for the Java servletN/Aservlet-languageOracleYesSpecifies 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-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mappings defined by Java are supported	realm-name	Java	No	HTTP(S) realm used for authentication	Not supported
security-roleJavaNoDefines a role for a servlet to useCall the database role. Used by the Servlet APIssecurity-role-refJavaYesA reference between a servlet and a roleN/AservletJavaYesConfiguration information for a servletN/Aservlet-classJavaYesSpecifies the classname for the Java servletN/Aservlet-languageOracleYesSpecifies the programming language in which the servlet is written.Either Java, C, or PL/ SQL. Currently, only Java is supported for customer-defined servletsservlet-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mappings defined by Java are supported	role-link	Java	Yes		role name. Make sure to
manually create roles using the SQL CREATE ROLEsecurity-role-refJavaYesA reference between a servlet and a roleN/AservletJavaYesConfiguration information for a servletN/Aservlet-classJavaYesSpecifies the classname for the Java servletN/Aservlet-languageOracleYesSpecifies 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-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mapping defined by Java are supported	role-name	Java	Yes	A servlet name for a role	call the database role. Used by the Servlet
roleservletJavaYesConfiguration information for a servletN/Aservlet-classJavaYesSpecifies the classname for the Java servletN/Aservlet-languageOracleYesSpecifies the programming language in which the servlet is written.Either Java, C, or PL/ SQL. Currently, only Java is supported for 	security-role	Java	No	Defines a role for a servlet to use	using the SQL CREATE
servlet-classJavaYesSpecifies the classname for the Java servletN/Aservlet-languageOracleYesSpecifies 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-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mappings defined by Java are supported	security-role-ref	Java	Yes		N/A
servlet-languageOracleYesSpecifies 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-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mappings defined by Java are supported	servlet	Java	Yes	Configuration information for a servlet	N/A
which the servlet is written.SQL. Currently, only Java is supported for customer-defined servlets.servlet-mappingJavaYesSpecifies a filename pattern with which to associate the servletAll of the mappings defined by Java are supported	servlet-class	Java	Yes	•	N/A
to associate the servlet defined by Java are supported	servlet-language	Oracle	Yes		SQL. Currently, only Java is supported for customer-defined
servlet-name Java Yes String name for a servlet Used by servlet APIs	servlet-mapping	Java	Yes		defined by Java are
	servlet-name	Java	Yes	String name for a servlet	Used by servlet APIs

 Table 31-1
 (Cont.) XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
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. The servlet schema is capitalized unless the value is enclosed in double quotation marks.
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	N/A
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	N/A
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

Table 31-1 (Cont.) XML Elements Defined for Servlet Deployment Descriptors

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, enventry-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, httpmethod, 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.

Related Topics

 Configuration of 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.

HTTP Request Processing for Oracle XML DB Servlets

Oracle XML DB processing of an HTTP request is described.

HTTP request handling proceeds as follows:

- **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.

Related Topics

 Repository Access Using Protocols You can access Oracle XML DB Repository data using protocols FTP and HTTP(S)/ WebDAV.

Session Pool and Oracle XML DB Servlets

Oracle Database uses one Java virtual machine (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

Java node class DOM has Oracle-specific method write(), which provides native XML stream support.

Java method write() takes the following arguments and returns 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.

Method write() has a shortcut implementation if the stream is the ServletOutputStream provided inside the database. The contents of the Node are written as XML data 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 described. They are defined by the Java Servlet 2.2 specification.

The Javadoc for this is available at http://download.oracle.com/javaee/1.2.1/api/index.html.

Table 31-2 lists Java Servlet 2.2 methods that are not implemented. They result in run-time exceptions.

Interface	Methods Not Implemented
HttpServletRequest	<pre>getSession(), isRequestedSessionIdValid()</pre>
HttpSession	all

Table 31-2 Java Servlet 2.2 Methods that Are Not Implemented



Table 31-2 (Cont.) Java Servlet 2.2 Methods that Are Not Implemented

Interface	Methods Not Implemented
HttpSessionBindingListener	all

Oracle XML DB Servlet Example

Examples show the definition of a simple Oracle XML DB servlet that prints the content of a file resource, and how to register and map that servlet.

The servlet shown in Example 31-1 prints the content of file resource /public/test/ fool.text.

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 31-2.

Example 31-1 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/fool.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(); }}
```

Example 31-2 Registering and Mapping an Oracle XML DB Servlet

EXEC DBMS_XDB_CONFIG.addServlet('TestServletFoo', 'Java', 'TestServletFoo', NULL, NULL, NULL, 'Lest', NULL, NULL, 'XDB');

EXEC DBMS_XDB_CONFIG.addServletMapping('/public/test/fool.txt', 'TestServletFoo');

COMMIT;

Data Access Using URIs

You can generate and store URIs in the database and use them to retrieve the database data they target. There are three kinds of URIs you can use this way: DBUris, XDBUris, and HTTPUris.

Note:

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

Overview of Oracle XML DB URI Features

You can use a URI as an indirection mechanism to access database data, and you can use a URI that targets database data to produce XML documents.

URIs and URLs

In developing Web-based XML applications, you often refer to data located on a network using **Uniform Resource Identifiers**, or **URIs**. A **URL**, or **Uniform Resource Locator**, is a URI that accesses an object using an Internet protocol.

URIType and its Subtypes

You can represent paths of various kinds as database objects. These provide unified access to data stored inside and outside the server, and they can be used to map URIs in XML documents to database columns, letting documents reference data stored in relational columns and expose it externally.

• Accessing Data Using URIType Instances

To use instances of URIType subtypes for indirection, you store such instances in the database and then query to retrieve the targeted data with a PL/SQL method such as getCLOB().

• XDBUris: Pointers to Repository Resources

XDBURIType is a subtype of URIType that exposes resources in Oracle XML DB Repository using URIs. Instances of object type XDBURIType are called **XDBURIs**.

DBUris: Pointers to Database Data

A DBUri is a URI that targets *database data*. As for all instances of URIType subtypes, a DBUri provides indirect access to data. DBURIType also lets you address database data using XPath and construct XML documents containing database data that is targeted by a DBUri that reflects the database structure.

Create New Subtypes of URIType Using Package URIFACTORY

You can define your own subtypes of URIType that correspond to particular protocols. You can use PL/SQL package URIFACTORY to obtain the URI of a URIType instance, escape characters in a URI string or remove such escaping, and register or unregister a type name for handling a given URL.

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 using an XPath that is composed from database columns and their values.



DBUriServlet

You can retrieve repository resources using the Oracle XML DB HTTP server. 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.

Overview of Oracle XML DB URI Features

You can use a URI as an indirection mechanism to access database data, and you can use a URI that targets database data to produce XML documents.

- 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:

- Web Services Activity Statement for an explanation of HTTP(S) URL notation
- XML Path Language (XPath)
- XML Pointer Language (XPointer)
- XML and MIME Media-Types

URIType and its Subtypes

You can represent paths of various kinds as database objects. These provide unified access to data stored inside and outside the server, and they can be used to map URIs in XML documents to database columns, letting documents reference data stored in relational columns and expose it externally.

The available path object types are HTTPURIType, DBURIType, and XDBURIType, all of which are derived from abstract object type URIType.

 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:

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"]

 XDBURIType – An object of this type is called an XDBUri, and represents a URI that targets a resource in Oracle XML DB Repository. With PL/SQL constructor XDBURIType you can create objects that represent links to *repository resources*. You can then retrieve all or part of any resource by calling methods on those objects. 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.

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.

Overview of DBUris and XDBUris

Important uses of DBUris and XDBUris include referencing XSLT stylesheets from Web pages, referencing data in database tables or in repository folders without using SQL, and improving performance by bypassing the Web server.

URIType PL/SQL 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.

Related Topics

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 instance or a CLOB instance.

DBUris: Pointers to Database Data

A DBUri is a URI that targets *database data*. As for all instances of URIType subtypes, a DBUri provides indirect access to data. DBURIType also lets you address database data using XPath and construct XML documents containing database data that is targeted by a DBUri that reflects the database structure.

• XDBUris: Pointers to Repository Resources

XDBURIType is a subtype of URIType that exposes resources in Oracle XML DB Repository using URIs. Instances of object type XDBURIType are called **XDBUris**.

Indexes for 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. There are various ways that you can index XMLType data, whether it is XML schema-based or non-schema-based, and regardless of the XMLType storage model you use.

See Also:

Create New Subtypes of URIType Using Package URIFACTORY for information about defining new URIType subtypes

Overview of DBUris and XDBUris

Important uses of DBUris and XDBUris include referencing XSLT stylesheets from Web pages, referencing data in database tables or in repository folders without using SQL, and improving performance by bypassing the Web server.

 You can reference XSLT stylesheets from within database-generated Web pages. PL/SQL package DBMS_METADATA uses DBUris to reference XSLT stylesheets. An XDBUri can be used to reference XSLT stylesheets 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 PL/SQL 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 32-1 lists the URIType PL/SQL methods. In addition, each of the subtypes has a constructor with the same name as the subtype.

Table 32-1	URIType PL/SQL Methods
-------------------	------------------------

URIType Method	Description
getURL()	Returns the URL of the URIType instance.
<u>,</u> ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,	Use this method instead of referencing a URL directly. URIType subtypes override this method to provide the correct URL. For example, <code>HTTPURIType</code> stores a URL without prefix <code>http://.</code> Method <code>getURL()</code> 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.
	<i>HTTPUri:</i> To return the content type, the URL is followed and the MIME header examined.
	<i>DBUri:</i> The returned content type is either text/plain (for a scalar value) or text/xml (otherwise).
	<i>XDBUri:</i> The value of the ContentType metadata property of the repository resource is returned.
getCLOB()	Returns the target of the URI as a CLOB instance. The database character set is used for encoding the data.
	<i>DBUri: XML</i> 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 <i>translated as hexadecimal character data</i> .



URIType Method	Description
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.
	<i>DBUri:</i> When applied to a DBUri that targets a BLOB column, getBLOB() returns the binary data <i>translated as hexadecimal character data.</i> When applied to a DBUri that targets <i>non</i> -binary data, the data is returned in the database character set.
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 instance or a CLOB instance.

 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.

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.

DBURIType PL/SQL Method GETBLOB()

When applied to a DBUri that targets a BLOB column, PL/SQL method getBLOB() returns the binary data *translated as hexadecimal character data*. When applied to a DBUri that targets *non*-binary data, method getBLOB() returns the data (as a BLOB value) in the database character set.

See Also:

Oracle Database PL/SQL Packages and Types Reference

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 instance or a CLOB instance.

For example, you might treat a Web page with a MIME type of x/jpeg as a BLOB instance, and one with a MIME type of text/plain or text/html as a CLOB instance.



Example 32-1 tests the HTTP content type to determine whether to retrieve data as a CLOB or BLOB instance. The content-type data is the HTTP header, for HTTPURIType, or the metadata of the database column, for DBURIType.

Example 32-1 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 nonbinary 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...</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>...data translated to hex...</ROW></DBTAB></HR>

DBURIType PL/SQL Method GETBLOB()

When applied to a DBUri that targets a BLOB column, PL/SQL method getBLOB() returns the binary data *translated as hexadecimal character data*. When applied to a DBUri that targets *non*-binary data, method 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 store such instances in the database and then query to retrieve the targeted data with a PL/SQL method such as getCLOB().

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.

Example 32-2 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.)

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.

PL/SQL factory method 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 32-3 is similar to Example 32-2, 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 32-3, 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 32-2 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.

Example 32-3 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.

Related Topics

Create New Subtypes of URIType Using Package URIFACTORY

You can define your own subtypes of URIType that correspond to particular protocols. You can use PL/SQL package URIFACTORY to obtain the URI of a URIType instance, escape characters in a URI string or remove such escaping, and register or unregister a type name for handling a given URL.

• XSL Transformation and Oracle XML DB

You can apply XSL transformations to XML Schema-based documents using the built-in Oracle XML DB XSLT processor. In-database XML-specific optimizations can significantly reduce the memory required, eliminate the overhead associated with parsing, and reduce network traffic.

XDBUris: Pointers to Repository Resources

XDBURIType is a subtype of URIType that exposes resources in Oracle XML DB Repository using URIs. Instances of object 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). An optional fragment portion of the URI, after the number-sign (#), uses XPath syntax to target parts of an XML document.

• Using XDBUri: Examples

XDBUri examples here use URIs in a table to access a repository resource and, together with PL/SQL method getXML, to query and retrieve XML documents.

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). An optional fragment portion of the URI, after the number-sign (#), uses XPath syntax to target parts of an XML document.

The optional fragment portion of the URI is appropriate only if the targeted resource is an *XML* document, in which case the fragment portion targets one or more its parts. 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 object type is <code>XDBURIType</code>, not any particular resource file extension or the presence of # followed by an XPath expression. Even if the resource were named <code>foo.bar</code> instead of <code>doc1.xml</code>, the returned <code>URIType</code> instance would still be an XDBUri.

Using XDBUri: Examples

XDBUri examples here use URIs in a table to access a repository resource and, together with PL/SQL method getXML, to query and retrieve XML documents.

Example 32-4 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.

Because PL/SQL method getXML() returns an XMLType instance, you can use it with SQL/XML functions such as XMLQuery. The query in Example 32-5 illustrates this. The query retrieves all purchase orders numbered 999.

Example 32-4 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_REPOS.createFolder('/public/orders/');
res:=DBMS_XDB_REPOS.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, pol.xml is an XML file that is stored in /public/orders/
-- of the XML repository.
INSERT INTO uri tab VALUES
  (URIFACTORY.getURI('/public/orders/pol.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.
```

Example 32-5 Using PL/SQL Method GETXML with XMLCAST and XMLQUERY

DBUris: Pointers to Database Data

A DBUri is a URI that targets *database data*. As for all instances of URITYPE subtypes, a DBUri provides indirect access to data. DBURITYPE also lets you address database data using XPath and construct XML documents containing database data that is targeted by a DBUri that reflects the database structure.

 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...</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 .

View the Database as XML Data

Using DBURIType, you can have what amounts to *XML* views of the portions of the database to which you have access, presented *in the form of XML data*. When visualized this way, the database data is effectively wrapped in XML elements, resulting in one or more XML documents.

DBUri URI Syntax

An XPath expression is a path into XML data that addresses one or more nodes. A DBUri exploits virtual XML visualization of the database to use a *simple form* of XPath expression as a URI to address database data. This is so, whether or not the data is XML.

• DBUris are Scoped to a Database and Session

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).

Using DBUris —Examples
 A DBUri can identify a table, a row, a column in a row, or an attribute of an object column.

 Examples here show how to target different object types.

View the Database as XML Data

Using DBURIType, you can have what amounts to *XML* views of the portions of the database to which you have access, presented *in the form of XML data*. When visualized this way, the database data is effectively wrapped in XML elements, resulting in one or more XML documents.

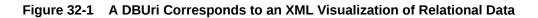
You can access only those database schemas to which you have been granted access privileges. This portion of the database is, in effect, your own view of the database. This applies to all kinds database data, not just data that is stored as XML.

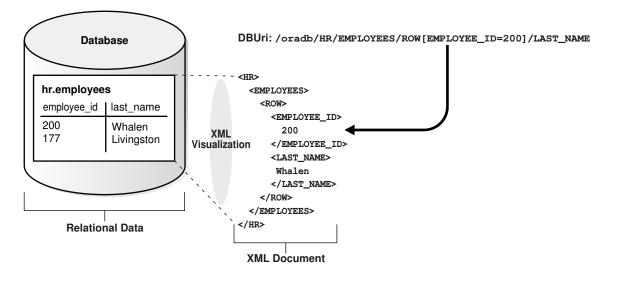
Such "XML views" are not database views, in the technical sense of the term. "View" here means only an abstract perspective that can be useful for understanding DBURIType. You can think of DBURIType as providing a way to visualize and access the database *as if it were* XML data.

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 32-1 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".





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.

Related Topics

Create New Subtypes of URIType Using Package URIFACTORY

You can define your own subtypes of URITYPE that correspond to particular protocols. You can use PL/SQL package URIFACTORY to obtain the URI of a URITYPE instance, escape characters in a URI string or remove such escaping, and register or unregister a type name for handling a given URL.

DBUriServlet

You can retrieve repository resources using the Oracle XML DB HTTP server. 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.

See Also:

Generation of XML Data from Relational Data 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 nodes. A DBUri exploits virtual XML visualization of the database to use a *simple form* of XPath expression as a URI to address database data. This is so, 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 *X*DBUri 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: XML Path Language (XPath) on XPath notation

DBUris are Scoped to a Database and Session

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).

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.

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.

Using DBUris — Examples

A DBUri can identify a table, a row, a column in a row, or an attribute of an object column. Examples here show how to target different object types.

• Targeting a Table Using a DBUri

An example 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.

- Targeting a Row in a Table Using a DBUri An example 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.
- Targeting a Column Using a DBUri You can target a given column, a given attribute of an object column, or an object column whose attributes have given values. Examples illustrate these possibilities.



Retrieving the Text Value of a Column Using a DBUri

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 stylesheets are stored in a CLOB column, you can retrieve the document text without having any enclosing column-name tags. An example illustrates this.

Targeting a Collection Using a DBUri You can target a database collection, such as an ordered collection table (OCT). You must, however, target the entire collection – you cannot target individual members.

Targeting a Table Using a DBUri

An example 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.

This is shown in Example 32-6. You target a complete database table using this syntax:

/database schema/table

Example 32-6 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>
. . .
```

Targeting a Row in a Table Using a DBUri

1 row selected.

An example 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.

This is shown in Example 32-7. You target one or more specific rows of a table using this syntax:

```
/database schema/table/ROW[predicate expression]
```

Example 32-7 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 Using a DBUri

You can target a given column, a given attribute of an object column, or an object column whose attributes have given values. Examples illustrate these possibilities.

You can target a specific column, using this syntax:

/database_schema/table/ROW[predicate_expression]/column

ORACLE

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 32-8 uses a DBUri that targets column last_name for the same employee as in Example 32-7. The top-level XML element is named for the targeted column.

Example 32-9 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 32-8 Targeting a Specific Column Using a DBUri

Example 32-9 Targeting an Object Column with Specific Attribute Values Using a DBUri



```
<CITY>Poughkeepsie</CITY>
<STATE_PROVINCE>NY</STATE_PROVINCE>
<COUNTRY_ID>US</COUNTRY_ID>
</CUST ADDRESS>
```

1 row selected.

The DBUri here 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 Using a DBUri

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 stylesheets are stored in a CLOB column, you can retrieve the document text without having any enclosing column-name tags. An example illustrates this.

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 32-10 retrieves the text value of the employee last_name column for employee number 200, without the XML tags.

Example 32-10 Retrieve Only the Text Value of a Node Using a DBUri

1 row selected.

Targeting a Collection Using a DBUri

You can target a database collection, such as an ordered collection table (OCT). You must, however, target the entire collection – you cannot target individual members.

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 32-11 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 32-11 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.
```

Create New Subtypes of URIType Using Package URIFACTORY

You can define your own subtypes of URITYPE that correspond to particular protocols. You can use PL/SQL package URIFACTORY to obtain the URI of a URITYPE instance, escape characters in a URI string or remove such escaping, and register or unregister a type name for handling a given URL.

Additional PL/SQL methods are listed in Table 32-2.

PL/SQL Method	Description	
getURI()	Returns the URI of the URIType instance.	
escapeURI()	Escapes the URI string by replacing characters that are not permitted in URIs by their equivalent escape sequence.	

Table 32-2 URIFACTORY PL/SQL Methods



PL/SQL Method	Description	
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.	

Table 32-2 (Cont.) URIFACTORY PL/SQL Methods

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.

Registering New URIType Subtypes with Package URIFACTORY

To provide specialized processing of URIs, you define and register a new URIType subtype.

- 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 32-12 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 32-12 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

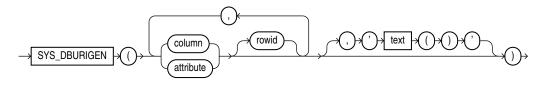
ECOMURIType('company1/company2=22/comp'); -- an ECOMURIType instance
```

SYS_DBURIGEN SQL Function

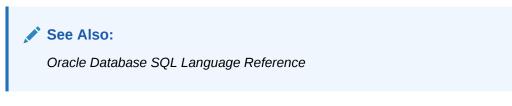
You can create a DBUri by providing an XPath expression to constructor DBURIType or to appropriate URIFACTORY PL/SQL methods. With Oracle SQL function sys_DburiGen, you can alternatively create a DBUri using an XPath that is composed from database columns and their values.

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 32-2.

Figure 32-2 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.



Example 32-13 uses Oracle SQL function sys_DburiGen to generate a DBUri that targets column email of table HR.employees where employee_id is 206:

Example 32-13 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 certain
 rules.



Using SQL Function SYS_DBURIGEN: Examples

Examples are presented that use SQL function sys_DburiGen to insert database references, return partial results from a large column, and return URLs to inserted objects.

Rules for Passing Columns or Object Attributes to SYS_DBURIGEN

A column or attribute passed to Oracle SQL function sys DburiGen must obey certain 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 32-14 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 32-14 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.

Using SQL Function SYS_DBURIGEN: Examples

Examples are presented that use SQL function sys_DburiGen to insert database references, return partial results from a large column, and return URLs to inserted objects.



- Inserting Database References Using SYS__DBURIGEN You can use SQL function sys_DburiGen to insert DBUris that reference specific database data. An example illustrates this.
- Returning Partial Results Using SYS__DBURIGEN
 When selecting data from a large column, you might sometimes want to retrieve only a portion of the result, and create a URL that provides access to the full column.
- Returning URLs to Inserted Objects Using SYS_DBURIGEN 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.

Inserting Database References Using SYS_DBURIGEN

You can use SQL function sys_DburiGen to insert DBUris that reference specific database data. An example illustrates this.

Example 32-15 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 Using SYS_DBURIGEN

When selecting data from a large column, you might sometimes want to retrieve only a portion of the result, and create a URL that provides access to the full column.

For example, consider the case of a travel story website. If travel stories are stored in a table and users search for a set of relevant stories, 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 32-16 creates the travel story table.

Example 32-17 creates a function that returns only the first 20 characters from the story.



Example 32-18 creates a view that selects only the first twenty characters from the travel story, and returns a DBUri to the story column.

Example 32-16 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 32-17 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 32-18 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 Using SYS_DBURIGEN

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 32-19, 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 32-19 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

You can retrieve repository resources using the Oracle XML DB HTTP server. 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 HTTPaware 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 stylesheets.

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:

Guidelines for Oracle XML DB Applications in Java, for information about Oracle XML DB servlets

Table 32-3 describes each of the optional URL parameters you can pass to DBUriServlet to customize its output.

Table 32-3	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 XSLT stylesheet at that location. This stylesheet 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 ¹

¹ This URL is split across two lines for the purpose of documentation.

 Overriding the MIME Type Using a URL
 You can override MIME content type by using a URL that passes a different MIME type to the servlet as the contenttype parameter.

- Customizing DBUriServlet To customize DBUriServlet you modify the Oracle XML DB configuration file, xdbconfig.xml.
- Using Roles for 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).



•

Configuring Package URIFACTORY to Handle DBUris

To improve efficiency, you can teach URIFACTORY that a URI of a given form represents database access and so should be realized as a DBUri, not an HTTPUri. You do this by registering a handler for the URI as a prefix, specifying DBURIType as the type of instance to generate.

Table or View Access from a Web Browser Using DBUri Servlet

Oracle XML DB includes the DBUri servlet, which lets you access the content of any table or view directly from a web browser. It uses DBURIType to generate a simple XML document from the table contents. The servlet is C language-based and installed in the Oracle XML DB HTTP server.

Overriding the MIME Type Using a URL

You can override MIME content type by using a URL that passes a different MIME type to the servlet as the contenttype parameter.

To retrieve column employee_id of table employee, 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

To customize DBUriServlet you modify the Oracle XML DB configuration file, xdbconfig.xml.

You can edit the Oracle XML DB configuration file, xdbconfig.xml, using database schema (user account) XDB with 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. PL/SQL package DBMS_XDB_CONFIG provides a particularly convenient way to access the file, and it provides subprograms that perform specific configuration modifications. For example, you can use DBMS_XDB_CONFIG.deleteservletmapping to remove a servlet mapping.

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 32-20, the configuration file is modified to install DBUriServlet under /dburi/*. (The long XPath expression has been split here for documentation purposes. It actually needs to be on a single line.)

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 XQuery Update to update the servlet-mapping element to NULL.



Example 32-20 Changing the Installation Location of DBUriServlet

```
DECLARE
  doc XMLType;
  doc2 XMLType;
BEGIN
  doc := DBMS XDB CONFIG.cfg get();
  SELECT XMLQuery('declare default element namespace
                   "http://xmlns.oracle.com/xdb/xdbconfig.xsd";
                   copy $i := $doc modify
                   for $j in
\texttt{si/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/servlet-mappings1}
/servlet-mapping[servlet-name="DBUriServlet"]/servlet-pattern
                   return replace value of node $j with $i/dburi/*
                  return $i'
                  PASSING DBMS XDB CONFIG.cfg get() AS "doc"
                  RETURNING CONTENT) INTO doc2 FROM DUAL;
  DBMS XDB CONFIG.cfg update(doc2);
  COMMIT;
END;
```

Related Topics

•

- Guidelines for Oracle XML DB Applications in Java
 Design guidelines are presented for writing Oracle XML DB applications in Java. This
 includes guidelines for writing and configuring Java servlets for Oracle XML DB.
- Administration of Oracle XML DB Administration of Oracle XML DB includes installing, upgrading, and configuring it.

```
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. You can use 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).
```

Using Roles for 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

¹ This XQuery expression is split across two lines only for the purpose of documentation.

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 32-21 changes the default role authenticatedUser to role servlet-users (which you must have created).

Example 32-21 Restricting Servlet Access to a Database Role

(The URLs in this XQuery expression are split across multiple lines only for the purpose of documentation.)

```
DECLARE
  doc XMLType;
  doc2 XMLType;
 doc3 XMLType;
BEGIN
 doc := DBMS XDB CONFIG.cfg get();
  SELECT
   XMLQuery('copy $i := $p1 modify
               (for $j in $i/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/servlet-list/
servlet[servlet-name="DBUriServlet"]/security-role-ref/role-name
               return replace value of node $j with $p2)
              return $i'
            PASSING DOC AS "p1", 'servlet-users' AS "p2" RETURNING CONTENT)
   INTO doc2 FROM DUAL;
  SELECT XMLQuery('copy $i := $p1 modify
                    (for $j in $i/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/
servlet-list/servlet[servlet-name="DBUriServlet"]/security-role-ref/role-link
                    return replace value of node $j with $p2)
                   return Si'
                  PASSING DOC2 AS "p1", 'servlet-users' AS "p2" RETURNING CONTENT)
   INTO doc3 FROM DUAL;
  DBMS XDB CONFIG.cfg update(doc3);
 COMMIT;
END:
```

Configuring Package URIFACTORY to Handle DBUris

To improve efficiency, you can teach URIFACTORY that a URI of a given form represents database access and so should be realized as a DBUri, not an HTTPUri. You do this by registering a handler for the URI as a prefix, specifying DBURIType as the type of instance to generate.

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 the methods of PL/SQL 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 teach URIFACTORY that URIS of the given form represent database accesses and so should be realized as DBUris, not HTTPUris, you register a handler for the URIs as a prefix, specifying DBURIType as the type of instance to generate.



Example 32-22 effectively tells URIFACTORY that any URI string starting with http://server/ servlets/oradb corresponds to a database access.

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 Referencefor information about URIFACTORY functions

```
Example 32-22 Registering a Handler for a DBUri Prefix
```

Table or View Access from a Web Browser Using DBUri Servlet

Oracle XML DB includes the DBUri servlet, which lets you access the content of any table or view directly from a web browser. It uses DBURIType to generate a simple XML document from the table contents. 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.

XSLT stylesheet processing is initiated by specifying a transform parameter as part of the URL passed to DBUri servlet. The stylesheet is specified using a URI that references the location of the stylesheet 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 stylesheet 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.

Related Topics

DBUriServlet

You can retrieve repository resources using the Oracle XML DB HTTP server. 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.

33 Native Oracle XML DB Web Services

Your applications can access Oracle Database using native Oracle XML DB Web services.

Note:

•

The Oracle XML DB Repository is deprecated with Oracle Database 23ai.

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.

- Configuring and Enabling Web Services for Oracle XML DB
 To make 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.
- Query 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 database host and HTTP(S) port. It has an associated WSDL that specifies the formats of the incoming and outgoing documents using XML Schema. This WSDL is located at URL http://host:port/orawsv?wsdl.
- Access to 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 standalone (not in a package), http://host:port/orawsv/
 dbschema/fn_or_proc.

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.

Related Topics

Configuration of 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.

See Also:

- Web Services Activity for more information about Web services
- Simple Object Access Protocol (SOAP) 1.1
- Web Services Description Language (WSDL) 1.1 for information about the Web Services Description Language (WSDL)
- Fault Scenarios for information about SOAP fault handling

Configuring and Enabling Web Services for Oracle XML DB

To make 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".
- Enable Web services for specific users, by granting them appropriate roles Enabling Web Services for a Specific User.

For security reasons, Oracle XML DB is not preconfigured with native Web services enabled.

Configuring Web Services for Oracle XML DB

To make Web services available for Oracle XML DB, configure the servlet by logging on as user SYS and adding the servlet configuration to your Oracle XML DB configuration file, xdbconfig.xml. Then use procedures in PL/SQL package DBMS_XDB_CONFIG to add the servlet that is named by the servlet configuration.

Enabling Web Services for a Specific User
 To enable Web services for a specific user, log on as user SYS and grant role
 XDB_WEBSERVICES to the user. This role enables Web services over HTTPS. This role is required to be able to use Web services.

Related Topics

 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.



Configuring Web Services for Oracle XML DB

To make Web services available for Oracle XML DB, configure the servlet by logging on as user SYS and adding the servlet configuration to your Oracle XML DB configuration file, xdbconfig.xml. Then use procedures in PL/SQL package DBMS_XDB_CONFIG to add the servlet that is named by the servlet configuration.

The servlet configuration to add is shown as the query output of Example 33-2.

Example 33-1 shows how to use procedures in PL/SQL package DBMS_XDB_CONFIG 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 CONFIG.deleteServletMapping(SERVLET_NAME);
 DBMS XDB CONFIG.deleteServlet(SERVLET NAME);
 DBMS XDB CONFIG.addServlet(
   NAME => SERVLET NAME,
   LANGUAGE => 'C',
   DISPNAME => 'Oracle Query Web Service',
   DESCRIPT => 'Servlet for issuing queries as a Web Service',
   SCHEMA => 'XDB');
 DBMS XDB CONFIG.addServletSecRole(SERVNAME => SERVLET NAME,
                                   ROLENAME => 'XDB WEBSERVICES',
                                   ROLELINK => 'XDB WEBSERVICES');
 DBMS_XDB_CONFIG.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.
```



Related Topics

Configuration of 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.

Enabling Web Services for a Specific User

To enable Web services for a specific user, log on as user SYS and grant 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.

Query 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 database host and HTTP(S) port. It has an associated WSDL that specifies the formats of the incoming and outgoing documents using XML Schema. This WSDL 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.

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.

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 33-3 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="SOL"/>
                      <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>
```

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>
```

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>
```

Access to 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 standalone (not in a package), http://host:port/orawsv/dbschema/ fn or proc.

Here, *host* and *port* are the database host and HTTP(S) port, *fn_or_proc* is the stored function or procedure name, *package* is its package, 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 functionnameInput, 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 mode
 - OUT mode
 - INOUT IN OUT mode
- The XML element introducing the output from a PL/SQL function is named sreturn-typefunction-nameOutput, 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. These are the only data types that are supported.

Table 33-1 Web Service Mapping Between XML and Oracle Database Data Types

Oracle Database 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
INT, INTEGER, SMALLINT, PLS_INTEGER, BINARY_INTEGER	xsd:integer
RAW, BLOB, REF	xsd:hexBinary
PL/SQL BOOLEAN	xsd:boolean
Object types	complexType
XMLType	empty 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.

Using a PL/SQL Function with a Web Service: Example
 Examples present a PL/SQL function and its access using a Web service. The function takes as input a department ID and name. It returns the salary total for the department. It also returns, as in-out and output parameters, respectively, the department name and the number of employees in the department.

Using a PL/SQL Function with a Web Service: Example

Examples present a PL/SQL function and its access using a Web service. The function takes as input a department ID and name. It returns the salary total for 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 inout 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"</pre>
```



```
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">
             <re><xsd:complexType/></re>
           </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"/>
           <rpre><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 VII

Oracle Tools that Support Oracle XML DB

You can use various Oracle tools with Oracle XML DB. These include tools for managing Oracle XML DB, loading XML data, and exchanging XML data.

- Administration of Oracle XML DB Administration of Oracle XML DB includes installing, upgrading, and configuring it.
- How to Load XML Data The main way to load XML data into Oracle XML DB is to use SQL*Loader.
- Export and Import of Oracle XML DB Data You can use Oracle Data Pump to export and import XMLType tables for use with Oracle XML DB.
- XML Data Exchange Using Oracle Database Advanced Queuing

You can exchange XML data using Oracle Database Advanced Queuing (AQ), which provides database-integrated message-queuing. AQ enables and manages asynchronous communication between applications, using messages. It supports point-to-point and publish/subscribe communication models.



34 Administration of Oracle XML DB

Administration of Oracle XML DB includes installing, upgrading, and configuring it.

- Upgrade or Downgrade of an Existing Oracle XML DB Installation Various considerations apply to all upgrades of Oracle Database from a release prior to Oracle Database 12c Release 1 (12.1.0.1).
- Administration of Oracle XML DB Using Oracle Enterprise Manager You can use Oracle Enterprise Manager to easily perform Oracle XML DB administration tasks. This includes configuring Oracle XML DB; creating, searching, editing, and deleting XML schemas, XMLType tables and views, and Oracle XML DB Repository resources and their access control lists (ACLs); registering XML schemas; and creating function-based indexes.
- Configuration of 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.

- Oracle XML DB and Database Consolidation
 Each pluggable database has its own Oracle XML DB Repository, and its own Oracle XML DB configuration file, xdbconfig.xml.
- Package DBMS_XDB_ADMIN PL/SQL package DBMS_XDB_ADMIN has subprograms for managing and configuring Oracle XML DB and Oracle XML DB Repository.

Related Topics

• Configuration of Repository Resources for XLink and XInclude The resource configuration file that you use as a resource to configure XLink (*deprecated*) and XInclude processing for other resources is described.

Upgrade or Downgrade of an Existing Oracle XML DB Installation

Various considerations apply to all upgrades of Oracle Database from a release prior to Oracle Database 12c Release 1 (12.1.0.1).

- Run script catproc.sql, as always.
- Replication of hierarchy-enabled tables is not supported for any replication method, including rolling upgrade.
- If supplemental logging is turned on then these operations are not supported:
 - Use of the APPEND hint for INSERT
 - SQL*Loader direct-path insertion of XMLType data
- If you use rolling upgrade and any of the following operations are invoked on the primary database, then an *unsupported operation* error is raised on the standby database:



- DBMS_XDB_ADMIN all operations
- DBMS_XMLSCHEMA.copyEvolve
- DBMS_XMLSCHEMA.compileSchema
- DBMS_XMLINDEX.dropparameter
- DBMS_XMLINDEX.modifyparameter
- DBMS_XMLINDEX.registerparameter
- ACL security: In releases prior to Oracle Database 11g 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 11g and later this deny-trumps-grant behavior is still available, but it is not the default behavior.
- Prior to Oracle Database 12c Release 1 (12.1.0.1), basic access authentication was the only available HTTP authentication mechanism. Starting with 12c Release 1, digest access authentication is available.
- Authentication Considerations for Database Installation, Upgrade and Downgrade Various authentication considerations apply to database installation, upgrades and downgrades.
- Automatic Installation of Oracle XML DB Oracle XML DB is automatically installed during an upgrade. Ensure that the compatibility level is at least 12.1.0.1 if you intend to use Oracle XML DB.
- Validation of ACL Documents and Configuration File
 During upgrade, existing ACL documents and the Oracle XML DB configuration file are
 validated against their XML schemas. If validation fails then you need to take certain
 measures.

See Also:

- Oracle Data Guard Concepts and Administration for information about performing a rolling upgrade
- ACL and ACE Evaluation for information about conflicts among ACEs

Authentication Considerations for Database Installation, Upgrade and Downgrade

Various authentication considerations apply to database installation, upgrades and downgrades.

- Authentication Considerations for a Database Installation
 In a default database installation, digest authentication is enabled, and basic authentication
 is disallowed. Digest verifiers are automatically generated for *all* users.
- Authentication Considerations for a Database Upgrade After an upgrade from a release prior to Oracle Database 12c Release 1 (12.1.0.1), digest authentication is appended to the list of allowed authentication mechanisms. But basic



authentication remains the current authentication method if it was enabled before the upgrade. This is for backward compatibility *only*.

Authentication Considerations for a Database Downgrade
 If you downgrade to a release where digest authentication was not supported, digest
 authentication is disabled and made unavailable as an authentication choice.

Related Topics

• Configuration and Management of Authentication Mechanisms for HTTP You configure the authentication mechanisms to allow for HTTP access to Oracle XML DB Repository by setting element authentication, a child of element httpconfig, in configuration file xdbconfig.xml.

Authentication Considerations for a Database Installation

In a default database installation, digest authentication is enabled, and basic authentication is disallowed. Digest verifiers are automatically generated for *all* users.

Authentication Considerations for a Database Upgrade

After an upgrade from a release prior to Oracle Database 12c Release 1 (12.1.0.1), digest authentication is appended to the list of allowed authentication mechanisms. But basic authentication remains the current authentication method if it was enabled before the upgrade. This is for backward compatibility *only*.

Oracle recommends that your database administrator *disable basic authentication as soon as possible* after upgrading. The reason that basic authentication remains in effect after such an upgrade is to allow users to change their passwords using a Web browser that does not support digest authentication.

For such an upgrade, digest verifiers are computed for all *new* users and for all previously existing *users whose passwords changed* during the upgrade. Other users do *not* have digest verifiers.

After an upgrade, a DBA can use database view DBA_DIGEST_VERIFIERS to check which users have digest verifiers and take appropriate action, as follows:

- Configure basic authentication as the first allowed authentication mechanism in the Oracle XML DB configuration file, xdbconfig.xml. This ensures that basic authentication can be used for HTTP access.
- 2. Expire all passwords for those users who do not have digest verifiers. This query returns those users:

SELECT USERNAME FROM DBA_DIGEST_VERIFIERS WHERE HAS DIGEST VERIFIERS = 'NO' AND DIGEST TYPE is NULL;

3. After the passwords for all such users have been changed, configure xdbconfig.xml to reenable digest as the first or (preferably) the only allowed authentication mechanism.

Authentication Considerations for a Database Downgrade

If you downgrade to a release where digest authentication was not supported, digest authentication is disabled and made unavailable as an authentication choice.

All digest verifiers are *erased* during a downgrade. This means, in particular, that if a downgrade is followed by an upgrade then users who were able to authenticate prior to the downgrade are denied digest authentication after the downgrade and the subsequent upgrade.



Automatic Installation of Oracle XML DB

Oracle XML DB is automatically installed during an upgrade. Ensure that the compatibility level is at least 12.1.0.1 if you intend to use Oracle XML DB.

If Oracle XML DB is not already installed in your database prior to an upgrade to Oracle Database 12c Release 1 (12.1.0.1) or later, then it is automatically installed in tablespace SYSAUX during the upgrade.

If Oracle XML DB has thus been automatically installed, and if you want to use Oracle XML DB, then, after the upgrade operation, you must set the database compatibility to at least 12.1.0.1. If the compatibility is less than 12.1.0.1 then an error is raised when you try to use Oracle XML DB.

If Oracle XML DB was automatically installed during an upgrade and the current compatibility level is less than 12.1.0.1, then Oracle DB is automatically uninstalled during any downgrade to a prior release.

Validation of ACL Documents and Configuration File

During upgrade, existing ACL documents and the Oracle XML DB configuration file are validated against their XML schemas. If validation fails then you need to take certain measures.

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 12c Release 1 (12.1.0.1), 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 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*.

Administration of Oracle XML DB Using Oracle Enterprise Manager

You can use Oracle Enterprise Manager to easily perform Oracle XML DB administration tasks. This includes configuring Oracle XML DB; creating, searching, editing, and deleting XML schemas, XMLType tables and views, and Oracle XML DB Repository *resources* and their *access control lists* (ACLs); registering XML schemas; and creating function-based indexes.

See the online help available with Oracle Enterprise Manager for information about using Enterprise Manager to perform the following tasks:

- Configure Oracle XML DB. View or edit parameters for the Oracle XML DB configuration file, xdbconfig.xml.
- Search, create, edit, undelete Oracle XML DB Repository resources and their associated access control lists (ACLs).
- Search, create, edit, and delete XMLType tables and views.
- Search, create, register, and delete *XML* schemas.

See Also:

- Configuration of Oracle XML DB Using xdbconfig.xml for information about configuring Oracle XML DB without using Oracle Enterprise Manager
- Oracle XML DB Repository for information about creating and managing resources without using Oracle Enterprise Manager
- XML Schema Storage and Query: Basic for information about manipulating XML schemas without using Oracle Enterprise Manager

Configuration of 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 the configuration file, xdbconfig.xml. You need administrator privileges to access file xdbconfig.xml.

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 the configuration file, xdbconfig.xml. You need administrator privileges to access file xdbconfig.xml.

The structure of the configuration file is described in the following sections.

- Element xdbconfig (Top-Level)
 Element xdbconfig is the top-level element of the Oracle XML DB configuration file.
- Element sysconfig (Child of xdbconfig) Element sysconfig is a child of xdbconfig.
- Element userconfig (Child of xdbconfig)
 Element userconfig is a child of xdbconfig. It contains any parameters that you may want to add.
- Element protocolconfig (Child of sysconfig) Element protocolconfig is a child of sysconfig.
- Element httpconfig (Child of protocolconfig)
 Element httpconfig, which is a child of protocolconfig, is described.
- Element servlet (Descendant of httpconfig)
 Element servlet is a descendent of element httpconfig. It is used to configure servlets, including Java servlets and embedded PL/SQL gateway servlets.
- Oracle XML DB Configuration File Example A sample Oracle XML DB configuration file is presented.
- 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. You can use 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).
- Configuration of Mappings from Default Namespace to Schema Location Oracle XML DB provides a mechanism to configure default XML schema location mappings.
- Configuration of 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 XML schema location (or uses the default mapping if present) and
 inserts the document into the appropriate default table.



See Also: xdbconfig.xsd: XML Schema for Configuring Oracle XML DB for a complete listing of the Oracle XML DB configuration XML schema

Element xdbconfig (Top-Level)

Element xdbconfig is the top-level element of the Oracle XML DB configuration file.

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.

Element 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). Child element protocolconfig contains protocol-specific parameters.

Note:

Element case-sensitive, child of element sysconfig, has no effect on the casesensitivity of XQuery or full-text search. Otherwise, it affects the behavior of all of Oracle XML DB.

Element userconfig (Child of xdbconfig)

Element userconfig is a child of xdbconfig. It contains any parameters that you may want to add.



Element 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 element 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.

Related Topics

 Repository Access Using Protocols You can access Oracle XML DB Repository data using protocols FTP and HTTP(S)/ WebDAV.

See Also:

Table 28-1, Table 28-2, and Table 28-3, for a list of protocol configuration parameters

Element httpconfig (Child of protocolconfig)

Element httpconfig, which is a child of protocolconfig, is described.

This is the structure of httpconfig:

```
<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 applicationspecific 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 element servlet-list, which has child element servlet (see Element servlet (Descendant of httpconfig)).

 plsql – used to define global configuration parameters when configuring the *embedded PL/SQL gateway*. Each global parameter is defined with a child element of 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.

Related Topics

Repository Access Using Protocols

You can access Oracle XML DB Repository data using protocols FTP and HTTP(S)/ WebDAV.

See Also:

- Table 28-1, Table 28-2, and Table 28-3, for a list of protocol configuration parameters
- Oracle Database Development Guide, for complete information about configuring and using the embedded PL/SQL gateway
- Oracle Database PL/SQL Packages and Types Reference, for information about package DBMS_EPG

Element servlet (Descendant of httpconfig)

Element servlet is a descendent of element httpconfig. It is used to configure servlets, including Java servlets and embedded PL/SQL gateway servlets.

See Element httpconfig (Child of protocolconfig).

Note:

The following servlets are preconfigured in file xdbconfig.xml. Do not delete them.

- ORSServlet
- EMExpressServlet

An optional element plsql, child of servlet¹, 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.

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*.

Element plsql has a child element for each embedded PL/SQL DAD attribute² 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.

💉 See Also:

- Guidelines for Oracle XML DB Applications in Java for information about configuring Java servlets
- Oracle Database Development Guide, for complete information about configuring and using the embedded PL/SQL gateway
- Oracle APEX App 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

A sample Oracle XML DB configuration file is presented.

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>
```

² DAD is an abbreviation for Database Access Descriptor. DAD attributes are parameters that define such a descriptor.

```
<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>200000000</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>
```



```
ORACLE
```

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. You can use 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).

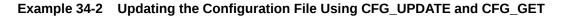
For convenience, you can use PL/SQL package DBMS_XDB_CONFIG package for configuration access. It exposes the following functions and procedures:

- 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.

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.





Related Topics

 DBMS_XDB_CONFIG: Configuration Management PL/SQL package DBMS_XDB_CONFIG provides Oracle XML DB configuration management functions and procedures.

Configuration of Mappings from Default Namespace to Schema Location

Oracle XML DB provides a mechanism to configure default XML schema location mappings.

Oracle XML DB identifies XML Schema-based XMLType instances by pre-parsing input XML documents. 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 nonschema-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:

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:

- Root element = root
 - Namespace = http://www.oracle.com/example
 - 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">

- Root element = null (any global element in the namespace)
 - Namespace = http://www.oracle.com/example2
 - Schema URL = http://www.oracle.com/example2/sch.xsd

This mapping is used when the instance document specifies:

```
<root xmlns="http://www.oracle.example2">
```

- Root element = specialRoot
 - Namespace = null (i.e null namespace)
 - 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.

Configuration of 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 XML 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:

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>
```

Oracle XML DB and Database Consolidation

Each pluggable database has its own Oracle XML DB Repository, and its own Oracle XML DB configuration file, xdbconfig.xml.

A multitenant container database (CDB) consists of zero or more pluggable databases (PDBs), a root, and a seed PDB (a template for creating PDBs). A given PDB can be associated with only one CDB at a time. A PDB appears to users and applications as a separate database. Your applications always interact with a single PDB at a time. Queries and dictionary views are local to a PDB. Each PDB has its own Oracle XML DB Repository, and its own Oracle XML DB configuration file, xdbconfig.xml.

The root of a CDB (CDB\$ROOT) contains no user data. It does, however, have its own configuration file, xdbconfig.xml.



The root configuration file for a CDB has only certain parameters, and those parameters are used only from the root configuration file. If any of those parameters are also present in a configuration file of a PDB that is part of a CDB, they are ignored in favor of the corresponding parameters in the root configuration file.

These are the configuration parameters that are used from the root (and ignored from any PDBs):

- acl-cache-size
- acl-max-age
- resource-view-cache-size
- xdbcore-loadableunit-size
- xdbcore-xobmem-bound

Parameters xdbcore-loadableunit-size and xdbcore-xobmem-bound are process-specific. The others are SGA configuration parameters.

Database schema (user account) XDB is a common user, which means that it can connect to, and perform operations within, both the root and any PDBs.

Note:

Oracle recommends that you never unlock database schema XDB, under any circumstances.

See Also:

- Oracle Multitenant Administrator's Guide and Oracle Multitenant Administrator's Guide for information about database consolidation and PDBs
- Performance Guidelines for Oracle XML DB Repository Operations for information about configuration parameters resource-view-cache-size, xdbcore-loadableunit-size, and xdbcore-xobmem-bound
- Considerations for Loading and Retrieving Large Documents with Collections for information about configuration parameters xdbcore-loadableunit-size and xdbcore-xobmem-bound
- ACL Caching for information about configuration parameter acl-max-age

Package DBMS_XDB_ADMIN

PL/SQL package DBMS_XDB_ADMIN has subprograms for managing and configuring Oracle XML DB and Oracle XML DB Repository.

Table 34-1 describes these subprograms.

Description
Move database schema (user account) XDB to the specified tablespace.
IndexRebuild the hierarchical repository index. This can be needed from time to time, in particular after invoking moveXDB_tablespace.
Note: Prior to Oracle Database 11g Release 2 (11.2.0.3), these procedures belonged to PL/SQL package DBMS_XDB. These two procedures in package DBMS_XDB are deprecated as of release 11.2.0.3.
Note:
The tablespace containing Oracle XML DB Repository <i>must not</i> be read-only, because that would adversely affect XML operations.
By default, Oracle XML DB Repository resides in tablespace SYSAUX, which is used for other things as well. Oracle recommends instead that you create a dedicated tablespace for use only by the repository.
Use procedure DBMS_XDB_ADMIN.moveXDB_tablespace to move database schema XDE to that tablespace dedicated to the repository. Ensure that the tablespace is not read only.

Oracle Database PL/SQL Packages and Types Reference

Table 34-1 DBMS_XDB_ADMIN Management Procedures

See Also:



The main way to load XML data into Oracle XML DB is to use SQL*Loader.

- Overview of Loading XMLType Data Into Oracle Database You can load XMLType data with SQL*Loader, using either the conventional method or the direct-path method, regardless of how it is stored (object-relational or binary XML storage).
- Load XMLType Data Using SQL*Loader
 SQL*Loader treats XMLType columns and tables like object-relational columns and tables.
 All methods 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.

Related Topics

 Overview of How To Use Oracle XML DB An overview of the various ways of using Oracle XML DB is presented.

Overview of Loading XMLType Data Into Oracle Database

You can load XMLType data with SQL*Loader, using either the conventional method or the direct-path method, regardless of how it is stored (object-relational or binary XML storage).

Starting with Oracle9i release 1 (9.0.1), the Export-Import utility and SQL*Loader support XMLType as a column type. Starting with Oracle Database 10g, SQL*Loader also supports loading XMLType tables.

Note:

For *object-relational 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.

Oracle XML DB Repository information is *not* exported when user data is exported. Neither the resources nor any information are exported.

Related Topics

Export and Import of Oracle XML DB Data

You can use Oracle Data Pump to export and import XMLType tables for use with Oracle XML DB.



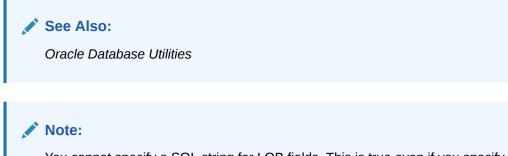
See Also:

Oracle Database Utilities

- Export and Import of Oracle XML DB Data
- Oracle Database Utilities

Load XMLType Data Using SQL*Loader

SQL*Loader treats XMLType columns and tables like object-relational columns and tables. All methods 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.



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).

- Load XMLType LOB Data 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 standard SQL*Loader formats.
- Load XMLType Data Directly from a Control File Using SQL*Loader You can load XMLType data directly from a control file. SQL*Loader treats XMLType data like any scalar type.
- Loading Large XML Documents Using SQL*Loader You can use SQL*Loader to load large amounts of XML data into Oracle Database.

Load XMLType LOB Data 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 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*

- Load LOB Data Using Predetermined Size Fields Predetermined size fields constitute a very fast and conceptually simple SQL*Loader format for loading LOBs.
- Load LOB Data Using Delimited Fields
 The delimited fields format handles LOBs of different sizes within the same column (datafile field). However, this added flexibility can affect performance, because SQL*Loader must scan through the data, looking for the delimiter string.
- Load XML Columns Containing LOB Data from LOBFILEs LOB data can be lengthy enough that it makes sense to load it from a LOBFILE instead of from a primary datafile.
- Specify 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).

Load LOB Data Using Predetermined Size Fields

Predetermined size fields constitute a very fast and conceptually simple SQL*Loader format for loading 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.

Load LOB Data Using Delimited Fields

The delimited fields format handles LOBs of different sizes within the same column (datafile field). 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.

Load XML Columns Containing LOB Data from LOBFILEs

LOB data can be lengthy enough 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.

Specify 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.

Load XMLType Data Directly from a Control File Using SQL*Loader

You can load XMLType data 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 Using SQL*Loader

You can use SQL*Loader to load large amounts of XML data into Oracle Database.

- 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 35-1, Example 35-2, and Example 35-3. File filelist.dat lists XML files that contain purchase orders for the year 2002.



If your application uses indexes or constraints then processing of these can impact loading performance. You can temporarily disable this processing using PL/SQL subprograms disableIndexesAndConstraints and enableIndexesAndConstraints in package DBMS_XMLSTORAGE_MANAGE.

See Also:

- Oracle Database Utilities for information about shell command sqlldr
- Oracle Database PL/SQL Packages and Types Reference for information about DBMS_XMLSTORAGE_MANAGE subprograms disableIndexesAndConstraints and enableIndexesAndConstraints

Example 35-1 Data File filelist.dat: List of XML Files to Load

```
2002/Jan/AMCEWEN-20021009123335370PDT.xm
2002/Jan/AWALSH-2002100912333570PDT.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
```



Related Topics

- Overview of How To Use Oracle XML DB An overview of the various ways of using Oracle XML DB is presented.
- Loading Large XML Files Using SQL*Loader You can use SQL*Loader to load large amounts of XML data into Oracle Database.



36

Export and Import of Oracle XML DB Data

You can use Oracle Data Pump to export and import XMLType tables for use with Oracle XML DB.

Note:

You can use the older export and import utilities exp and imp to migrate data to database releases that are prior to Oracle Database 11g. However, these older utilities do not support using XMLType data that is stored as binary XML.

- Overview of Exporting and Importing XMLType Tables
 Oracle XML DB supports export and import of XMLType tables and columns that store XML data, whether it is XML schema-based or not.
- Export/Import Limitations for Oracle XML DB Repository
 When you export or import tables that store data for Oracle XML DB Repository resources
 that are based on a registered XML schema, only that XML data is exported. The
 repository structure is lost during export, so that when these tables are imported they are
 not hierarchy-enabled.
- Export/Import Syntax and Examples Guidelines and examples are presented for using commands expdp and impdp with XMLType data.

Overview of Exporting and Importing XMLType Tables

Oracle XML DB supports export and import of XMLType tables and columns that store XML data, whether it is XML schema-based or not.

Oracle Data Pump enables high-speed movement of data and metadata from one database to another. There are two modes for using Oracle Data Pump: transportable tablespaces mode and non-transportable tablespaces mode.

For the transportable tablespaces mode there is this restriction regarding XMLType data: you cannot change the XMLType storage model.

As with other database objects, XML data is exported in the character set of the exporting server. During import, the data is converted to the character set of the importing server.

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.

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 using impdp).

Oracle XML DB supports export and import of XMLType tables and columns that store XML data, whether it is 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 object-relational storage.

Therefore, exporting a user who has XML schema-based XMLType tables also exports the following:

- SQL objects types (if object-relational storage was used)
- XML schemas
- XML tables

You can export and import this data regardless of the XMLType storage format (object-relational or binary XML). However, Oracle Data Pump exports and imports XML data as text or binary XML data only. The underlying tables and columns used for object-relational storage of XMLType are thus not exported. Instead, they are converted to binary form and then exported as self-describing binary XML data.

Note:

Oracle Data Pump for Oracle Database 11g 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 XMLType storage model, the format of the dump file is either text or selfdescribing binary XML with a token map preamble. By default, self-describing binary XML is used.

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.

Note:

Do not use option table_exists_action=append to import more than once from the same dump file into an XMLType table, regardless of the XMLType storage model used. Doing so raises a unique-constraint violation error because rows in XMLType tables are always exported and imported using a unique object identifier.

See Oracle Database Utilities for information about table exists action.

Export/Import Limitations for Oracle XML DB Repository

When you export or import tables that store data for Oracle XML DB Repository resources that are based on a registered XML schema, only that XML data is exported. The repository structure is lost during export, so that when these tables are imported they are not hierarchy-enabled.

You can export and import the XMLType tables that store the XML data for Oracle XML DB Repository resources that are based on a registered XML schema.

However, only the XML data is exported. The repository structure is lost during export. Relationships in the folder hierarchy, 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.

Export/Import Syntax and Examples

Guidelines and examples are presented for using commands expdp and impdp with XMLType data.

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.

Export and import using Oracle Data Pump is described fully in Oracle Database Utilities.

- Performing a Table-Mode Export /Import Examples are presented of performing a table-mode export and a table-mode import, to and from a dump file, respectively.
- Performing a Schema-Mode Export/Import
 Examples here perform schema-mode exporting and importing. 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.

Performing a Table-Mode Export /Import

Examples are presented of performing a table-mode export and a table-mode import, to and from a dump file, respectively.

An XMLType table has a dependency on the XML schema that was used to define it. Similarly, that XML schema has dependencies on the SQL object types that were created or specified for it. Importing an XMLType table requires the existence of the corresponding XML schema and 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, you must ensure that the relevant 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 schema user23
- A table user23.tab41 with an XMLType column stored as binary XML



 A directory object dpump_dir, for which READ and WRITE privileges have been granted to the user running expdp or impdp

Example 36-1 shows a table-mode export, specified using the TABLES parameter. It exports table tab41 to dump file tab41.dmp.

Note:

In table mode, if you do not specify a schema prefix in the expdp command then the schema of the exporter is used by default.

Example 36-2 shows a table-mode import. It imports table tab41 from dump file tab41.dmp.

If a table named tab41 already exists at the time of the import then specifying table_exists_action = append causes rows to be appended to that table. Whenever you use parameter value append the data is loaded into new space; existing space is never reused. For this reason you might need to compress your data after the load operation.

🖍 See Also:

Oracle Database Utilities, for more information about Oracle Data Pump and its command-line clients, expdp and impdp

Example 36-1 Exporting XMLType Data in TABLE Mode

expdp system directory=dpump dir dumpfile=tab41.dmp tables=user23.tab41

Example 36-2 Importing XMLType Data in TABLE Mode

```
impdp system tables=user23.tab41 directory=dpump_dir dumpfile=tab41.dmp
table_exists_action=append
```

Performing a Schema-Mode Export/Import

Examples here perform schema-mode exporting and importing. 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 examples here assume 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.

Table po2 is then populated and exported, as shown in Example 36-4.

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 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 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.

Note:

If you import an XML schema into the same database that it was exported from, and if that XML schema is still registered with Oracle XML DB at the time of importing, do not use remap_schema unless you also specify impdp parameter transform=oid:n. See Oracle Database Utilities for information about parameter transform.

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;
```

Example 36-4 Exporting XMLType Data in SCHEMA Mode

expdp x4a directory=tkxm xmldir dumpfile=x4a.dmp

Example 36-5 Importing XMLType Data in SCHEMA Mode

impdp x4a directory=tkxm xmldir dumpfile=x4a.dmp

Example 36-6 Importing XMLType Data in SCHEMA Mode, Remapping Schema

impdp x4a directory=tkxm_xmldir dumpfile=x4a.dmp remap_schema=x4a:quine



37

XML Data Exchange Using Oracle Database Advanced Queuing

You can exchange XML data using Oracle Database Advanced Queuing (AQ), which provides database-integrated message-queuing. AQ enables and manages asynchronous communication between 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.

XML and Oracle Database Advanced Queuing

Oracle Database Advanced Queuing (AQ) supports native XML messages. AQ operations can 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.

Oracle Database Advanced Queuing

Oracle Database Advanced Queuing (AQ) lets you share data and events in a queue. It can propagate information within a database or from one database to another, routing information to specified destinations. It provides functionality and flexibility for capturing and managing events, and for sharing events with other databases and applications.

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.

Internet Data Access Presentation (iDAP): SOAP for AQ

You can access Oracle Database Advanced Queuing (AQ) over the Internet using Simple Object Access Protocol (SOAP). Internet Data Access Presentation (iDAP) is the SOAP specification for AQ operations. iDAP defines XML message structure for a SOAP request body.

iDAP Architecture
 Oracle Database Advanced Queuing (AQ) operations that use HTTP(S) require an iDAP
 HTTP client, a Web server, and an Oracle server.

• Guidelines for Using XML and Oracle Database Advanced Queuing Guidelines are presented for using XML data with Oracle Database Advanced Queuing.

XML and Oracle Database Advanced Queuing

Oracle Database Advanced Queuing (AQ) supports native XML messages. AQ operations can 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.

 Oracle Database Advanced Queuing and XML Message Payloads XML messages can be passed asynchronously among applications using Oracle Database Advanced Queuing (AQ).



- Advantages of Using Oracle Database Advanced Queuing
 - Oracle Database Advanced Queuing (AQ) provides flexibility in configuring communication between applications. It makes an integrated solution easy to manage, easy to configure, and easy to modify, to meet changing business needs. It enables applications to cooperate, coordinate, and synchronize, to carry out complex business transactions.

Oracle Database Advanced Queuing and XML Message Payloads

XML messages can be passed asynchronously among applications using Oracle Database Advanced Queuing (AQ).

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

Use cases of passing XML messages 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.



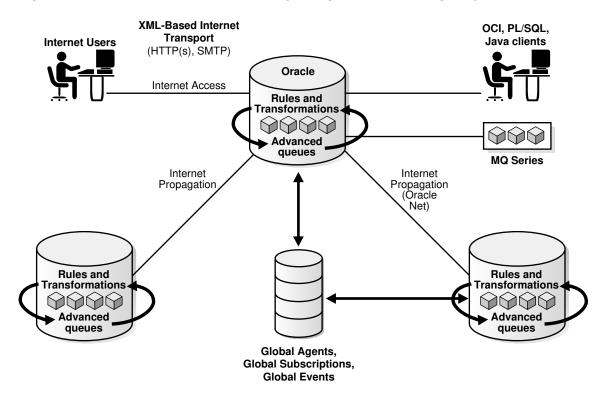


Figure 37-1 Oracle Database Advanced Queuing and XML Message Payloads

Advantages of Using Oracle Database Advanced Queuing

Oracle Database Advanced Queuing (AQ) provides flexibility in configuring communication between applications. It makes an integrated solution easy to manage, easy to configure, and easy to modify, to meet changing business needs. It enables 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 Database Advanced Queuing

Oracle Database Advanced Queuing (AQ) lets you share data and events in a queue. It can propagate information within a database or from one database to another, routing information to specified destinations. It provides functionality and flexibility for capturing and managing events, and for sharing events with other databases and applications.

AQ lets you break the cycle of trading off one solution for another. You can build and operate distributed enterprises and applications, data warehouses, and high availability solutions.

You can use AQ to do all of the following:

 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 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.

Message Queuing

Oracle Database Advanced Queuing (AQ) lets your applications enqueue, propagate, and dequeue messages.

Message Queuing

Oracle Database Advanced Queuing (AQ) lets your applications enqueue, propagate, and dequeue messages.

AQ 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. AQ supports all of the standard features of message queuing systems, including multi-consumer queues, publishing and subscribing, content-based routing, internet propagation, transformations, and gateways to other messaging subsystems.

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 instances
- 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): SOAP for AQ

You can access Oracle Database Advanced Queuing (AQ) over the Internet using Simple Object Access Protocol (SOAP). Internet Data Access Presentation (iDAP) is the SOAP



specification for AQ operations. iDAP defines XML message structure for a SOAP request body.

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 Database Advanced Queuing User's Guide

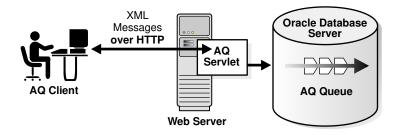
iDAP Architecture

Oracle Database Advanced Queuing (AQ) operations that use HTTP(S) require an iDAP HTTP client, a Web server, and an Oracle server.

Figure 37-2 shows the relationships among these components.

- A client program sends XML messages that conform to iDAP format, to the AQ Servlet. This can be any HTTP client, such as a Web browser.
- The Web server or ServletRunner hosts the AQ servlet that can interpret the incoming XML messages, for example, Apache/Jserv or Tomcat.
- Oracle AQ servlet connects to Oracle Database to perform queue operations.

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.



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 application sketched in the examples here, assume that an overseas shipping site represents an order using SYS.XMLType. An order-entry site represents an order as an Oracle object, ORDER TYP.

Example 37-1 creates the queue table and queue for overseas shipping.

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.

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 Database 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 37-1 Creating a Queue Table and Queue

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 XMLElement("Row", input order) INTO new order FROM DUAL;
```



```
RETURN new order;
END convert to order xml;
/
BEGIN
 SYS.DBMS TRANSFORM.create transformation (
   schema => 'OE',
                    'OE2XML',
   name =>
   from_schema => 'OE',
                    'ORDER TYP',
   from type =>
   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(
                => 'OS bookedorders_que',
    queue name
    subscriber
                  => subscriber,
                    => 'XMLSerialize(CONTENT XMLQuery(''//orderregion''' ||
    rule
                       'PASSING tab.user data RETURNING CONTENT)' ||
                       ' AS VARCHAR2(1000)) = ''INTERNATIONAL''',
     transformation => 'OE.OE2XML');
END;
/
```

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;

      po_massages
      EXCEPTION.

   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
                                                    => 'OS.OS bookedorders_que',
         DBMS AQ.dequeue(queue_name
                             dequeue_options => dopt,
                             message_properties => mprop,
                             payload => deq order data,
                             msqid
                                                     => 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;
ENDLOOP;
```

Guidelines for Using XML and Oracle Database Advanced Queuing

Guidelines are presented for using XML data with Oracle Database Advanced Queuing.

- Store AQ XML Messages with Many PDFs as One Record You can exchange XML documents between businesses using Oracle Database Advanced Queuing (AQ), where each message includes an XML header, an XML attachment (XML data stream), DTDs, and PDF files. The data can be stored in a database table, such as a queue table.
- Add New Recipients After Messages Are Enqueued You can use a queue table to support message assignments.
- Enqueue and Dequeue XML Messages
 Oracle Database Advanced Queuing (AQ) supports enqueuing and dequeuing objects.
 The objects can have an attribute of type XMLType that contains an XML document, in addition to having metadata attributes.
- Parse Messages with XML Content from AQ Queues You can parse messages with XML content from an Oracle Database Advanced Queuing (AQ) queue and then update tables and fields in an Operational Data Store (ODS).
- Prevent the Listener from Stopping Until an XML Document Is Processed After receiving a message, you can submit a job using PL/SQL package DBMS_JOB. The job is invoked asynchronously in a different database session. This can prevent messages accumulating in the queue because the listener must wait until a received XML message is processed.

HTTPS with AQ

You can use Oracle Database Advanced Queuing (AQ) Internet access to send XML messages to suppliers using HTTPS and receive a response. You can enqueue and dequeue XML messages over HTTP(S) securely and transactionally.

Store XML in Oracle AQ Message Payloads

You can store XML data in Oracle Database Advanced Queuing (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.

iDAP and SOAP

iDAP is the SOAP specification for Oracle Database Advanced Queuing (AQ) operations. SOAP defines a generic mechanism to invoke a service. iDAP defines these mechanisms to perform AQ operations.

Store AQ XML Messages with Many PDFs as One Record

You can exchange XML documents between businesses using Oracle Database Advanced Queuing (AQ), where each message includes an XML header, an XML attachment (XML data stream), DTDs, and PDF files. The data can be stored in a database table, such as a queue table.

You can enqueue the messages into Oracle queue tables as one record or piece. Or you can enqueue the messages as multiple records, for example, 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.

Add New Recipients After Messages Are Enqueued

You can use a 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 department Human Resources (HR). 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 predefined as subscribers for this queue.

You cannot change the recipient list after a 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.

Enqueue and Dequeue XML Messages

Oracle Database Advanced Queuing (AQ) supports enqueuing and dequeuing objects. The objects can have an attribute of type XMLType that contains an XML document, in addition to having metadata attributes.

Refer to Oracle Database Advanced Queuing User's Guide for specific details and more examples.

Parse Messages with XML Content from AQ Queues

You can parse messages with XML content from an Oracle Database Advanced Queuing (AQ) queue and then update tables and fields in an Operational Data Store (ODS).



You can use Oracle XML Parser for Java and Java Stored Procedures together with 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.

Prevent the Listener from Stopping Until an XML Document Is Processed

After receiving a message, you can submit a job using PL/SQL package DBMS_JOB. The job is invoked asynchronously in a different database session. This can prevent messages accumulating in the queue because the listener must wait until a received XML message is processed.

When receiving XML messages from clients you might 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 Database Advanced Queuing notification framework.

HTTPS with AQ

You can use Oracle Database Advanced Queuing (AQ) Internet access to send XML messages to suppliers using HTTPS and receive a response. You can enqueue and dequeue XML messages over HTTP(S) securely and transactionally.

💉 See Also:

Oracle Database Advanced Queuing User's Guide

Store XML in Oracle AQ Message Payloads

You can store XML data in Oracle Database Advanced Queuing (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.

iDAP and SOAP

iDAP is the SOAP specification for Oracle Database Advanced Queuing (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 VIII Appendixes

Appendixes here provide background material for Oracle XML DB.

Oracle-Supplied XML Schemas and Examples

Full listings are provided here for the Oracle XML DB-supplied XML schemas, purchaseorder XML schemas and an XSLT stylesheet used in various examples, and C-language (OCI) examples for loading XML content into Oracle XML DB and initializing and terminating an XML context.

- Oracle XML DB Restrictions
 The restrictions associated with Oracle XML DB are listed here.
- 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.

Indexing XML Data for Full-Text Queries (pre-23ai)

When you need full-text search over XML data, Oracle recommends that you store your XMLType data as binary XML and you use XQuery Full Text (XQFT). You use an XML search index for this. This is the topic of this section.

A

Oracle-Supplied XML Schemas and Examples

Full listings are provided here for the Oracle XML DB-supplied XML schemas, purchase-order XML schemas and an XSLT stylesheet used in various examples, and C-language (OCI) examples for loading XML content into Oracle XML DB and initializing and terminating an XML context.

- XDBResource.xsd: XML Schema for Oracle XML DB Resources A full listing is presented of the Oracle XML DB-supplied XML schema XDBResource.xsd, which is used to represent Oracle XML DB resources.
- XDBResConfig.xsd: XML Schema for Resource Configuration
 A full listing is presented of the Oracle XML DB-supplied XML schema used to configure
 repository resources. It is accessible in Oracle XML DB Repository at path /sys/schemas/
 PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd.
- acl.xsd: XML Schema for ACLs The Oracle Database-supplied XML schema used to represent access control lists (ACLs) is presented.
- xdbconfig.xsd: XML Schema for Configuring Oracle XML DB A full listing is presented of file xdbconfig.xsd, which contains the XML schema used to configure Oracle XML DB.
- xdiff.xsd: XML Schema for Comparing Schemas for In-Place Evolution
 A full listing is presented of xdiff.xsd, the Oracle XML DB-supplied XML schema to which
 the document specified as the diffXML parameter to procedure
 DBMS_XMLSCHEMA.inPlaceEvolve must conform.
- Purchase-Order XML Schemas The full listings of purchase-order XML schemas that are used in various examples are presented.
- XSLT Stylesheet Example, PurchaseOrder.xsl The full listing is presented of XSLT stylesheet PurchaseOrder.xsl, which is used in various examples.
- Loading XML Data Using C (OCI)
 A full listing of a C program that inserts XML data into an XMLType table is presented.
- Initializing and Terminating an XML Context (OCI) An example shows how to use OCI functions OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx() to initialize and terminate an XML context. It constructs an XML document using the C DOM API and saves it to the database.

XDBResource.xsd: XML Schema for Oracle XML DB Resources

A full listing is presented of 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"</pre>
        targetNamespace="http://xmlns.oracle.com/xdb/XDBResource.xsd" version="1.0"
        xdb:numProps="73" elementFormDefault="qualified" xdb:flags="23"
        xdb:mapStringToNCHAR="false" xdb:mapUnboundedStringToLob="false"
        xdb:storeVarrayAsTable="false" xdb:schemaOwner="XDB"
        xmlns="http://www.w3.org/2001/XMLSchema" xmlns:xdb="http://xmlns.oracle.com/xdb"
        xmlns:xdbres="http://xmlns.oracle.com/xdb/XDBResource.xsd">
  <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"</pre>
             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"</pre>
             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"</pre>
             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"</pre>
             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"</pre>
             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"</pre>
              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"</pre>
              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"</pre>
             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"</pre>
```

```
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"</pre>
         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"</pre>
         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"</pre>
             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"</pre>
             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"</pre>
             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"</pre>
             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"</pre>
             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"</pre>
             xdb:system="false" xdb:mutable="false" xdb:SQLName="RESCOMMENT"
```



<element xdb:propNumber="720" name="0wner" 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"/>

- 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"/>
- maxOccurs="1"/>
 <element xdb:propNumber="719" name="ACLOID" type="hexBinary" xdb:memType="23"</pre>
- <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"</pre>
- <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"/>
- <delaultlableschema= XDB xdb:humcols= 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="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"/>
- 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="ydb:ResMetaStr"</pre>
- 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"</pre>
- 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"

<element xdb:propNumber="721" name="OwnerID" type="xdb:GUID" xdb:memType="23"</pre> 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"</pre> 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"</pre> 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:SOLInline="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"</pre> 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:SOLInline="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"</pre> 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"</pre> 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"</pre> 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"</pre> 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"</pre> 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"</pre> 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="0" minOccurs="0" maxOccurs="1"/> <element xdb:propNumber="730" name="XMLLob" type="hexBinary" xdb:memType="113"</pre> 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="0" minOccurs="0" maxOccurs="1"/> <element xdb:propNumber="731" name="Flags" type="nonNegativeInteger"</pre> 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="0" minOccurs="1" maxOccurs="1"/> <element xdb:propNumber="740" name="VCRUID" type="xdb:GUID" xdb:memType="23"</pre> 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"</pre> 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"</pre> 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="0" minOccurs="0" maxOccurs="2147483647"/> <element xdb:propNumber="746" name="Snapshot" type="hexBinary" xdb:memType="23"</pre> 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"

	<pre>minOccurs="1" maxOccurs="1"/></pre>
<element< td=""><td>xdb:propNumber="747" name="AttrCopy" type="xdb:AttrCopyType"</td></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< td=""><td>xdb:propNumber="749" name="CtsCopy" type="hexBinary" xdb:memType="113"</td></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"
	<pre>minOccurs="0" maxOccurs="1"/></pre>
<element< td=""><td>xdb:propNumber="750" name="NodeNum" type="hexBinary" xdb:memType="23"</td></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="0"
	<pre>minOccurs="1" maxOccurs="1"/></pre>
<element< td=""><td>xdb:propNumber="751" name="ContentSize" type="integer"</td></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"
	<pre>xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0" maxOccurs="1"/></pre>
<element< td=""><td>xdb:propNumber="752" name="SizeOnDisk" type="nonNegativeInteger"</td></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"
	<pre>xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0" maxOccurs="1"/></pre>
<element< td=""><td>xdb:propNumber="754" name="RCList" type="xdb:RCListType" xdb:memType="258"</td></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"
	<pre>maxOccurs="1"/></pre>
<element< td=""><td>xdb:propNumber="762" name="Branch" type="string" xdb:memType="1"</td></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"
_	<pre>xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0" maxOccurs="1"/></pre>
<element< td=""><td>xdb:propNumber="763" name="CheckedOutBy" type="xdb:OracleUserName"</td></element<>	xdb:propNumber="763" name="CheckedOutBy" type="xdb:OracleUserName"

xdb:memType="1" xdb:system="false" xdb:mutable="false"

xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="0"

ORACLE

	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< td=""><td><pre>xdb:propNumber="764" name="CheckedOutByID" type="xdb:GUID" xdb:memType="23"</pre></td></element<>	<pre>xdb:propNumber="764" name="CheckedOutByID" type="xdb:GUID" xdb:memType="23"</pre>
	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"/>
Colomont	xdb:propNumber="765" name="BaseVersion" type="hexBinary" xdb:memType="23"
<erement.< td=""><td>xdb:system="false" xdb:mutable="false" xdb:SQLName="BASEVERSION"</td></erement.<>	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"
	<pre>minOccurs="0" maxOccurs="1"/></pre>
<element< td=""><td><pre>xdb:propNumber="766" name="Locks" type="xdb:locksType" xdb:memType="258"</pre></td></element<>	<pre>xdb:propNumber="766" name="Locks" type="xdb:locksType" xdb:memType="258"</pre>
	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"
	<pre>xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0" maxOccurs="1"/></pre>
<any td="" xdb<=""><td>:propNumber="735" name="ResExtra" xdb:memType="258" xdb:system="false"</td></any>	: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"/>
<td>\rangle</td>	\rangle
<attribute< td=""><td>xdb:propNumber="705" name="Hidden" type="boolean" xdb:memByteLength="1"</td></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< td=""><td>xdb:propNumber="706" name="Invalid" type="boolean" xdb:memByteLength="1"</td></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< td=""><td>xdb:propNumber="708" name="ActivityID" type="integer" xdb:memByteLength="4"</td></attribute<>	xdb:propNumber="708" name="ActivityID" type="integer" xdb:memByteLength="4"
(accribace	xdb:memType="3" xdb:system="false" xdb:mutable="false"
	xdb:SQLName="ACTIVITYID" xdb:SQLType="INTEGER" xdb:JavaType="long"/>
<pre>/attributa</pre>	xdb:propNumber="738" name="Container" type="boolean" xdb:memByteLength="1"
<i>AULIIDULE</i>	xdb:memType="252" xdb:system="false" xdb:mutable="true"
	xdb:JavaType="boolean" default="false" xdb:hidden="false"
	xdb:transient="generated" xdb:baseProp="false"/>
<attribute< td=""><td>xdb:propNumber="739" name="CustomRslv" type="boolean" xdb:memByteLength="1"</td></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"
	<pre>xdb:transient="generated" xdb:baseProp="false"/></pre>
<attribute< td=""><td>xdb:propNumber="742" name="VersionHistory" type="boolean"</td></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< td=""><td>xdb:propNumber="743" name="StickyRef" type="boolean" xdb:memByteLength="1"</td></attribute<>	xdb:propNumber="743" name="StickyRef" type="boolean" xdb:memByteLength="1"

xdb:JavaType="String" xdb:global="false" xdb:hidden="false"

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"</pre> 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"</pre> 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"</pre> 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"</pre> 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="777" name="IsXMLIndexed" type="boolean"</pre> xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="true" xdb:JavaType="boolean" default="false" xdb:hidden="true" xdb:transient="generated" xdb:baseProp="false"/> </complexType> <element xdb:propNumber="734" name="Resource" type="xdb:ResourceType" xdb:memType="258"</pre> 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"</pre> 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

A full listing is presented of the Oracle XML DB-supplied XML schema used to configure repository resources. It is accessible in Oracle XML DB Repository at path /sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd.

The use of repository events to trigger application actions is deprecated in Oracle Database 21c (21.3). There is no replacement.

Repository events are events that can be used to trigger application actions. Repository events include repository changes, such as 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 a resource. The deprecation of the use of repository events includes deprecation of the DBMS_XEVENT package, and the following subprogram groups:

XDBResConfig.xsd

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"</pre>
        targetNamespace="http://xmlns.oracle.com/xdb/XDBResConfig.xsd"
        xmlns:xdb="http://xmlns.oracle.com/xdb"
        xmlns:rescfg="http://xmlns.oracle.com/xdb/XDBResConfig.xsd"
        elementFormDefault="qualified" xdb:schemaOwner="XDB" 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" minOccurs="1" maxOccurs="1"/>
      <element name="namespace" type="string" minOccurs="0" maxOccurs="1"/>
    </all>
  </complexType>
  <!-- listener pre-condition element -->
  <complexType name="condition">
    <all>
      <element name="existsNode" type="rescfg:existsNode" minOccurs="0" maxOccurs="1"/>
    </all>
  </complexType>
```

</complexType>

```
A-12
```

```
<complexType name="events">
    <all>
      <element name="Render" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-Create" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-Create" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-Delete" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-Delete" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-Update" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-Update" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-Lock" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-Lock" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-Unlock" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-Unlock" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-LinkIn" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-LinkIn" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-LinkTo" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-LinkTo" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-UnlinkIn" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-UnlinkIn" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-UnlinkFrom" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-UnlinkFrom" type="string" minOccurs="0"/ maxOccurs="1">
      <element name="Pre-CheckIn" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-CheckIn" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-CheckOut" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-CheckOut" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-UncheckOut" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-UncheckOut" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-VersionControl" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-VersionControl" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-Open" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-Open" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Pre-InconsistentUpdate" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="Post-InconsistentUpdate" type="string" minOccurs="0" maxOccurs="1"/>
    </all>
  </complexType>
  <!-- event listener element -->
  <complexType name="event-listener">
    <all>
      <element name="description" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="schema" type="string" minOccurs="0" maxOccurs="1"/>
      <element name="source" type="string" minOccurs="1" maxOccurs="1"/>
      <element name="language" type="rescfg:language" minOccurs="0" maxOccurs="1"/>
      <element name="pre-condition" type="rescfg:condition" minOccurs="0" maxOccurs="1"/>
      <element name="events" type="rescfg:events" minOccurs="0" maxOccurs="1"/>
    </all>
  </complexType>
  <complexType name="event-listeners">
    <sequence>
      <element name="listener" type="rescfg:event-listener" minOccurs="1"</pre>
maxOccurs="unbounded"/>
    </sequence>
    <attribute name="default-schema" type="string" xdb:baseProp="true" use="optional"/>
    <attribute name="default-language" type="rescfg:language" xdb:baseProp="true"
use="optional"/>
    <attribute name="set-invoker" type="boolean" xdb:baseProp="true" default="false"/>
```

```
<complexType name="defaultPath">
    <all>
      <element name="pre-condition" type="rescfg:condition" minOccurs="0" maxOccurs="1"/>
      <element name="path" type="string" minOccurs="0" maxOccurs="1"</pre>
xdb:transient="generated"/>
      <element name="resolvedpath" type="string" minOccurs="1" maxOccurs="1"</pre>
               xdb:baseProp="true" xdb:hidden="true"/>
      <element name="oid" type="hexBinary"minOccurs="1" maxOccurs="1"</pre>
               xdb:baseProp="true" xdb:hidden="true"/>
    </all>
  </complexType>
  <complexType name="defaultACL">
    <sequence>
      <element name="ACL" type="rescfg:defaultPath" minOccurs="1" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
  <complexType name="defaultConfig">
    <sequence>
      <element name="configuration" type="rescfg:defaultPath"</pre>
               minOccurs="1" 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" minOccurs="0" default="OID"/>
      <element name="LinkMetadata" type="rescfg:link-metadata" minOccurs="0" default="None"/>
     <element name="pre-condition" type="rescfg:condition" minOccurs="0 "maxOccurs="1"/>
    </sequence>
    <attribute name="UnresolvedLink" type="resofg: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" minOccurs="0" default="OID"/>
      <element name="ConflictRule" type="rescfg:conflict-rule" minOccurs="0"</pre>
default="Error"/>
    </sequence>
    <attribute name="UnresolvedLink" type="resofg: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"</pre>
maxOccurs="1"/>
      <element name="defaultChildACL" type="rescfq:defaultACL" minOccurs="0" maxOccurs="1"/>
```

```
<element name="event-listeners" type="rescfg:event-listeners" minOccurs="0"</pre>
maxOccurs="1"/>
      <element name="XLinkConfig" type="rescfg:xlink-config" minOccurs="0" maxOccurs="1"/>
      <element name="XIncludeConfig" type="rescfg:xinclude-config" minOccurs="0"</pre>
maxOccurs="1"/>
      <element name="SectionConfig" type="rescfg:section-config" minOccurs="0"</pre>
maxOccurs="1"/>
      <element name="ContentFormat" type="rescfg:content-format" minOccurs="0"</pre>
maxOccurs="1"/>
      <!-- application data -->
      <element name="applicationData" minOccurs="0" maxOccurs="1">
        <complexType>
          <sequence>
            <any namespace="##other" maxOccurs="unbounded" processContents="lax"/>
          </sequence>
        </complexType>
      </element>
    </all>
    <attribute name="enable" type="boolean" xdb:baseProp="true" 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

The Oracle Database-supplied XML schema used to represent access control lists (ACLs) is presented.

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="gualified">
  <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"</pre>
         xdb:defaultTable=""/>
<!-- all system privileges in the XDB ACL namespace -->
<element name = "read-properties" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "read-contents" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "read-acl" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "update" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "link" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "unlink" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "unlink-from" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "write-acl-ref" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "update-acl" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "link-to" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "resolve" type="xdbacl:privilegeNameType"</pre>
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "all" type="xdbacl:privilegeNameType"</pre>
         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"</pre>
                       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"</pre>
           processContents="lax"/>
      <!-- HIDDEN ELEMENTS -->
      <choice minOccurs="0">
        <element name = "principalID" type = "hexBinary"</pre>
                 xdb:baseProp="true" xdb:hidden="true"/>
        <element name = "principalString" type = "string"</pre>
                 xdb:baseProp="true" xdb:hidden="true"/>
      </choice>
      <element name = "flags" type = "unsignedInt" minOccurs="0"</pre>
               xdb:baseProp="true" xdb:hidden="true"/>
    </sequence>
    <attribute name = "collection" type = "boolean"</pre>
               xdb:transient="generated" use="optional"/>
    <attribute name = "principalFormat"</pre>
               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"</pre>
             xdb:transient="generated"/>
    <element name = "elementName" type = "string" minOccurs="0"</pre>
             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"</pre>
          processContents="lax"/>
    <!-- HIDDEN ELEMENTS -->
     <element name = "schemaOID" type = "hexBinary" minOccurs="0"</pre>
              xdb:baseProp="true" xdb:hidden="true"/>
     <element name = "elementNum" type = "unsignedInt" minOccurs="0"</pre>
              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"</pre>
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
</schema>
```

xdbconfig.xsd: XML Schema for Configuring Oracle XML DB

A full listing is presented of file xdbconfig.xsd, which contains the 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"</pre>
        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"</pre>
                       default="3600"/>
              <element name="xdbcore-logfile-path" type="string"</pre>
                       default="/sys/log/xdblog.xml"/>
              <element name="xdbcore-log-level" type="unsignedInt"</pre>
                       default="0"/>
              <element name="resource-view-cache-size" type="unsignedInt"</pre>
                       default="1048576"/>
              <element name="case-sensitive-index-clause" type="string"</pre>
                       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"</pre>
                                          type="xdbc:mime-mapping-type"/>
                                 <element name="lang-mappings"</pre>
                                          type="xdbc:lang-mapping-type"/>
                                 <element name="charset-mappings"</pre>
                                          type="xdbc:charset-mapping-type"/>
                                 <element name="encoding-mappings"</pre>
```

```
type="xdbc:encoding-mapping-type"/>
            <element name="xml-extensions"</pre>
                      type="xdbc:xml-extension-type"
                      minOccurs="0"/>
          </sequence>
        </complexType>
      </element>
      <element name="session-pool-size" type="unsignedInt"</pre>
               default="50"/>
      <element name="session-timeout" type="unsignedInt"</pre>
               default="6000"/>
    </sequence>
  </complexType>
</element>
<!-- FTP specific -->
<element name="ftpconfig">
  <complexType>
    <sequence>
      <element name="ftp-port" type="unsignedShort"</pre>
               default="2100"/>
      <element name="ftp-listener" type="string"/>
      <element name="ftp-protocol" type="string"/>
      <element name="logfile-path" type="string"</pre>
               default="/sys/log/ftplog.xml"/>
      <element name="log-level" type="unsignedInt"</pre>
               default="0"/>
      <element name="session-timeout" type="unsignedInt"</pre>
               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"</pre>
               minOccurs="0" maxOccurs="1"/>
      <element name="host-name" type="string"</pre>
               minOccurs="0" maxOccurs="1"/>
    </sequence>
  </complexType>
</element>
<!-- HTTP specific -->
<element name="httpconfig">
  <complexType>
    <sequence>
      <element name="http-port" type="unsignedShort"</pre>
               default="8080"/>
      <element name="http-listener" type="string"/>
      <element name="http-protocol" type="string"/>
      <element name="max-http-headers" type="unsignedInt"</pre>
               default="64"/>
      <element name="max-header-size" type="unsignedInt"</pre>
```

```
default="4096"/>
<element name="max-request-body" type="unsignedInt"</pre>
         default="200000000" minOccurs="1"/>
<element name="session-timeout" type="unsignedInt"</pre>
         default="6000"/>
<element name="server-name" type="string"/>
<element name="logfile-path" type="string"</pre>
         default="/sys/log/httplog.xml"/>
<element name="log-level" type="unsignedInt"</pre>
         default="0"/>
<element name="servlet-realm" type="string"</pre>
         minOccurs="0"/>
<element name="webappconfig">
  <complexType>
    <sequence>
      <element name="welcome-file-list"</pre>
                type="xdbc:welcome-file-type"/>
      <element name="error-pages"</pre>
                type="xdbc:error-page-type"/>
      <element name="servletconfig"</pre>
                type="xdbc:servlet-config-type"/>
    </sequence>
  </complexType>
</element>
<element name="default-url-charset" type="string"</pre>
         minOccurs="0"/>
<element name="http2-port" type="unsignedShort"</pre>
         minOccurs="0"/>
<element name="http2-protocol" type="string"</pre>
         default="tcp" minOccurs="0"/>
<element name="plsql" minOccurs="0">
  <complexType>
    <sequence>
      <element name="log-level"</pre>
                type="unsignedInt" minOccurs="0"/>
      <element name="max-parameters"</pre>
                type="unsignedInt" minOccurs="0"/>
    </sequence>
  </complexType>
</element>
<element name="allow-repository-anonymous-access"</pre>
         minOccurs="0" default="false" type="boolean"/>
<element name="authentication" minOccurs="0"</pre>
         maxOccurs="1">
  <complexType>
    <sequence>
      <element name="allow-mechanism" minOccurs="1"</pre>
               maxOccurs="unbounded">
        <simpleType>
          <restriction base="string">
            <enumeration value="digest"/>
            <enumeration value="basic" />
            <enumeration value="custom"/>
            <enumeration value="digestMD5"/>
            <enumeration value="digestSHA256" />
            <enumeration value="digestSHA512"/>
```

```
</restriction>
              </simpleType>
            </element>
            <element name="digest-auth" minOccurs="0"</pre>
                      maxOccurs="1">
              <complexType>
                 <sequence>
                   <element name="nonce-timeout"</pre>
                            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"</pre>
                type="xdbc:custom-authentication-type"
               minOccurs="0"/>
      <element name="realm" type="string" minOccurs="0"/>
      <element name="respond-with-server-info" type="boolean"</pre>
               default="true" minOccurs="0"/>
      <element name="expire" type="xdbc:expire-type"</pre>
               minOccurs="0"/>
      <element name="white-list" minOccurs="0">
        <complexType>
          <sequence>
            <element name="white-list-pattern" minOccurs="0"</pre>
                      maxOccurs="unbounded">
              <simpleType>
                <restriction base="string">
                   <pattern value="(/[^\*/]+)*(/\*)?"/>
                </restriction>
              </simpleType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
<element name="nfsconfig" minOccurs="0">
  <complexType>
    <sequence>
      <element name="nfs-port" type="unsignedShort"</pre>
               default="2049"/>
      <element name="nfs-listener" type="string"/>
      <element name="nfs-protocol" type="string"/>
      <element name="logfile-path" type="string"</pre>
                default="/sys/log/nfslog.xml"/>
      <element name="log-level" type="unsignedInt"</pre>
               default="0"/>
      <element name="nfs-exports"</pre>
```

```
type="xdbc:nfs-exports-type"/>
                 </sequence>
               </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
      <element name="schemaLocation-mappings"</pre>
                type="xdbc:schemaLocation-mapping-type" minOccurs="0"/>
      <element name="xdbcore-xobmem-bound" type="unsignedInt"</pre>
                default="1024" minOccurs="0"/>
      <element name="xdbcore-loadableunit-size" type="unsignedInt"</pre>
                default="16" minOccurs="0"/>
      <element name="folder-hard-links" type="boolean" default="false"</pre>
               minOccurs="0"/>
      <element name="non-folder-hard-links" type="boolean" default="true"</pre>
               minOccurs="0"/>
      <element name="copy-on-inconsistent-update" type="boolean"</pre>
               default="false" minOccurs="0"/>
      <element name="rollback-on-sync-error" type="boolean"</pre>
                default="false" minOccurs="0"/>
      <element name="acl-evaluation-method" default="deny-trumps-grant"</pre>
               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"</pre>
               minOccurs="0"/>
      <element name="allow-authentication-trust" type="boolean"</pre>
                default="false" minOccurs="0"/>
      <element name="custom-authentication-trust"</pre>
                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"</pre>
                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"</pre>
                   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/SOL"/>
                    </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"</pre>
                           minOccurs="0"/>
                </choice>
                <element name="servlet-schema" type="string" minOccurs="0"/>
                <element name="init-param" minOccurs="0"</pre>
                         maxOccurs="unbounded" type="xdbc:param"/>
                <element name="load-on-startup" type="string" minOccurs="0"/>
                <element name="security-role-ref" minOccurs="0"</pre>
                         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>
                <!-- session-state-cache-param captures all the parameters
                     of the session state cache. expiration-timeout is
                     specified in centi-seconds. -->
                <element name="session-state-cache-param" minOccurs="0">
                  <complexType>
                    <sequence>
                      <element name="cache-size" type="unsignedInt"/>
                      <element name="expiration-timeout" type="unsignedInt"/>
                    </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"</pre>
             minOccurs="0" maxOccurs="unbounded">
   </element>
 </sequence>
</complexType>
<complexType name="schemaLocation-mapping-type">
 <sequence>
   <element name="schemaLocation-mapping"</pre>
             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"</pre>
         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"</pre>
        maxOccurs="unbounded"/>
<element name="cgi-environment-list" type="string" minOccurs="0"</pre>
        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"</pre>
         maxOccurs="unbounded"/>
<element name="bind-bucket-widths" type="unsignedInt" minOccurs="0"</pre>
        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"/>
          <enumeration value="SecurityOn"/>
          <enumeration value="SecurityOff"/>
        </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"</pre>
                                  type="xdbc:ipaddress"/>
                        <element name="nfs-client-dnsname" type="string"/>
                        <element name="nfs-client-address"</pre>
                                  type="xdbc:ipaddress"/>
                      </choice>
                      <element name="nfs-client-netmask"</pre>
                                type="xdbc:ipaddress"/>
                    </sequence>
```

```
</complexType>
                </element>
              </sequence>
            </complexType>
          </element>
          <element name="nfs-export-paths">
            <complexType>
              <sequence>
                <element name="nfs-export-path" minOccurs="1"</pre>
                         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"</pre>
                   maxOccurs="unbounded">
            <complexType>
              <sequence>
                <element name="authentication-pattern" type="string"/>
                <element name="authentication-name" type="string"/>
                <element name="authentication-trust-name" type="string"</pre>
                         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"</pre>
                         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"</pre>
             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"</pre>
                   minOccurs="0"/>
          <element name="allowRegistration" type="boolean" default="true"</pre>
                   minOccurs="0"/>
          <element name="trust-scheme-description" type="string" minOccurs="0"/>
          <element name="trusted-session-user" type="string" minOccurs="1"</pre>
                   maxOccurs="unbounded"/>
          <element name="trusted-parsing-schema" type="string" minOccurs="0"</pre>
                   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

A full listing is presented of xdiff.xsd, 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"</pre>
    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:
        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.
```

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.

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

The full listings of purchase-order XML schemas that are used in various examples are presented.

Example A-1 shows an unannotated purchase-order XML schema.

Example A-1 Unannotated Purchase-Order XML Schema

```
<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>
```

Example A-3 represents a modified version of Example A-2. The modification is used in XML Schema Evolution to illustrate XML schema evolution. Example A-2 is the complete listing of the annotated XML schema used in examples of XML Schema Storage and Query: Basic.

Example A-2 Annotated Purchase-Order XML Schema



```
<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"</pre>
               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"</pre>
               xdb:SQLName="DESCRIPTION"/>
   <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"</pre>
               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">

Example A-3 is the complete listing of the revised annotated XML schema presented in Example 20-1. Text that is in **bold face** is additional or significantly different from that in the schema of Example A-2.



```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"</pre>
           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"</pre>
                  xdb:SQLName="BILLING_ADDRESS"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                  xdb:SQLName="SHIPPING INSTRUCTIONS"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"</pre>
                 xdb:SQLName="SPECIAL INSTRUCTIONS"/>
      <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
      <xs:element name="Notes" type="NotesType" minOccurs="0" xdb:SQLType="CLOB"</pre>
                  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"</pre>
                  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"</pre>
                  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"</pre>
```

```
xdb:SQLName="DESCRIPTION"/>
      <rs: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"</pre>
                  xdb:SQLName="SHIP TO ADDRESS"/>
   </r>
   <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>
<rs:complexType name="AddressType" xdb:SQLType="ADDRESS_T">
  <xs:sequence>
    <rs:element name="StreetLine1" type="StreetType"/>
    <xs:element name="StreetLine2" type="StreetType" minOccurs="0"/>
    <xs:element name="City" type="CityType"/>
    <xs:choice>
      <xs:sequence>
        <rs:element name="State" type="StateType"/>
        <xs:element name="ZipCode" type="ZipCodeType"/>
      </xs:sequence>
      <xs:sequence>
        <rs: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"/> <rs: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="TA"/> <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"/>
      <rs:maxLength value="32"/>
   </xs:restriction>
 </xs:simpleType>
  <xs:simpleType name="PostCodeType">
   <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="12"/>
   </ms: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">
      <rs: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>
```

XSLT Stylesheet Example, PurchaseOrder.xsl

The full listing is presented of XSLT stylesheet PurchaseOrder.xsl, which is used in various examples.

Example A-4 PurchaseOrder.xsl XSLT Stylesheet

```
<?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">
        <P>
                 <B>
                   <FONT SIZE="+1" COLOR="#FF0000"
                   FACE="Arial, Helvetica, sans-serif">Internal
                   </FONT>
                 </B>
                </P>
                <B>
```

```
<FONT COLOR="#FFFF00">Actions</FONT>
  </B>
 <xsl:for-each select="Actions">
    <xsl:for-each select="Action">
     <xsl:if test="position()=1">
        <thead>
          <FONT
             COLOR="#FFFF00">User</FONT>
           <FONT
             COLOR="#FFFF00">Date</FONT>
           </t.r>
        </thead>
       </xsl:if>
       <xsl:for-each select="User">
             <xsl:apply-templates/>
           </xsl:for-each>
          <xsl:for-each select="Date">
             <xsl:apply-templates/>
           </xsl:for-each>
          </xsl:for-each>
  </xsl:for-each>
 <t.r>
 <B>
    <FONT COLOR="#FFFF00">Requestor</FONT>
  </B>
 <xsl:for-each select="Requestor">
    <xsl:apply-templates/>
   </xsl:for-each>
 </t.r>
<B>
    <FONT COLOR="#FFFF00">User</FONT>
```

```
</B>
     <xsl:for-each select="User">
        <xsl:apply-templates/>
       </xsl:for-each>
     </t.r>
    <B>
        <FONT COLOR="#FFFF00">Cost Center</FONT>
       </B>
     <xsl:for-each select="CostCenter">
        <xsl:apply-templates/>
       </xsl:for-each>
     </t.d>
    <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">
    <B>Name</B>
        </t.d>
        <xsl:for-each
           select="../ShippingInstructions">
         <xsl:for-each select="name">
            <xsl:apply-templates/>
           </xsl:for-each>
         </xsl:for-each>
       <B>Address</B>
        <xsl:for-each
            select="../ShippingInstructions">
```

```
<xsl:for-each select="address">
                  <span style="white-space:pre">
                   <xsl:apply-templates/>
                  </span>
                </xsl:for-each>
               </xsl:for-each>
            <B>Telephone</B>
             <xsl:for-each
                 select="../ShippingInstructions">
               <xsl:for-each select="telephone">
                  <xsl:apply-templates/>
                </xsl:for-each>
               </xsl:for-each>
            </xsl:if>
     </xsl:for-each>
    <br/>
<B>
 <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
</B>
<br/>
<br/>
<xsl:for-each select="LineItems">
   <xsl:for-each select="LineItem">
    <xsl:if test="position()=1">
     <thead>
       <t.d>
          <FONT COLOR="#FF0000">
           <B>ItemNumber</B>
          </FONT>
         <FONT COLOR="#FF0000">
           <B>Description</B>
          </FONT>
         </t.d>
         <FONT COLOR="#FF0000">
            <B>PartId</B>
          </FONT>
```

```
<FONT COLOR="#FF0000">
        <B>Quantity</B>
       </FONT>
     <FONT COLOR="#FF0000">
        <B>Unit Price</B>
       </FONT>
     <FONT COLOR="#FF00000">
        <B>Total Price</B>
       </FONT>
     </thead>
</xsl:if>
<FONT COLOR="#000000">
       <xsl:for-each select="@ItemNumber">
         <xsl:value-of select="."/>
       </xsl:for-each>
     </FONT>
   <FONT COLOR="#000000">
       <xsl:for-each select="Description">
        <xsl:apply-templates/>
       </xsl:for-each>
     </FONT>
   <t.d>
     <FONT COLOR="#000000">
       <xsl:for-each select="Part">
         <xsl:for-each select="@Id">
          <xsl:value-of select="."/>
         </xsl:for-each>
       </xsl:for-each>
     </FONT>
   <t.d>
     <FONT COLOR="#000000">
       <xsl:for-each select="Part">
         <xsl:for-each select="@Quantity">
          <xsl:value-of select="."/>
         </xsl:for-each>
       </xsl:for-each>
     </FONT>
   <t.d>
     <FONT COLOR="#000000">
       <xsl:for-each select="Part">
```

```
<xsl:for-each select="@UnitPrice">
                            <xsl:value-of select="."/>
                          </xsl:for-each>
                        </xsl:for-each>
                      </FONT>
                    <FONT FACE="Arial, Helvetica, sans-serif"
                           COLOR="#000000">
                        <xsl:for-each select="Part">
                          <xsl:value-of select="@Quantity*@UnitPrice"/>
                        </xsl:for-each>
                      </FONT>
                    </xsl:for-each>
             </xsl:for-each>
           </xsl:for-each>
       </FONT>
     </body>
   </html>
 </xsl:template>
</xsl:stylesheet>
```

Loading XML Data Using C (OCI)

A full listing of a C program that inserts XML data into an XMLType table is presented.

The same program is partially listed in Loading XML Content Using C.

Example A-5 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)

An example shows how to use OCI functions <code>OCIXmlDbInitXmlCtx()</code> and <code>OCIXmlDbFreeXmlCtx()</code> to initialize and terminate an XML context. It constructs an XML document using the C DOM API and saves it to the database.

Example A-6 is partially listed in C DOM API for XMLType and Initializing and Terminating an XML Context. It assumes that the following SQL code has first been executed to create table my_table in database schema CAPIUSER:

CONNECT CAPIUSER Enter password: cpassword>

Connected.

CREATE TABLE my_table OF XMLType;

Example A-6 Using OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx()

#ifndef S ORACLE #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 */ static sword exec bind xml (OCISvcCtx *svchp, OCIError *errhp, OCIStmt *stmthp, void *xml, OCIType *xmltdo, OraText *sqlstmt); /* Helper function 2: Initialize OCI handles and connect */ static sword init_oci_handles(test ctx *ctx); /* Helper function 3: Free OCI handles and disconnect */ static 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. The changes are reflected here ^{\star/}
 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. 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 */
static 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 */
static 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 */
static 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

The restrictions associated with Oracle XML DB are listed here.

- Thin JDBC Driver Not Supported by Some XMLType Functions XMLType method transform() works 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, NVARCHAR2, and NCLOB Not Supported Oracle XML DB does not support the use of SQL data types NCHAR, NVARCHAR2, 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 (desupported), 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 a maximum of 32767 bytes or 4000 bytes, depending on the value of initialization parameter MAX STRING SIZE. See Oracle Database SQL Language Reference.
- *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 Column-Level Encryption for XMLType Column-level encryption is not supported for XMLType. Tablespace-level encryption is supported for all XMLType storage models.
- No Composite Partitioning for XMLType Composite partitioning is not supported for XMLType tables or columns (regardless of the XMLType storage model).
- No Partitioning for Hierarchically Enabled Tables You cannot partition a hierarchyenabled table. (See Repository Resources and Database Table Security for information about hierarchy-enabled tables.)
- No Oracle Real Application Testing (RAT) for XMLType- Oracle Real Application Testing (RAT) is not 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 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.
- Transportable tablespaces and database consolidation If your Oracle XML DB Repository in Oracle Database 11g Release 2 (11.2) has existing data then you cannot use transportable tablespaces to plug that database directly into a container database (CDB). Instead, upgrade the 11.2 database to 12.1, unplug it, and then plug it in.
- No XMLType virtual columns for object-relational storage of XMLType You cannot add a virtual XMLType column to an object-relational XMLType table or to a table that has an object-relational XMLType column.
- No sharding for XMLType or repository Sharding is not supported for XMLType data or for Oracle XML DB Repository.
- XML schema URL uniqueness ignores protocol part Attribute schemaurl for a registered XML schema must be unique apart from the protocol part. The protocol part (for example, http or https) is ignored in the test for uniqueness.

Related Topics

 Oracle XML DB Support for XQuery Oracle XML DB support for the XQuery language includes SQL support and support for XQuery functions and operators.



C 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.

Note:

Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes. Oracle recommends that you recreate unstructured XML indexes as XML search indexes and use it alongside Transportable Binary XML.

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 C-1 shows the main information¹ that is in the 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.

Table C-1 XMLIndex Path Table

Tasks Involving XMLIndex Indexes with an Unstructured Component identifies the documentation for some user tasks involving XMLIndex indexes that have an *unstructured* component.

Table C-2 Tasks Involving XMLIndex Indexes with an Unstructured Component

For information about how to	See
Create an XMLIndex index with an unstructured component	Example C-2, Example C-4, Example C-17, Example C-19, Example 6-17, Example 6-18, Example C-15
Drop the unstructured component of an XMLIndex index (drop the path table)	Example C-5
Name the path table when creating an XMLIndex index	Example C-2
Specify storage options when creating an XMLIndex index	Example C-4
Show all existing secondary indexes on an XMLIndex path table	Example C-6, Example C-14
Obtain the name of a path table for an XMLIndex index	Example C-3
Obtain the name of an XMLIndex index with an unstructured component, given its path table	Example C-7
Create a secondary index on an XMLIndex path table	Using XMLIndex with an Unstructured Component
Obtain information about all of the secondary indexes on an XMLIndex path table	Example C-14
Create a function-based index on a path-table VALUE column	Example C-9

¹ The actual path table implementation may be slightly different.

Table C-2 (Cont.) Tasks Involving XMLIndex Indexes with an Unstructured Component

For information about how to	See
Create a numeric index on a path-table VALUE column	Example C-11
Create a date index on a path-table VALUE column	Example C-12
Create an Oracle Text CONTEXT index on a path-table VALUE column	Example C-13
Exclude or include particular XPath expressions from use by an ${\tt XMLIndex}$ index	XMLIndex Path Subsetting: Specifying the Paths You Want to Index
Specify namespace prefixes for XPath expressions used for XMLIndex	XMLIndex Path Subsetting: Specifying the Paths You Want to Index
Exclude or include particular XPath expressions from use by an XMLIndex index	XMLIndex Path Subsetting: Specifying the Paths You Want to Index
Specify namespace prefixes for XPath expressions used for ${\tt XMLIndex}$	XMLIndex Path Subsetting: Specifying the Paths You Want to Index

Figure C-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	Variable, free-form data, with some fixed embedded structures	Variable, free-form data
Typical Data	Employee record	Technical article, with author, date, and title fields	Web document or book chapter
Storage Model	Object-Relational (Structured)	Binary XML	
Indexing	B-tree index	XMLIndex index with structured and unstructured components XML search index	XMLIndex index with unstructured component XML search index

If you need to support ad-hoc XML queries that involve predicates, then you can use XMLIndex with an *unstructured component* – see XMLIndex Unstructured Component.

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 C-1 explores the contents of the path table for two purchase-order documents.

Example C-1 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>
<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

- Guidelines for Using XMLIndex with an Unstructured Component There are several guidelines that can help you use XMLIndex with an unstructured component.
- 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.
- 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:

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.

- XPath Expressions That Are Not Indexed by an XMLIndex Unstructured Component A few types of XPath expressions are *not* indexed by XMLIndex.
- Using XMLIndex with an Unstructured Component
 You can perform various operations on an XMLIndex index that has an unstructured component, including manipulating the path table and the secondary indexes of that component.
- Asynchronous (Deferred) Maintenance of XMLIndex Indexes

You can defer the cost of maintaining an XMLIndex index that has *only* an *unstructured* component, performing maintenance only at commit time or when database load is reduced. This can improve DML performance, and it can enable bulk loading of unsynchronized index rows when an index is synchronized.

Advantages of Unstructured XMLIndex B-tree indexes can be used advantageously with object-relational XMLType 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. That is the special domain of XMLIndex.

- XMLIndex Path Subsetting: Specifying the Paths You Want to Index If you know which XPath expressions you are most likely to query then you can narrow the focus of XMLIndex indexing and thus improve performance.
- PARAMETERS Clause for CREATE INDEX and ALTER INDEX in Unstructured Index

Guidelines for Using XMLIndex with an Unstructured Component

There are several guidelines that can help you use 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:

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_binxml, XMLTable('//LineItem'
PASSING OBJECT VALUE);
```

Do not use this:

• 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_binxml 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_binxml 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_binxml 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 C-8.

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.

Related Topics

 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.

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.

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.

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 for examples of creating secondary indexes on column VALUE
- PARAMETERS Clause for CREATE INDEX and ALTER INDEX for the syntax of the PARAMETERS clause

XPath Expressions That Are Not Indexed by an XMLIndex Unstructured Component

A few types of XPath expressions are not indexed by XMLIndex.

- Applications of XPath functions. 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).

Using XMLIndex with an Unstructured Component

You can perform various operations on an XMLIndex index that has an unstructured component, including manipulating the path table and the secondary indexes of that component.

To include an unstructured component in an XMLIndex index, you can use a path_table_clause in the PARAMETERS clause when you create or modify the XMLIndex index — see path_table_clause ::=.

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 C-2 shows how to name the path table ("my_path_table") when creating an XMLIndex index with an unstructured component.

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 C-3 shows this for the XMLIndex index created in Example 6-6.

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 C-4. The PARAMETERS clause of CREATE INDEX (and ALTER INDEX) must be between single quotation marks (').

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 C-5 illustrates this. (This assumes that you also have a structured component — Example 6-11 results in an index with both structured and unstructured components.)

In addition to specifying storage options for the path table, Example C-4 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 C-6 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.

See Also:

Example C-14 for a similar, but more complex example

Example C-2 Naming the Path Table of an XMLIndex Index

```
CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex PARAMETERS ('PATH TABLE my path_table');
```

Example C-3 Determining the System-Generated Name of an XMLIndex Path Table

```
SELECT PATH_TABLE_NAME FROM USER_XML_INDEXES
WHERE TABLE_NAME = 'PO_BINXML' AND INDEX_NAME = 'PO_XMLINDEX_IX';
```

PATH_TABLE_NAME

SYS67567 PO XMLINDE PATH TABLE

1 row selected.

Example C-4 Specifying Storage Options When Creating an XMLIndex Index

```
CREATE INDEX po_xmlindex_ix ON po_binxml (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)');
```

Example C-5 Dropping an XMLIndex Unstructured Component

ALTER INDEX po xmlindex ix PARAMETERS('DROP PATH TABLE');

Example C-6 Determining the Names of the Secondary Indexes of an XMLIndex Index

INDEX_NAME	COLUMN_NAME COLUMN_POSIT:	ION
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.

 Creating Additional Secondary Indexes on an XMLIndex Path Table You can add extra secondary indexes to an XMLIndex unstructured component.

Related Topics

 PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX The syntax for the PARAMETERS clause for CREATE INDEX and ALTER INDEX is defined.

Creating Additional Secondary Indexes on an XMLIndex Path Table

You can add extra secondary indexes to an XMLIndex unstructured component.

Examples Example C-9, Example C-11, Example C-12, and Example C-13 add extra secondary indexes to the XMLIndex index created in Example C-4.

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 C-9 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.

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 C-9, raises an ORA-01722 error (invalid number) if some of the element values are *not* numbers. This is illustrated in Example C-10.

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 C-11.

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 C-12 shows this .

Example C-13 creates an Oracle Text CONTEXT index on column VALUE. This is useful for fulltext queries on text values of XML elements. 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.

The query in Example C-14 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.

To know whether a particular XMLIndex index has been used in resolving a query, you can examine an execution plan for the query.

Similar to XMLIndex with Structured Component, 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.

You can examine the execution plan for a query to see whether a particular XMLIndex index has been used in resolving 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 C-8.

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 C-7

The unstructured component of an 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, which are deprecated for XMLType, XMLIndex indexes can be used when you extract data from an XML fragment in a document.



See Also:

- Column VALUE of an XMLIndex Path Table for information about the possibility of an Oracle Text CONTEXT index created on the VALUE column returning incorrect results
- Oracle Text Reference for information about CREATE INDEX parameter TRANSACTIONAL
- Oracle Database PL/SQL Packages and Types Reference for information on PL/SQL procedures createNumberIndex and createDateIndex in package DBMS XMLINDEX

Example C-7 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.

SET AUTOTRACE ON EXPLAIN

Example C-8 Extracting Data from an XML Fragment Using XMLIndex

```
SELECT li.description, li.itemno
 FROM po binxml, 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);
                              ITEMNO
DESCRIPTION
_____
A Night to Remember
                                   1
The Unbearable Lightness Of Being
                                   2
Sisters
                                   З
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
                              I
               1
|* 2 | TABLE ACCESS BY INDEX ROWID | MY_PATH_TABLE
                                           | 1 | 3524
```



```
ORACLE
```

```
C-14
```

```
* 3 | INDEX RANGE SCAN
                                  | SYS67616 PO XMLINDE PIKEY IX |
                                                                  1 |
  2 (0) | 00:00:01 |
|* 4 | FILTER
                                   1
                 | MY PATH TABLE
|*
   5 |
      TABLE ACCESS BY INDEX ROWID
                                                             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
                                   L
                                                                   8 | NESTED LOOPS
                                                             1 | 1546
1
   30 (4) |00:00:01 |
  9 | NESTED LOOPS
                                   1 | 24
  28 (4) 00:00:01
1
| 10 | VIEW
                                   | VW SQ 1
                                                             1 | 12
26 (0) 00:00:01
                                                                  1 | 5046
| 11 |
         HASH UNIQUE
                                                             1
| 12 | NESTED LOOPS
                                                             1
                                                                  1 | 5046
| 26 (0)|00:00:01 |
|* 13 | TABLE ACCESS BY INDEX ROWID | MY PATH TABLE
                                                             1
                                                                  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
                                                            1
2 (0) 00:00:01
|* 16 |
       INDEX RANGE SCAN | SYS67616 PO XMLINDE PIKEY IX |
                                                                  1 |
  1 (0) | 00:00:01 |
TABLE ACCESS BY USER ROWID | PO_BINXML
| 17 |
                                                            1 | 12
1 (0) | 00:00:01 |
                         | SYS67616_PO_XMLINDE_PIKEY_IX | 1 |
|* 18 | INDEX RANGE SCAN
| 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))</pre>
  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))</pre>
  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")
```

3 (0) | 00:00:01 |

```
+1=SYS_ORDERKEY_DEPTH("SYS_P10"."ORDER_KEY"))
   18 - access("PO_BINXML".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)
```

Example C-9 Creating a Function-Based Index on Path-Table Column VALUE

CREATE INDEX fn based ix ON po_path_table (substr(VALUE, 1, 100));

Example C-10 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

Example C-11 Creating a Numeric Index on Column VALUE with Procedure createNumberIndex

CALL DBMS XMLINDEX.createNumberIndex('OE', 'PO XMLINDEX IX', 'API NUM IX');

Example C-12 Creating a Date Index on Column VALUE with Procedure createDateIndex

```
CALL DBMS_XMLINDEX.createDateIndex('OE', 'PO_XMLINDEX_IX', 'API_DATE_IX', 'dateTime');
```

Example C-13 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');
```

Example C-14 Showing All Secondary Indexes on an XMLIndex Path Table

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.



Related Topics

Indexing XML Data for Full-Text Queries (pre-23ai)

When you need full-text search over XML data, Oracle recommends that you store your XMLType data as binary XML and you use XQuery Full Text (XQFT). You use an XML search index for this. This is the topic of this section.

Asynchronous (Deferred) Maintenance of XMLIndex Indexes

You can defer the cost of maintaining an XMLIndex index that has *only* an *unstructured* component, performing maintenance only at commit time or when database load is reduced. This can improve DML performance, and it can enable bulk loading of unsynchronized index rows when an index is synchronized.

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 (even if it also has an unstructured 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 C-3 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.

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)

Table C-3 Index Synchronization



When to Synchronize	ASYNC Clause Syntax
Periodically	ASYNC (SYNC EVERY "repeat_interval")
	<i>repeat_interval</i> 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.

Table C-3	(Cont.) Index Synchronization
-----------	-------------------------------

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 C-15 creates an XMLIndex index that is synchronized every Monday at 3:00 pm, starting tomorrow.

Example C-16 manually synchronizes the index created in Example C-15.

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 C-16, 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

Example C-15 Specifying Deferred Synchronization for XMLIndex

CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex PARAMETERS ('ASYNC (SYNC EVERY "FREQ=HOURLY; INTERVAL = 1")');

Example C-16 Manually Synchronizing an XMLIndex Index Using SYNCINDEX

EXEC DBMS_XMLINDEX.syncIndex('OE', 'PO_XMLINDEX_IX', REINDEX => TRUE);



Syncing an XMLIndex Index in Case of Error ORA-08181

If a query raises error ORA-08181, check whether the base XMLType table of the query has an XMLIndex index with an unstructured component. If so, then manually synchronize the XMLIndex index using DBMS XMLINDEX.syncIndex.

Syncing an XMLIndex Index in Case of Error ORA-08181

If a query raises error ORA-08181, check whether the base XMLType table of the query has an XMLIndex index with an unstructured component. If so, then manually synchronize the XMLIndex index using DBMS XMLINDEX.syncIndex.

This applies only if error ORA-08181 is raised in the following situation:

- 1. In a pluggable database, *PDB1*, you created an XMLType table or column *XTABCOL*, which you indexed using an XMLIndex index that has an unstructured component.
- 2. You plugged *PDB1* into a container database.
- **3.** You cloned *PDB1* to a new pluggable database, *PDB2*.
- 4. Error ORA-08181 is raised when you query XTABCOL in PDB2.

If the error is raised even after synchronizing then seek another cause. Error ORA-08181 is a general error that can be raised in various situations, of which this is only one.

Related Topics

Oracle XML DB and Database Consolidation
 Each pluggable database has its own Oracle XML DB Repository, and its own Oracle XML DB configuration file, xdbconfig.xml.

Advantages of Unstructured XMLIndex

B-tree indexes can be used advantageously with object-relational XMLType 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. That is the special domain 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 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, which are deprecated, cannot be used to extract document fragments.
- 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.

Data Dictionary Static Public Views Related to Unstructured XMLIndex

Data Dictionary views reporting information about Unstructured XMLIndex indexes are shared with Structured XMLIndex indexes through public views <code>USER_XML_INDEXES</code>, <code>ALL_XML_INDEXES</code>, and <code>DBA_XML_INDEXES</code>.



Similar to Structured XMLIndex, statistics information is shared across multiple views.

When querying USER_TAB_STATISTICS, ALL_TAB_STATISTICS, DBA_TAB_STATISTICS, Statistics over the Path-Table can be queried by filtering over the TABLE_NAME column using the Path-Table name.

See Also:

Data Dictionary Static Public Views Related to XMLIndex

XMLIndex Path Subsetting: Specifying the Paths You Want to Index

If you know which XPath expressions you are most likely to query then you can narrow the focus of XMLIndex indexing and thus improve performance.

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 component 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 ::=.



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.

- Examples of XMLIndex Path Subsetting Some examples are presented of defining XMLIndex indexes on subsets of XPath expressions.
- XMLIndex Path-Subsetting Rules Rules that apply to XMLIndex path subsetting are described.

Related Topics

• PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX The syntax for the PARAMETERS clause for CREATE INDEX and ALTER INDEX is defined.

Examples of XMLIndex Path Subsetting

Some examples are presented of defining XMLIndex indexes on subsets of XPath expressions.

Example C-17 XMLIndex Path Subsetting with CREATE INDEX

```
CREATE INDEX po_xmlindex_ix ON po_binxml (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 C-18 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 C-19 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_binxml (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

Rules that apply to XMLIndex path subsetting are described.

- 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.

PARAMETERS Clause for CREATE INDEX and ALTER INDEX in Unstructured Index

- Usage of PATHS Clause Certain considerations apply to using the PATHS clause.
- Usage of create_index_paths_clause and alter_index_paths_clause Certain considerations apply to using create_index_paths_clause and alter_index_paths_clause.
- 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
 Certain considerations apply to using value_clause.
- Usage of async_clause Certain considerations apply to using the ASYNC clause.

Usage of PATHS Clause

Certain 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

Certain 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 <code>pikey_clause</code>, <code>path_id_clause</code>, and <code>order_key_clause</code> is optional. A pikey index is created even if you do not specify a <code>pikey_clause</code>. To create a path id index or an order-key index, you must specify a <code>path_id_clause</code> or an order_key_clause, respectively.

Usage of value_clause

Certain 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

Certain 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 quotation marks ("). 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 false.
 - For any other ASYNC option besides ALWAYS, STALE is TRUE.

D

Indexing XML Data for Full-Text Queries (pre-23ai)

When you need full-text search over XML data, Oracle recommends that you store your XMLType data as binary XML and you use XQuery Full Text (XQFT). You use an XML search index for this. This is the topic of this section.

XML search indexes can be created using the SEARCH INDEX FOR XML syntax or by creating an XQFT enabled CONTEXT index. Oracle recommends that you create XML search indexes by using the SEARCH INDEX FOR XML.

If portability and standardized code are not a concern, or if your XMLType data is stored objectrelationally, then you can alternatively use the Oracle-specific full-text constructs and syntax provided by Oracle Text, specifically Oracle SQL function contains.

You can perform XQuery Full Text (XQFT) queries on XMLType data that is stored as binary XML. If you use an XQFT full-text predicate in an XMLExists expression within a SQL WHERE clause, then you must create an XML search index. This section describes the creation and use of such an index.

- Creating and Using an XML Search Index An XQuery Full Text query can use an XML search index to improve performance.
- What To Do If an XML Search Index Is Not Picked Up You can modify your query to ensure that certain conditions are satisfied, so its evaluation picks up an XML search index.
- Pragma ora:no_schema: Using XML Schema-Based Data with XQuery Full Text Oracle recommends in general that you use *non* XML Schema-based XMLType data when you use XQuery Full Text and an XML search index. But you can in some circumstances use XML Schema-based XMLType data that is stored as binary XML. Oracle XQuery pragma ora:no schema can be useful in this context.
- Pragma ora:use_xmltext_idx: Forcing the Use of an XML Search Index You can use XQuery pragma ora:use_xmltext_idx to force the use of an XML search index.
- Migrating from Using Oracle Text Index to XML Search Index

If you have legacy queries for XMLType data stored as binary XML that use SQL function CONTAINS and an Oracle Text index that is not XML-enabled, then consider using XQuery Full Text constructs instead.

Related Topics

Support for XQuery Full Text

Oracle XML DB supports XQuery Full Text for XMLType data that is stored as binary XML. Oracle Text technology provides the full-text indexing and search that is the basis of this support.



See Also: Example D-6

Creating and Using an XML Search Index

An XQuery Full Text query can use an XML search index to improve performance.

To create an XML search index you must be granted database role CTXAPP. More generally, this role is needed to create Oracle Text indexes, to set Oracle Text index preferences, or to use Oracle Text PL/SQL packages.

Before creating the index, you must create an Oracle Text path section group and set its XML ENABLE attribute to t. This makes the path section group XML-aware.

For best performance, create an index preference of type BASIC_STORAGE in the Oracle Text data dictionary, specifying the following attributes:

- D_TABLE_CLAUSE Specify SECUREFILE storage for column DOC of index data table \$D, which contains information about the structure of your XML documents. Specify caching and medium compression.
- I_TABLE_CLAUSE Specify SECUREFILE storage for column TOKEN_INFO of index data table \$1, which contains information about full-text tokens and their occurrences in the indexed documents. Specify *caching* (but not compression).

This is illustrated in Example D-1, which uses a non XML-schema-based XMLType table, po_binxml (which has the same data as table purchaseorder in standard database schema OE).

Index preference BASIC_STORAGE specifies the tablespace and creation parameters for the database tables and indexes that constitute an Oracle Text index.

See Also:

- Oracle Text Reference for information about section groups
- Oracle Text Reference for information about procedure CTX_DDL.set_sec_grp_attr
- Oracle Text Reference for information about procedure CTX_DDL.create_preference
- Oracle Text Reference for information about procedure CTX_DDL.set_attribute
- Oracle Text Reference for information about preference BASIC_STORAGE, D_TABLE_CLAUSE, and I_TABLE_CLAUSE

Example D-2 queries the data to retrieve the Description elements whose text contains both Big and Street, in that order.

Example D-3 shows the execution plan for the query, which indicates that index po_ctx_idx is picked up.



Example D-1 Creating an XML Search Index

```
BEGIN
  CTX DDL.create section group('mysecgroup', 'PATH SECTION GROUP');
  CTX DDL.set sec grp attr('mysecgroup', 'XML ENABLE', 'T');
  CTX DDL.create preference ('mypref', 'BASIC STORAGE');
 CTX_DDL.set_attribute('mypref',
                         'D TABLE CLAUSE',
                         'TABLESPACE my_ts
                         LOB (DOC) STORE AS SECUREFILE
                          (TABLESPACE my ts COMPRESS MEDIUM CACHE)');
  CTX DDL.set attribute('mypref',
                         'I TABLE CLAUSE',
                         'TABLESPACE my ts
                         LOB (TOKEN INFO) STORE AS SECUREFILE
                          (TABLESPACE my ts NOCOMPRESS CACHE) ');
END;
/
CREATE INDEX po_ctx_idx ON po_binxml(OBJECT_VALUE)
  INDEXTYPE IS CTXSYS.CONTEXT
```

PARAMETERS('storage mypref section group mysecgroup');

Example D-2 XQuery Full Text Query

Example D-3 Execution Plan for XQuery Full Text Query

Id Operation	Name		Rows		Bytes		Cost	(%CPU)	Time	
0 SELECT STATEMENT 1 TABLE ACCESS BY * 2 DOMAIN INDEX	 INDEX ROWID PO_BINXML PO CTX IDX	Ì		İ	2014 2014	İ	4	L (0)	00:00:01 00:00:01 00:00:01	

Predicate Information (identified by operation id):

2 - access("CTXSYS"."CONTAINS"(SYS_MAKEXML(0,"XMLDATA"),'<query><textquery grammar="CONTEXT" lang="english"> (({Big}) and ({Street})) INPATH (/PurchaseOrder/LineItems/LineItem/Description)</textquery></query>')>0)

Note

- dynamic sampling used for this statement (level=2)
- Unoptimized XML construct detected (enable XMLOptimizationCheck for more information)

21 rows selected.



What To Do If an XML Search Index Is Not Picked Up

You can modify your query to ensure that certain conditions are satisfied, so its evaluation picks up an XML search index.

If you use an XQuery full-text predicate in an XMLExists expression within a SQL WHERE clause, but you do not create an XML search index or the index cannot be used for some reason, then compile-time error ORA-18177 is raised.

If this error is raised then your execution plan does *not* indicate that the index is picked up. In the plan you do not see operation DOMAIN INDEX followed by the name of the index.

In that case, try to change your query to enable the index to be used. The following conditions must both apply for the index to be picked up:

- The expression that computes the XML nodes for the search context must be an XPath expression whose steps are only along *forward* and *descendent axes*.
- You can pass only one XMLType instance as a SQL expression in the PASSING clause of SQL/XML function XMLExists, and each of the other, non-XMLType SQL expressions in that clause must be either a *compile-time constant* of a SQL built-in data type or a *bind variable* that is bound to an instance of such a data type.

Pragma ora:no_schema: Using XML Schema-Based Data with XQuery Full Text

Oracle recommends in general that you use *non* XML Schema-based XMLType data when you use XQuery Full Text and an XML search index. But you can in some circumstances use XML Schema-based XMLType data that is stored as binary XML. Oracle XQuery pragma ora:no schema can be useful in this context.

By default, when an XML search index is used to evaluate XML Schema-based data, compiletime error ORA-18177 is raised. This is because the full-text indexing itself makes no use of the associated XML schema: it is not type-aware. It treats all of the text that it applies to as untyped. This error is raised even if you type-cast data appropriately and thus do not depend on the XML schema to cast types implicitly. Example D-4 illustrates this.

The error raised draws this to your attention, in case you might be expecting a full-text condition in your query to depend on XML Schema types and typed operations.

In order to use a condition that depends on types you must explicitly cast the relevant XQuery expressions to the appropriate types. Do not expect Oracle XML DB to use the XML schema to perform implicit type casting. Failure to type-cast appropriately can lead to results that you might not expect.

Example D-5 shows a query of XML Schema-based data that uses explicit type-casting to ensure that the proper condition is evaluated.

However, most uses of XQuery Full Text expressions, even with XML Schema-based data, do not involve data that is typed. Just remember that if you do use a condition that makes use of typed data then you must cast to the proper type.

In sum, if you are sure that your query does not involve typed data, or if you judge that it is all right to treat particular typed data as if it were untyped, or if you explicitly type-cast any data that needs to be typed, then you can use Oracle XQuery pragma ora:no_schema in your query to inhibit raising the error and allow evaluation of the query using an XML search index.



Example D-4 XQuery Full Text Query with XML Schema-Based Data: Error ORA-18177

```
SELECT XMLQuery('/PurchaseOrder/LineItems/LineItems'
PASSING OBJECT_VALUE RETURNING CONTENT)
FROM oe.purchaseorder
[LineItems/LineItem/@ItemNumber > xs:integer("20")
and Actions/Action/User contains text "KPARTNER"]'
PASSING OBJECT_VALUE);
FROM oe.purchaseorder
*
ERROR at line 3:
ORA-18177: XQuery full text expression '/PurchaseOrder
[LineItems/LineItem/@ItemNumber > xs:integer("20")
and Actions/Action/User contains text "KPARTNER"]'
cannot be evaluated using XML text index
```

Example D-5 Using XQuery Pragma ora:no_schema with XML Schema-Based Data

Pragma ora:use_xmltext_idx: Forcing the Use of an XML Search Index

You can use XQuery pragma ora:use xmltext idx to force the use of an XML search index.

A given query involving XML data can be evaluated in various ways, depending on the existence of different indexes and other factors. Sometimes the default evaluation method is not the most performant and it would be more efficient to force the use of an existing XML search index. You can use XQuery pragma ora:use_xmltext_idx to do this. (An XML search index applies only to XMLType data stored as binary XML.)

For example, a WHERE clause might include two XMLExists expressions, only one of which involves an XQuery full-text condition, and you might have an XMLIndex index that applies to the XMLExists expression that has no full-text condition. With such a query it is typically more efficient to use an XML search index to evaluate the entire WHERE clause.

Even in some cases where there is no full-text condition in the query, the use of an XML search index can provide the most efficient query evaluation.

The query in Example D-6 illustrates the use of pragma ora:use_xmltext_idx. Only the first of the XMLExists clauses uses a full-text condition. Because of the pragma, the full-text index (po ctx idx, created in Example D-1) is used for both XMLExists clauses.

Example D-6 Full-Text Query with XQuery Pragma ora:use_xmltext_idx

```
SELECT XMLQuery('/PurchaseOrder/LineItems/LineItem'

PASSING OBJECT_VALUE RETURNING CONTENT)

FROM po_binxml

WHERE XMLExists('/PurchaseOrder/LineItems/LineItem

[Description contains text "Picnic"]' PASSING OBJECT VALUE)
```



AND XMLExists('(# ora:use_xmltext_idx #) {/PurchaseOrder[User="SBELL"]}' PASSING OBJECT_VALUE);

Migrating from Using Oracle Text Index to XML Search Index

If you have legacy queries for XMLType data stored as binary XML that use SQL function CONTAINS and an Oracle Text index that is not XML-enabled, then consider using XQuery Full Text constructs instead.

The XQuery and XPath Full Text (XQFT) standard is supported by Oracle XML DB starting with Oracle Database 12c Release 1 (12.1). This support applies only to XMLType data stored as binary XML. Prior to that release, for full-text querying of XML data you could use only an Oracle Text index that was not XML-enabled (not an XML search index), and your full-text queries necessarily used Oracle-specific constructs: SQL function CONTAINS.

If you have legacy code that does this, Oracle recommends that you migrate that code to use XQFT. This section provides information about which XQFT constructs you can use to replace the use of <code>CONTAINS</code>.

This use of an Oracle Text index can also be replaced by the use of an XML search index. To replace a query that uses HASPATH by one that uses a simple XQuery expression, you use Oracle XQuery pragma ora:use_xmltext_idx to specify that the XML search index is to be picked up. This section also illustrates this.

Table D-1 provides a mapping from typical queries that use Oracle-specific constructs to queries that use XQuery Full Text.

Table D-1 Migrating Oracle-Specific XML Queries to XQuery Full Text

Original Example	Replacement Example
CONTAINS(t.x, 'HASPATH (/P/LIS/LI/ Description ¹)') > 0	<pre>XMLExists('(# ora:use_xmltext_idx #) {\$d/P/LIs/LI/Description¹}' PASSING t.x AS "d")</pre>
	Or if the data is XML Schema-based:
	<pre>XMLExists('(# ora:use_xmltext_idx #) {(# ora:no_schema #) {\$d/P/LIs/LI/Description¹}}' PASSING t.x AS "d")</pre>

Original Example		Replacement Example
CONTAINS(t.x,	<pre>'Big INPATH (/P/LIs/LI/Description)') > 0</pre>	XMLExists('\$d/P/LIs/LI/Description [. contains text "Big"]' PASSING t.x AS "d")
		Or if the data is XML Schema-based:
		<pre>XMLExists('(# ora:no_schema #) {\$d/P/LIs/LI/Description [. contains text "Big"]}' PASSING t.x AS "d")</pre>
CONTAINS(t.x,	<pre>'(Big) AND (Street) INPATH (/P/LIs/LI/Description)') > 0</pre>	<pre>XMLExists('\$d/P/LIs/LI/Description [. contains text "Big" ftand "Street"]' PASSING t.x AS "d")</pre>
CONTAINS(t.x,	<pre>'(Big) OR (Street) INPATH (/P/LIs/LI/Description)') > 0</pre>	<pre>XMLExists('\$d/P/LIs/LI/Description [. contains text "Big" ftor "Street"]' PASSING t.x AS "d")</pre>
CONTAINS(t.x,	<pre>'({Big}) NOT ({Street}) INPATH (/P/LIs/LI/Description)') > 0</pre>	<pre>XMLExists('\$d/P/LIs/LI/Description [. contains text "Big" ftand ftnot "Street"]' PASSING t.x AS "d")</pre>
CONTAINS(t.x, Street}) INPAT	'({Street}) MNOT ({Big H (/P/LIs/LI/Description)') > 0	<pre>XMLExists('\$d/P/LIs/LI/Description [. contains text "Street" not in "Big Street"]' PASSING t.x AS "d")</pre>
CONTAINS(t.x, INPATH	<pre>'(NEAR (({Big}, {Street}), 3) (/P/LIs/LI/Description)') > 0</pre>	XMLExists('\$d/P/LIs/LI/Description [. contains text "Big" ftand "Street" window 3
		words]' PASSING t.x AS "d")

Table D-1 (Cont.) Migrating Oracle-Specific XML Queries to XQuery Full Text

Original Example	Replacement Example
(Not applicable – Oracle Text queries are not XML namespace aware.)	XMLExists('declare namespace ipo="http://www.example.com/IPO",
	/ipo:P/ipo:LIs/ipo:LI/
	ipo: Description
	[. contains text "Big"]'
	PASSING t.x AS "d")

Table D-1 (Cont.) Migrating Oracle-Specific XML Queries to XQuery Full Text

¹ The path test can contain a predicate expression, which is the same for both the original query (with HASPATH) and its replacement. For example: /PurchaseOrder/LineItems/LineItem/Part[@Id < "31415927"].



Index

A

access control entry (ACE) definition, 27-5 access control list (ACL) definition, 27-6 overview, 1-17 system, 27-10 account XDB, 2-1 ACE See access control entry ACL See access control list administering Oracle XML DB, 34-1 Advanced Queuing (AQ) IDAP, 37-4 messaging scenarios, 37-2 point-to-point support, 37-1 publish/subscribe support, 37-1 XMLType queue payloads, 37-6 aggregate privilege definition, 27-4 annotations XML schema, 17-6, 18-7 querying to obtain, 18-20 atomic privilege definition, 27-4 attribute XML Schema data types mapping to SQL, 18-23 attributes, 1-13 xdb:columnProps, 18-14 xdb:defaultTable. 18-14 xdb:maintainDOM, 18-14 xdb:SQLCollType, 18-14 xdb:SQLInline, 18-14 xdb:SQLName, 18-14 xdb:SQLType, 18-14 xdb:tableProps, 18-14 auditing of HTTP and FTP protocols, 28-13 authentication definition, 27-3 authorization definition. 27-3

В

backward-compatible XML schema evolution definition, 20-17 binary XML definition, 1-12 bootstrap ACL definition, 27-10

С

C API for XML. 14-1 Cascading Style Sheets (CSS), 11-7 chaining XMLTable calls, 4-16 circular dependencies XML schemas, 18-50 collection in out-of-line tables. 18-47 loading and retrieving large documents, 18-62 XMI definition, 17-25 collection attribute (SQL), 18-30 column pattern, XMLTable SQL function definition, 4-13 columnProps attribute, 18-14 complex XLink link, 23-3 definition, 23-3 See also extended XLink link complexType handling cycles, 18-51 handling inheritance, 18-31 mapping fragments to LOBs, 18-60 XML Schema data types to SQL, 18-30 Oracle XML DB restrictions and extensions, 18-31 component of a resource path name definition, 21-7 compound XML document, 23-4 definition, 23-2 compression for online transaction processing (OLTP) using CREATE TABLE, 17-25 XMLType support, B-1 configuring Oracle XML DB protocol server, 28-4

configuring Oracle XML DB (continued) repository, 22-1 servlets, 31-3 using DBMS_XDB API, 34-12 using Oracle Enterprise Manager, 34-5 xdbconfig.xml configuration file, 34-5 constraints on XMLType data, 3-8, 18-40 repetitive XML elements, 18-42 content of a resource definition, 21-7 content-management application definition, 23-2 copy-based XML schema evolution, 20-2 copy-namespace mode, XQuery, 4-29 copyEvolve PL/SQL procedure, 20-1 COST XML QUERY REWRITE optimizer hint, 5-43 cost-based XML guery rewrite definition, 5-43 **CREATE TABLE statement** encoding options for binary XML, 17-22 storage options, 17-22 creating an XMLType table for nested collections, 18-35 storage options, 17-22 XML schema-based, 17-19, 18-35 CTXAPP role, D-2 cyclical dependencies XML schemas, 18-50

D

database role definition, 27-3 database user definition, 27-3 date and time XML Schema data types mapping to SQL, 18-25 DBMS XDB CONFIG PL/SQL package, 26-7 DBMS XDB REPOS PL/SQL package, 26-1 DBMS XMLDOM PL/SQL package, 11-5 examples, **11-16** DBMS XMLGEN PL/SQL package, 8-23 DBMS XMLINDEX PL/SQL package modifyParameter procedure, 6-28 registerParameter procedure, 6-28 syncIndex procedure, C-16 DBMS XMLPARSER PL/SQL package, 11-26 DBMS XMLSCHEMA PL/SQL package copyEvolve procedure, 20-1 inPlaceEvolve procedure, 20-1 registerSchema procedure, 17-10 DBMS XMLSCHEMA ANNOTATE PL/SQL package, 18-13

DBMS XMLSTORAGE MANAGE PL/SQL package disableIndexesAndConstraints procedure, 35-4 enableIndexesAndConstraints procedure, 35-4 exchangePostProc procedure, 6-10 exchangePreProc procedure, 6-10 refPartitionExchangeIn procedure, 6-10 refPartitionExchangeOut procedure, 6-10 renameCollectionTable procedure, 19-6 XPath2TabColMapping procedure, 19-5, 19-7 DBMS XMLSTORE PL/SQL package, 12-1 DBMS XSLPROCESSOR PL/SQL package, 11-27 DBUri definition, 32-3 generating using sys DburiGen SQL function, 32-26 identifying a row, 32-20 identifying a target column, 32-20 retrieving column text value, 32-22 retrieving the whole table, 32-18 security, 32-33 servlet, installation, 32-32 DBUri-refs, 32-14 HTTP access, 32-30 **DBUriServlet** definition, 32-30 debugging XML schema registration, 18-64 default tables creating during XML schema registration, 18-5 defaultTable attribute, 18-14 deleting resource, 24-17 XML schema using DBMS XMLSCHEMA, 17-17 depth SOL function, 24-9 derived XML Schema data types mapping to SQL, 18-25 digest access authentication, 28-11 digest access authentication, configuring, 28-11 digest authentication, 28-11 digest authentication, configuring, 28-11 directory See folder document (DOM) definition, 11-13 document link definition. 21-13, 23-3 document location hint definition. 17-8 **Document Object Model** See DOM

Document Type Definition See DTD document-correlated recursive query definition, 18-56 DOM, 11-6 definition, 11-1 difference from SAX, 11-8 document definition, 11-13 fidelity, 17-7 for XML schema mapping, 11-12 Java API for XMLType, 13-2 NamedNodeMap object, 11-15 NodeList object, 11-15 overview, 11-6 PL/SQL API for XMLType, 11-5 DOM fidelity definition, 17-7 DTD. definition, 1-15 support in Oracle XML DB, 1-15 use with Oracle XML DB, 1-15 dynamic type-checking XQuery language, 4-26

Е

effective text value of a node definition. C-7 element XML Schema data types mapping to SQL, 18-24 Enterprise Manager administering Oracle XML DB, 34-5 entities, XML using a DTD with binary XML storage, 1-15 equipartitioning of XMLType tables definition, 18-37 error ORA-08181, C-18 ORA-18177, D-4 event repository, 30-1 configuring, 30-9 predefined, 30-4 event handler, repository definition, 30-3 event listener, repository definition, 30-3 evolution, XML schema, 20-1 extended XLink link, 23-3 definition, 23-3 extracting data from XML, 5-19

F

fidelity DOM, 17-7 for XML schema mapping, 11-12 FLWOR XQuery expression, 4-8 fn:available XQuery function support, 4-30 fn:collection XQuery function support, 4-29 fn:doc XQuery function support, 4-29 fn:id XQuery function support, 4-29 fn:idref XQuery function support, 4-29 fn:replace XQuery function, 4-22 folder. 21-7 definition, 21-7 folder link definition, 21-13 folder-restricted query definition, 21-38 foldering, 21-2 fragment, XML definition, 3-25 SQL operations on, 3-25 fragments, XML mapping to LOBs, 18-60 FROM list order XMLTable PASSING clause, 5-10 FTP configuration parameters, 28-4 creating default tables, 18-5 protocol server, features, 28-14 FTP protocol auditing, 28-13 full-text indexing. D-1 full-text search XML data, 4-5, 4-30 fully gualified XML schema URLs, 17-16 functional evaluation definition. 19-6 functions SQL depth, 24-9 sys DburiGen, 32-26 sys_XMLAgg, 8-48 under path, 24-7, 24-10 updating XML data, 5-28 XMLAgg, 8-13 XMLAttributes, 8-3 XMLCast, 4-19 XMLCDATA, 8-22 XMLColAttVal, 8-21 XMLComment, 8-17 XMLConcat, 8-12

functions (continued) SQL (continued) XMLElement, 8-3 XMLExists, 4-17 XMLForest, 8-11 XMLIsValid, 7-14 XMLParse, 8-20 XMLPI, 8-16 XMLQuery, 4-11, 4-12 XMLSerialize, 8-18 XMLTable, 4-11, 4-13 XMLtransform, 7-3

G

generating XML, 8-1 DBMS XMLGEN PL/SQL package, 8-23 SQL functions, 8-1 sys_XMLAgg SQL function, 8-48 XMLAgg SQL function, 8-13 XMLAttributes SQL function, 8-3 XMLCDATA SQL function, 8-22 XMLColAttVal SQL function, 8-21 XMLComment SQL function, 8-17 XMLConcat SQL function, 8-12 XMLElement SOL function, 8-3 XMLForest SQL function, 8-11 XMLParse SQL function, 8-20 XMLPI SQL function, 8-16 XMLSerialize SQL function, 8-18 getCLOB() XMLType method, 13-9 getSchemaURL() XMLType method, 17-8 global XML schema using fully qualified URL to override, 17-16 group in an XMLIndex structured component definition, 6-14

Η

hard link definition, 21-14 hierarchical repository index, 21-41 hierarchy-enabled table definition, 27-21 HTTP access for DBUri-refs, 32-30 accessing Java servlet or XMLType, 31-2 accessing repository resources, 21-19 configuration parameters, WebDAV, 28-4 creating default tables, 18-5 improved performance, 28-2 Oracle XML DB servlets, 31-7 protocol server, features, 28-23 requests, 31-7 servlets, 31-3 URIFACTORY, 32-34

HTTP (continued) using UriRefs to store pointers, 32-3 HTTP protocol auditing, 28-13 httpconfig element, xdbconfig.xml, 34-8 HTTPUri definition, 32-3 hybrid columnar compression, *B-1*

I

IDAP architecture, 37-5 transmitted over Internet, 37-4 in-place XML schema evolution, 20-16 index hierarchical repository, 21-41 indexing full-text, D-1 XMLType, 6-2 choosing, 16-1 inheritance XML schema, restrictions in complexTypes, 18-31 inPlaceEvolve PL/SQL procedure, 20-1 instance document definition, 1-13, 17-3 specifying root element namespace, 17-3 Internet Data Access Presentation (IDAP) SOAP specification for AQ, 37-4 Internet Protocol Version 6 FTP, 28-22 HTTP(S), 28-26 IPv6 FTP, 28-22 HTTP(S), 28-26 isSchemaBased() XMLType method, 17-8 isSchemaValidated() XMLType method, 17-8

J

Java DOM API for XMLType, *13-2* Oracle XML DB applications, *31-2* JDBC accessing XML documents, *13-3* drivers, OCI and thin, *13-5* loading large XML documents, *13-9* manipulating XML documents, *13-5*

L

large node handling, *11-18* lazy XML loading (lazy manifestation), *11-2* LDAP principal definition, *27-3* link

document definition, 21-13 folder definition, 21-13 hard definition, 21-14 repository definition, 21-13 weak definition, 21-14 link name definition, 21-7 link-properties document definition, 21-23 linking, repository definition, 21-23 loading large documents with collections, 18-62 loading large XML documents using JDBC, 13-9 loading of XML data, lazy, 11-2 LOB locator, 25-6 LOBs mapping XML fragments to, 18-60 local XML schema definition, 17-14 using fully qualified URL to specify, 17-16

Μ

maintainDOM attribute, 18-14 manifestation, lazy, 11-2 mapping complexType to SQL out-of-line storage, 18-44, 19-4 overriding using SQLType attribute, 18-25 simpleContent to object types, 18-34 mapping XML Schema complexType data types to SOL, 18-30 mapping XML Schema data types to SQL data types, 18-21 metadata definition, 29-1 system-defined definition, 1-17 user-defined definition, 1-17 methods **XMLType** getCLOB(), 13-9 getSchemaURL(), 17-8 isSchemaBased(), 17-8 isSchemaValid(), 17-8 isSchemaValidated(), 17-8 schemaValidate(), 17-8 setSchemaValidated(), 17-8

methods (continued) XMLType (continued) XML schema, 17-8 MIME overriding with DBUri servlet, 32-32 model, XML Schema definition, 20-19 modifyParameter PL/SQL procedure, 6-28

Ν

NamedNodeMap object (DOM), 11-15 namespace XQuery, 4-10, 5-17 naming SOL objects, 18-7 navigational access to repository resources, 21-16 nested XML generating using DBMS_XMLGEN, 8-31 generating with XMLElement, 8-7 NESTED TABLE ID pseudocolumn, 18-35 NodeList object (DOM), 11-15 nodes, large (DBMS XMLDOM), 11-18 non-schema-based view definition, 10-1 nonce definition, 28-12 nonce key definition, 28-12 numeric XML Schema data types mapping to SQL, 18-25

0

object attributes for collection (SQL), 18-30 sys DburiGen SOL function passing to, 32-27 XMLType, in AQ, 37-6 object identifier, 27-9 definition, 27-9 object-based persistence of XML data definition, 1-12 object-relational storage of XML data definition, 1-12 occurrence indicator definition, 4-6 OCI API for XML, 14-1 OCT definition, 18-3 ODP.NET, 15-1 OID See object identifier optimizer hints COST_XML_QUERY_REWRITE, 5-43 ORA-08181 error, C-18

ORA-18177 error, D-4 ora:defaultTable Oracle XQuery pragma, 5-50 ora:no schema Oracle XQuery pragma, D-4 ora:no xmlquery rewrite Oracle XQuery pragma, 6-22 ora:sqrt Oracle XQuery function, 4-23 ora:tokenize Oracle XQuery function, 4-23 ora:use xmltext idx Oracle XQuery pragma, D-5 Oracle ASM files accessing, 21-21 using FTP, 28-19 Oracle ASM virtual folder, 21-10 Oracle Data Provider for .NET, 15-1 Oracle Enterprise Manager administering Oracle XML DB, 34-5 Oracle Internet Directory, 27-23 Oracle Net Services, 1-9 **Oracle Text** searching for resources, 24-23 Oracle XML DB access models, 2-6 architecture, 1-9 features, 1-10 Java applications, 31-2 overview, 1-1 Repository See repository, 21-6 upgrading, 34-1 versioning, 25-1 when to use, 2-1 order index of XMLIndex definition, C-1 ordered collection definition, 18-3 ordered collection table (OCT) definition, 18-3 ordered collections in tables (OCTs) default storage of varray, 18-30 out-of-line storage, 18-44, 19-4 collections, 18-47 XPath rewrite, 19-4

Ρ

partial validation of XML data definition, 7-13 partitioning XMLType tables binary XML storage, 3-5 exchange partitioning, 6-10 object-relational storage, 18-37 examples, 18-38 with an XMLIndex index, 6-23 PASSING clause of XMLTable FROM list order, 5-10 path component of a resource path name definition, 21-7

path index of XMLIndex definition, C-1 path name definition, 21-7 resolution, 21-12 path table of XMLIndex, C-1 PATH VIEW, 24-1 path-based access to repository resources, 21-16 path-index trigger definition, 27-21 PD (positional descriptor), 18-6 persistence models for XML data, 1-12 PL/SOL functions See functions, PL/SQL PL/SQL packages DBMS_XDB_CONFIG, 26-7 DBMS XDB REPOS, 26-1 DBMS XMLDOM, 11-5 DBMS XMLGEN, 8-23 DBMS XMLPARSER, 11-26 DBMS XMLSCHEMA ANNOTATE, 18-13 DBMS_XMLSTORE, 12-1 DBMS XSLPROCESSOR, 11-27 for XMLType, 11-1 PL/SQL procedures See procedures, PL/SQL point-to-point support in AQ, 37-1 port, FTP, 28-4 port, HTTP, 28-4 ports configuring FTP, 28-4 HTTP, 28-4 HTTPS, 28-8 positional descriptor (PD), 18-6 post-parse persistence of XML data definition, 1-12 pragmas, XQuery See XQuery pragmas, Oracle predefined ACLs, 27-10 pretty-printing, 8-18 in book examples, *xlvi* not done by SQL/XML functions, 3-31 Web service output, 33-4 primitive XML Schema data types mapping to SQL, 18-25 principal definition, 27-3 LDAP definition, 27-3 private (local) XML schema, definition, 17-14 privilege definition, 27-4

procedures PL/SQL copyEvolve, 20-1 disableIndexesAndConstraints, 35-4 enableIndexesAndConstraints, 35-4 exchangePostProc, 6-10 exchangePreProc, 6-10 inPlaceEvolve, 20-1 modifyParameter, 6-28 refPartitionExchangeIn, 6-10 refPartitionExchangeOut, 6-10 registerParameter, 6-28 registerSchema, 17-10 renameCollectionTable, 19-6 syncIndex, C-16 XPath2TabColMapping, 19-5, 19-7 protocol auditing, HTTP and FTP, 28-13 protocol server, 28-1 architecture, 28-2 configuration parameters, 28-4 event-based logging, 28-13 FTP, 28-14 configuration parameters, 28-4 HTTP, 28-23 configuration parameters, 28-4 WebDAV configuration parameters, 28-4 protocolconfig element, xdbconfig.xml, 34-8 protocols, access to repository resources, 21-19 public (global) XML schema, definition, 17-14 publish/subscribe support in AQ, 37-1 purchase-order XML schema, A-34 annotated, A-34 revised, 20-3, A-34

Q

qualified XML schema URLs, 17-16 query-based access to resources using RESOURCE_VIEW and PATH_VIEW, 24-2 using SQL, 21-23 querying XMLType data choices, 5-19 transient data, 5-19

R

recursive schema support, *18-56* registered XML schemas, list of, *17-18* registering an XML schema debugging, *18-64* default tables, creating, *18-5* SQL object types, creating, *18-4* registerParameter PL/SQL procedure, *6-28*

registerSchema PL/SQL procedure, 17-10 renaming an XMLIndex index, 6-12 replace XQuery function, 4-22 repository, 21-6 data storage, 21-9 event, 30-1 configuring, 30-9 predefined, 30-4 event handler definition, 30-3 event listener definition, 30-3 hierarchical index, 21-41 use with XQuery, 5-3 repository link definition, 21-13 RESID definition, 25-4 resource access, 21-5 using protocols, 28-13 definition, 1-17, 29-1 deleting, 21-14 nonempty container, 24-17 using DELETE, 24-17 management using DBMS_XDB_REPOS, 26-1 managing with DBMS XDB, 34-16 required privileges for operations, 27-6 searching for, using Oracle Text, 24-23 setting property in ACLs, 27-12 simultaneous operations, 24-21 updating, 24-18 resource configuration file definition, 22-2 resource configuration list definition, 22-2 resource content definition, 21-7 resource document definition, 21-6 resource ID definition, 25-4 resource name definition, 21-7 resource version definition, 25-1 **RESOURCE VIEW** explained, 24-1 resource-view-cache-size configuration parameter, 24-22 retrieving large documents with collections, 18-62 **RETURNING SEQUENCE BY REF clause of** XMLTable, 5-10 revalidation mode, XQuery Update, 4-28

rewrite XPath (XPath), 19-1 XQuery, 5-42 role CTXAPP, D-2 database definition, 27-3 root configuration, 34-15 root folder, repository, 21-2 root XML Schema definition, 17-4 row pattern, XMLTable SQL function definition, 4-13 rule-based XML query rewrite definition, 5-43

S

schema evolution See XML schema evolution schema for schemas (XML Schema) definition, 1-13 schema location hint definition, 17-8 schemaValidate() XMLType method, 17-8 security DBUri, 32-33 servlet element, xdbconfig.xml, 34-9 servlets accessing repository data, 21-24 APIs, 31-8 configuring, 31-3 session pooling, 31-8 writing, 31-9 in Java, 31-3 XML manipulation, 31-3 session pooling, 31-8 protocol server, 28-2 simple XLink link definition. 23-3 simpleContent mapping to object types, 18-34 SOAP, 33-1 IDAP, 37-4 SOL functions See functions, SQL SQL object types creating during XML schema registration, 18-4 SQL*Loader, 35-2 SOL*Plus XQUERY command, 5-24 SQL/XML generation functions definition, 1-16, 8-2 SQL/XML publishing functions definition, 1-16, 8-2

SOL/XML guery and update functions definition, 1-16 SOL/XML standard generating XML data, 8-2 querying XML data XMLQuery and XMLTable, 4-11 SOLCollType attribute, 18-14 SQLInline attribute, 18-14 SOLJ, 13-12 SQLName attribute, 18-14 SQLType attribute, 18-14 sqrt XQuery function (Oracle), 4-23 static type-checking XQuery language, 4-26 storage out of line, 18-44, 19-4 collections, 18-47 storage models for XMLType, 1-12 choosing, 16-1 string XML Schema data types mapping to SQL, 18-25 mapping to VARCHAR2 vs CLOB, 18-29 structured storage of XMLType data definition, 1-12, 16-2 structured XMLIndex component definition, 6-5 style sheet, CSS, 11-7 stylesheet for updating XML instance documents, 20-11 syncIndex PL/SQL procedure, C-16 sys DburiGen SQL function, 32-26 inserting database references, 32-27 SYS NC ARRAY INDEX\$ column, 18-35 sys XMLAgg SOL function, 8-48 sysconfig element, xdbconfig.xml, 34-7 system ACL definition, 27-10 system ACLs, 27-10 system-defined metadata definition, 1-17

Т

tableProps attribute, *18-14* text value of a node, effective definition, *C-7* third-party XLink link definition, *23-3* time zone support, implicit, *4-28* trigger, path-index definition, *27-21* type-checking, static and dynamic XQuery language, *4-26*

U

UDT generating an element from, 8-7 under_path SQL function, 24-7 different correlations for different folders, 24-10 uniform access control mechanism definition, 27-21 unique constraint on parent element of an attribute, 18-42 unresolved XLink and XInclude links, 23-13 updating repository resource. 24-18 updating XML data using SQL functions, 5-28 upgrading Oracle XML DB, 34-1 **Uri-reference** database and session, 32-18 DBUri-ref. 32-14 HTTP access for DBUri-ref, 32-30 URIFACTORY PL/SQL package, 32-23 URIType examples, 32-8 **URIFACTORY PL/SQL package** configuring to handle DBURI-ref. 32-34 creating subtypes of URIType, 32-23 URIType examples, 32-8 user definition, 27-3 user XDB. 2-1 user-defined metadata definition, 1-17 userconfig element, xdbconfig.xml, 34-7

V

validating examples, 7-15 XMLIsValid SQL function use as CHECK constraint, 7-14 validation of XML data, partial definition, 7-13 value index of XMLIndex definition. C-1 varray in a LOB definition, 18-3 varray in a table definition, 18-3 VCR See version-controlled resource version resource definition, 25-1 version series of a resource definition, 25-4 version-controlled resource, 25-1 definition, 25-1

versionable resource definition, 25-1 versioning, 1-17, 25-1 views RESOURCE and PATH, 24-1

W

weak link definition, 21-14 deletion, 23-13 Web service, 33-1 pretty-printing output, 33-4 WebDAV definition, 21-2 WebFolder creating in Windows 2000, 28-34 well-formed XML document definition, 3-7 WSDL Web service for accessing stored PL/SQL, 33-6 Web service for database gueries, 33-4

Х

XDB database schema (user account), 2-1, 21-6 xdb namespace, 27-7 xdb:columnProps attribute, 18-14 xdb:defaultTable attribute, 18-14 xdb:maintainDOM attribute, 18-14 xdb:SQLCollType attribute, 18-14 xdb:SQLInline attribute, 18-14 xdb:SQLName attribute, 18-14 xdb:SOLType attribute, 18-14 xdb:tableProps attribute, 18-14 **XDBBinary element** definition, 21-7 xdbconfig element, xdbconfig.xml, 34-7 xdbconfig.xml configuration file, 34-5 xdbcore-loadableunit-size configuration parameter, 18-62, 18-64 xdbcore-xobmem-bound configuration parameter, 18-64 XDBSchema.xsd definition, 17-4 XDBUri, 32-4 definition, 32-3, 32-11 XInclude, 23-1 definition, 23-2 unresolved link, 23-13 XLink complex link definition, 23-3 definition, 23-2

XLink (continued) third-party link definition, 23-3 XLink (deprecated), 23-1 extended link definition, 23-3 link types, 23-3 simple link definition, 23-3 unresolved link, 23-13 XML attributes See attributes XML diagnosability mode, 5-47 XML entities using a DTD with binary XML storage, 1-15 XML fragment definition, 3-25 mapping to LOBs, 18-60 SOL operations on. 3-25 XML instance document definition, 1-13, 17-3 XML query rewrite definition, 5-42 cost-based. 5-43 rule-based. 5-43 XML schema annotations, 17-6, 18-7 querying to obtain, 18-20 circular dependencies, 18-50 complexType declarations, 18-31, 18-51 cyclical dependencies, 18-50 definition, 1-13, 17-2 deletion, 17-17 evolution, 20-1 backward-compatible, definition, 20-17 for XML schemas that can be registered, 17-4 inheritance in, complexType restrictions, 18-31 local and global, 17-14 mapping to SQL object types, 11-11 updating after registering, 20-1 URLs, 17-16 W3C Recommendation, 17-1, 17-2 XMLType methods, 17-8 XML Schema definition. xliv XML Schema data types mapping to SQL data types, 18-21 XML schema definition definition, 1-13 XML schema evolution, 20-1 copy-based, 20-2 in-place, 20-16 XML schema-based tables and columns, creating, 17-19

XML schema-based view definition. 10-1 XML search index definition, D-1 XML ENABLE path section group attribute, D-2 XMLAgg SOL function, 8-13 XMLAttributes SQL function, 8-3 XMLCast SQL function, 4-19 XMLCDATA SOL function, 8-22 XMLColAttVal SQL function, 8-21 XMLComment SQL function, 8-17 XMLConcat SOL function, 8-12 XMLElement SQL function, 8-3 XMLExists SQL function, 4-17 XMLForest SOL function, 8-11 XMLIndex, 6-5 creating index, 6-12 dropping index, 6-12 order index definition, C-1 partitioning and parallelism, 6-23 path index definition, C-1 path table, C-1 renaming index, 6-12 structured component definition, 6-5 synchronizing if ORA-08181, C-18 value index definition, C-1 XMLIsValid SOL function, 7-14 XMLNAMESPACES clause, 4-13 XMLParse SQL function, 8-20 XMLPI SOL function, 8-16 XMLOuery SOL function, 4-11, 4-12 XMLSerialize SQL function, 8-18 XMLTable SQL function, 4-11, 4-13 breaking up an XML fragment, 3-25 column pattern definition, 4-13 PASSING clause and FROM list order, 5-10 **RETURNING SEQUENCE BY REF clause,** 5-10 reverse node references in COLUMNS clause, 5-10 row pattern definition, 4-13 XMLtransform SQL function, 7-3 **XMLType** as abstract data type, 1-12 constructors, 3-12 DBMS XMLDOM PL/SQL API, 11-5 DBMS XMLPARSER PL/SQL API, 11-26 DBMS XSLPROCESSOR PL/SQL API, 11-27 extracting data, 5-19 indexing columns, 6-2

XMLType (continued) instances, PL/SQL APIs, 11-1 loading data, 35-1 loading with SQL*Loader, 35-2 methods getCLOB(), 13-9 getSchemaURL(), 17-8 isSchemaBased(), 17-8 isSchemaValid(), 17-8 setSchemaValidated(), 17-8 XML schema, 17-8 PL/SQL packages, 11-1 querving, 5-19 querying transient data, 5-19 querying XMLType columns, 5-19 queue payloads, 37-6 storage models, 1-12 tables, views, columns, 17-19 views, access with PL/SQL DOM APIs, 11-12 XPath language, 19-1 syntax, 4-2 See also XQuery language XPath rewrite, 19-1 definition. 5-42 indexes on singleton elements and attributes, 6-46 out-of-line storage, 19-4 XQuery copy-namespace mode, 4-29 extension expressions See XQuery pragmas, Oracle, 4-23 pending update list, 4-4 pragmas, Oracle ora:defaultTable, 5-50 ora:no schema, D-4 ora:no xmlguery rewrite, 6-22 ora:use xmltext idx, D-5 revalidation mode, 4-28 simple expression, 4-6 static typing feature, 4-29 time zone support, implicit, 4-28 XDM instance, 4-3 XQUERY command, SQL*Plus, 5-24 XQuery Data Model (XDM), 4-3 XQuery functions and operators support, 4-29

XQuery functions and operators (continued) XOuery language, 4-1, 19-1 expressions, 4-6 FLWOR, 4-8 rewrite, 5-42 functions fn:replace, 4-22 ora:sqrt (Oracle), 4-23 ora:tokenize (Oracle), 4-23 item definition, 4-4 namespaces, 4-10, 5-17 optimization, 5-42 optimization over relational data, 5-44 Oracle extension functions, 4-22 Oracle XML DB support, 4-27 performance, 5-42 predefined namespaces and prefixes, 4-10 referential transparency definition, 4-4 sequence definition, 4-4 SQL*Plus XQUERY command, 5-24 tuning, 5-42 type-checking, static and dynamic, 4-26 unordered mode definition, 4-4 update snapshot, 4-5 use with Oracle XML DB Repository, 5-3 use with XMLType relational data, 5-10 optimization, 5-45 XMLQuery and XMLTable SQL functions, 4-11 examples, 5-1 XQuery Update Facility, 4-1 XSD definition, 1-13 XSL stylesheet definition, 11-27 XSLT stylesheets for updating XML instance documents, 20-11 use with DBUri servlet, 7-9, 32-35 use with Oracle XML DB, 7-1 use with package DBMS XSLPROCESSOR, 11-29