

Oracle9i

XML Database Developer's Guide - Oracle XML DB

Release 2 (9.2)

March 2002

Part No. A96620-01

ORACLE®

Part No. A96620-01

Copyright © 2002 Oracle Corporation. All rights reserved.

Primary Author: Shelley Higgins

Graphics: Valarie Moore

Contributing Authors: Nipun Agarwal, Abhay Agrawal, Omar Alonso, Sandeepan Banerjee, Mark Bauer, Ravinder Booreddy, Yuen Chan, Sivasankaran Chandrasekar, Vincent Chao, Mark Drake, Fei Ge, Wenyun He, Sam Idicula, Neema Jalali, Bhushan Khaladkar, Viswanathan Krishnamurthy, Muralidhar Krishnaprasad, Wesley Lin, Annie Liu, Anand Manikutty, Jack Melnick, Nicolas Montoya, Steve Muench, Ravi Murthy, Eric Paapanen, Syam Pannala, John Russell, Eric Sedlar, Vipul Shah, Cathy Shea, Tarvinder Singh, Simon Slack, Muralidhar Subramanian, Priya Vennapusa, James Warner

Contributors: Harish Akali, Deanna Bradshaw, Paul Brandenstein, Lisa Eldridge, Susan Kotsovolos, Sonia Kumar, Roza Leyderman, Diana Lorentz, Yasuhiro Matsuda, Bhagat Nainani, Visar Nimani, Sunitha Patel, Denis Raphaely, Rebecca Reitmeyer, Ronen Wolf

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent and other intellectual and industrial property laws. Reverse engineering, disassembly or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Oracle9i, Oracle8i, Oracle8, SQL*Plus, On Oracle, Oracle Store, Oracle Press, ConText, and PL/SQL are trademarks or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments	xxvii
Preface.....	xxix
Audience	xxx
Organization.....	xxx
Related Documentation	xxxv
Conventions.....	xxxvi
Documentation Accessibility	xxxix
What's New In Oracle XML DB?	xli
Oracle XML DB: XMLType Enhancements	xli
Oracle XML DB: Repository.....	xlili
Oracle Tools Enhancements for Oracle XML DB.....	xliv
Oracle Text Enhancements	xliv
Oracle Advanced Queuing (AQ) Support.....	xliv
Oracle XDK Support for XMLType.....	xlvi
Part I Introducing Oracle XML DB	
1 Introducing Oracle XML DB	
Introducing Oracle XML DB	1-2
Not a Separate Database Server	1-2
Benefits of Oracle XML DB.....	1-3
Key Features of Oracle XML DB.....	1-4

Oracle XML DB and XML Schema	1-7
Oracle XML DB Architecture.....	1-8
XMLType Tables and Views Storage.....	1-10
Oracle XML DB Repository	1-11
XMLType Storage Architecture	1-12
Cached XML Object Management Architecture	1-15
XML Repository Architecture.....	1-16
Why Use Oracle XML DB?	1-17
Unifying Data and Content with Oracle XML DB.....	1-18
Oracle XML DB Offers Faster Storage and Retrieval of Complex XML Documents.....	1-21
Oracle XML DB Helps You Integrate Applications.....	1-22
When Your Data Is Not XML You Can Use XMLType Views.....	1-22
Searching XML Data Stored in CLOBs Using Oracle Text	1-24
Building Oracle XML DB XML Messaging Applications with Advanced Queueing.....	1-24
Managing Oracle XML DB Applications with Oracle Enterprise Manager.....	1-25
Requirements for Running Oracle XML DB	1-26
Standards Supported by Oracle XML DB.....	1-26
Oracle XML DB Technical Support	1-27
Terminology Used in This Manual.....	1-27
Oracle XML DB Examples Used in This Manual	1-30

2 Getting Started with Oracle XML DB

Getting Started with Oracle XML DB.....	2-2
Installing Oracle XML DB.....	2-2
When to Use the Oracle XML DB	2-2
Designing Your XML Application.....	2-3
Oracle XML DB Design Issues: Introduction.....	2-3
a. Data.....	2-3
b. Access	2-3
c. Application Language	2-4
d. Processing.....	2-4
Storage	2-4
Oracle XML DB Application Design: a. How Structured Is Your Data?.....	2-5
Oracle XML DB Application Design: b. Access Models	2-7
Oracle XML DB Application Design: c. Application Language	2-8

Oracle XML DB Application Design: d. Processing Models	2-9
Oracle XML DB Design: Storage Models	2-10
Using XMLType Tables	2-11
Using XMLType Views.....	2-12

3 Using Oracle XML DB

Storing Data in an XMLType Column or XMLType Table	3-3
Accessing Data in XMLType Columns or XMLType Tables	3-4
Using XPath with Oracle XML DB	3-5
Using existsNode()	3-6
Using extractValue().....	3-8
Using extract()	3-10
Using XMLSequence()	3-11
Updating XML Documents with updateXML()	3-12
Introducing the W3C XSLT Recommendation	3-14
Using XSL/XSLT with Oracle XML DB	3-16
Other XMLType Methods	3-17
Introducing the W3C XML Schema Recommendation	3-17
Using XML Schema with Oracle XML DB.....	3-19
XMLSchema-Instance Namespace	3-21
Validating an XML Document Using an XML Schema	3-22
Storing XML: Structured or Unstructured Storage	3-24
Data Manipulation Language (DML) Independence.....	3-27
DOM Fidelity in Structured and Unstructured Storage	3-27
Structured Storage: XML Schema-Based Storage of XMLType	3-28
Structured Storage: Storing complexType Collections	3-32
Structured Storage: Data Integrity and Constraint Checking.....	3-33
Oracle XML DB Repository	3-34
Query-Based Access to Oracle XML DB Repository	3-36
Using RESOURCE_VIEW	3-37
Using PATH_VIEW.....	3-37
Creating New Folders and Documents.....	3-37
Querying Resource Documents.....	3-38
Updating Resources	3-38
Deleting Resources	3-39

Storage Options for Resources	3-40
Defining Your Own Default Table Storage for XML Schema-Based Documents	3-40
Accessing XML Schema-Based Content	3-43
Accessing Non-Schema-Based Content With XDBUriType	3-44
Oracle XML DB Protocol Servers	3-44
Using FTP Protocol Server	3-44
Using HTTP/WebDAV Protocol Server	3-49

Part II Storing and Retrieving XML Data in Oracle XML DB

4 Using XMLType

What Is XMLType?	4-2
Benefits of the XMLType Data Type and API	4-3
When to Use XMLType	4-4
Storing XMLType Data in Oracle XML DB	4-4
Pros and Cons of XML Storage Options in Oracle XML DB	4-5
When to Use CLOB Storage for XMLType	4-6
XMLType Member Functions	4-7
How to Use the XMLType API	4-7
Creating, Adding, and Dropping XMLType Columns	4-8
Inserting Values into an XMLType Column	4-9
Using XMLType in an SQL Statement	4-9
Updating an XMLType Column	4-9
Deleting a Row Containing an XMLType Column	4-10
Guidelines for Using XMLType Tables and Columns	4-11
Specifying Storage Characteristics on XMLType Columns	4-12
Changing Storage Options on an XMLType Column Using XMLData	4-13
Specifying Constraints on XMLType Columns	4-14
Manipulating XML Data in XMLType Columns/Tables	4-14
Inserting XML Data into XMLType Columns/Tables	4-15
Using INSERT Statements	4-15
Selecting and Querying XML Data	4-17
Selecting XML Data	4-17
Querying XML Data	4-18
Using XPath Expressions for Searching XML Documents	4-18

Querying XML Data Using XMLType Member Functions	4-19
existsNode Function.....	4-20
extract () Function.....	4-21
extractValue() Function	4-23
More SQL Examples That Query XML	4-26
Updating XML Instances and Data in Tables and Columns	4-31
updateXML() SQL Function.....	4-31
Creating Views of XML Data with updateXML().....	4-35
updateXML() and NULL Values.....	4-35
Updating the Same XML Node More Than Once.....	4-36
XMLTransform() Function	4-36
Deleting XML Data.....	4-37
Using XMLType In Triggers.....	4-37
Indexing XMLType Columns	4-38
Creating Function-Based Indexes on XMLType Columns.....	4-38
Creating Oracle Text Indexes on XMLType Columns	4-39

5 Structured Mapping of XMLType

Introducing XML Schema	5-3
XML Schema and Oracle XML DB.....	5-3
Using Oracle XML DB and XML Schema	5-5
Why Do We Need XML Schema?	5-6
Introducing DBMS_XMLSCHEMA.....	5-7
Registering Your XML Schema Before Using Oracle XML DB	5-8
Registering Your XML Schema Using DBMS_XMLSCHEMA	5-8
Local and Global XML Schemas.....	5-10
Registering Your XML Schema: Oracle XML DB Sets Up the Storage and Access Infrastructure.....	5-12
Deleting Your XML Schema Using DBMS_XMLSCHEMA.....	5-13
Guidelines for Using Registered XML Schemas.....	5-14
Objects That Depend on Registered XML Schemas	5-14
Creating XMLType Tables, Views, or Columns.....	5-14
Validating XML Instances Against the XML Schema: schemaValidate()	5-15
Fully Qualified XML Schema URLs.....	5-16
Transactional Behavior of XML Schema Registration.....	5-17

Java Bean Generation During XML Schema Registration	5-17
Generating XML Schema Using DBMS_XMLSCHEMA.generateSchema()	5-18
XML Schema-Related Methods of XMLType	5-20
Managing and Storing XML Schema	5-20
Root XML Schema, XDBSchema.xsd	5-20
How Are XML Schema-Based XMLType Structures Stored?	5-21
DOM Fidelity	5-21
How Oracle XML DB Ensures DOM Fidelity with XML Schema	5-22
DOM Fidelity and SYS_XDBPD\$	5-22
Creating XMLType Tables and Columns Based on XML Schema	5-23
SQL Object-Relational Types Store XML Schema-Based XMLType Tables.....	5-24
Specifying SQL Object Type Names with SQLName, SQLType Attributes	5-25
SQL Mapping Is Specified in the XML Schema During Registration	5-29
Mapping of Types Using DBMS_XMLSCHEMA	5-32
Setting Attribute Mapping Type Information	5-32
Setting Element Mapping Type Information.....	5-32
XML Schema: Mapping SimpleTypes to SQL	5-34
simpleType: Mapping XML Strings to SQL VARCHAR2 Versus CLOBs	5-37
XML Schema: Mapping complexTypes to SQL	5-38
Setting the SQLInLine Attribute to FALSE for Out-of-Line Storage.....	5-38
Mapping XML Fragments to Large Objects (LOBs)	5-40
Oracle XML DB complexType Extensions and Restrictions	5-42
complexType Declarations in XML Schema: Handling Inheritance	5-42
Mapping complexType: simpleContent to Object Types	5-45
Mapping complexType: Any and AnyAttributes.....	5-46
Handling Cycling Between complexTypes in XML Schema.....	5-47
Further Guidelines for Creating XML Schema-Based XML Tables	5-50
Specifying Storage Clauses in XMLType CREATE TABLE Statements	5-51
Inserting New Instances into XMLType Columns	5-52
Query Rewrite with XML Schema-Based Structured Storage	5-52
What Is Query Rewrite?.....	5-52
When Does Query Rewrite Occur?	5-53
What XPath Expressions Are Rewritten?.....	5-54
How are the XPaths Rewritten?.....	5-56
Rewriting XPath Expressions: Mapping Types and Issues	5-58

XPath Expression Rewrites for existsNode()	5-63
Rewrite for extractValue()	5-66
Rewrite for extract().....	5-68
Optimizing Updates Using updateXML()	5-70
Creating Default Tables During XML Schema Registration.....	5-71
Ordered Collections in Tables (OCTs).....	5-72
Using OCT for VARRAY Storage.....	5-72
Cyclical References Between XML Schemas.....	5-72

6 Transforming and Validating XMLType Data

Transforming XMLType Instances.....	6-2
XMLTransform() and XMLType.transform()	6-2
XMLTransform() Examples	6-3
Validating XMLType Instances	6-8
Validating XML Data Stored as XMLType: Examples	6-10

7 Searching XML Data with Oracle Text

Searching XML Data with Oracle Text	7-3
Introducing Oracle Text.....	7-3
Assumptions Made in This Chapter's Examples.....	7-4
Oracle Text Users and Roles	7-5
Querying with the CONTAINS Operator.....	7-6
Using the WITHIN Operator to Narrow Query Down to Document Sections	7-7
Introducing SECTION_GROUPS	7-8
XML_SECTION_GROUP	7-8
AUTO_SECTION_GROUP/ PATH_SECTION_GROUP for INPATH and HASPATH	7-10
Dynamically Adding Sections or Stop Section Using ALTER INDEX	7-10
WITHIN Syntax for Section Querying	7-11
WITHIN Operator Limitations.....	7-11
INPATH or HASPATH Operators <i>Search Using XPath-Like Expressions</i>	7-12
Using INPATH Operator for Path Searching in XML Documents	7-12
Using HASPATH Operator for Path Searching in XML Documents	7-19
Building a Query Application with Oracle Text.....	7-21
What Role Do You Need?.....	7-21

Step 1. Create a Section Group Preference	7-21
Deciding Which Section Group to Use	7-23
Creating a Section Preference with XML_SECTION_GROUP	7-23
Creating a Section Preference with AUTO_SECTION_GROUP.....	7-23
Creating a Section Preference with PATH_SECTION_GROUP	7-24
Step 2. Set the Preference's Attributes	7-24
2.1 XML_SECTION_GROUP: Using CTX_DDL.add_zone_section.....	7-25
2.2 XML_SECTION_GROUP: Using CTX_DDL.Add_Attr_Section	7-25
2.3 XML_SECTION_GROUP: Using CTX_DDL.Add_Field_Section.....	7-26
2.5 AUTO_SECTION_GROUP: Using CtX_DDL.Add_Stop_Section.....	7-28
Step 3. Create an Index Using the Section Preference Created in Step 2	7-28
Step 4. Create Your Query Syntax	7-30
Querying Within Attribute Sections	7-30
Presenting the Results of Your Query	7-34
XMLType Indexing	7-34
You Need Query Rewrite Privileges.....	7-35
System Parameter is Set to the Default, CTXSYS.PATH_SECTION_GROUP.....	7-36
XMLType Indexes Work Like Other Oracle Text Indexes.....	7-36
Using Oracle Text with Oracle XML DB	7-37
Creating an Oracle Text Index on a UriType Column	7-37
Querying XML Data: Use CONTAINS or existsNode()?.....	7-38
Full-Text Search Functions in XPath Using ora:contains	7-40
ora:contains Features.....	7-40
ora:contains Syntax.....	7-40
ora:contains Examples.....	7-41
Oracle XML DB: Creating a Policy for ora:contains()	7-42
Oracle XML DB: Using CTXXPATH Indexes for existsNode()	7-45
Why do We Need CTXXPATH When ConText Indexes Can Perform XPath Searches?	7-45
CTXXPATH Index Type	7-46
Creating CTXXPATH Indexes	7-46
Creating CTXXPATH Storage Preferences with CTX_DDL. Statements	7-47
Performance Tuning CTXXPATH Index: Synchronizing and Optimizing the Index	7-47
Using Oracle Text: Advanced Techniques	7-49
Distinguishing Tags Across DocTypes.....	7-49
Specifying Doctype Limiters to Distinguish Between Tags	7-50

Doctype-Limited and Unlimited Tags in a Section Group.....	7-50
XML_SECTION_GROUP Attribute Sections	7-50
Constraints for Querying Attribute Sections.....	7-52
Repeated Zone Sections.....	7-53
Overlapping Zone Sections.....	7-54
Nested Sections.....	7-54
Using Table CTX_OBJECTS and CTX_OBJECT_ATTRIBUTES View.....	7-55
Case Study: Searching XML-Based Conference Proceedings	7-56
Searching for Content and Structure in XML Documents.....	7-56
Searching XML-Based Conference Proceedings Using Oracle Text	7-56
Searching Conference Proceedings Example: jsp	7-60
Frequently Asked Questions About Oracle Text	7-64
FAQs: General Questions About Oracle Text	7-64
Can I Use a CONTAINS() Query with an XML Function to Extract an XML Fragment?	7-64
Can XML Documents Be Queried Like Table Data?	7-64
Can I Edit Individual XML Elements?.....	7-65
How Are XML Files Locked in CLOBs and BLOBs?.....	7-65
How Can I Search XML Documents and Return a Zone?	7-65
How Do I Load XML Documents into the Database?.....	7-66
How Do I Search XML Documents with Oracle Text?.....	7-66
How Do I Search XML Using the WITHIN Operator?	7-66
Where Can I Find Examples of Using Oracle Text to Search XML?	7-67
Does Oracle Text Automatically Recognize XML Tags?	7-67
Can I Do Range Searching with Oracle Text?.....	7-68
Can Oracle Text Do Section Extraction?.....	7-68
Can I Create a Text Index on Three Columns?.....	7-68
How Fast Is Oracle9i at Indexing Text? Can I Just Enable Boolean Searches?	7-69
FAQs: Searching Attribute Values with Oracle Text	7-69
Can I Build Text Indexes on Attribute Values?.....	7-69
FAQs: Searching XML Documents in CLOBs Using Oracle Text	7-70
How Can I Search Different XML Documents Stored in CLOBs?.....	7-70
How Do I Store an XML Document in a CLOB Using Oracle Text?	7-71
Is Storing XML in CLOBs Affected by Character Set?	7-72
Can I Only Insert Structured Data When the Table is Created?.....	7-72
Can I Break an XML Document Without Creating a Custom Development?.....	7-72

What Is the Syntax for Creating a Substring Index with XML_SECTION_GROUP?.....	7-73
Why Does the XML Search for Topic X with Relevance Y Give Wrong Results?	7-74

Part III Using XMLType APIs to Manipulate XML Data

8 PL/SQL API for XMLType

Introducing PL/SQL APIs for XMLType	8-2
Backward Compatibility with XDK for PL/SQL, Oracle9i Release 1 (9.0.1)	8-2
PL/SQL APIs For XMLType Features.....	8-3
With PL/SQL APIs for XMLType You Can Modify and Store XML Elements	8-4
PL/SQL DOM API for XMLType (DBMS_XMLDOM)	8-5
Introducing W3C Document Object Model (DOM) Recommendation	8-5
PL/SQL DOM API for XMLType (DBMS_XMLDOM): Features	8-7
Designing End-to-End Applications Using XDK and Oracle XML DB.....	8-8
Using PL/SQL DOM API for XMLType: Preparing XML Data	8-9
Generating an XML Schema Mapping to SQL Object Types	8-10
Wrapping Existing Data into XML with XMLType Views.....	8-11
PL/SQL DOM API for XMLType (DBMS_XMLDOM) Methods.....	8-12
PL/SQL DOM API for XMLType (DBMS_XMLDOM) Exceptions	8-18
PL/SQL DOM API for XMLType: Node Types.....	8-18
Working with XML Schema-Based XML Instances.....	8-20
DOM NodeList and NamesNodeMap Objects.....	8-21
PL/SQL DOM API for XMLType (DBMS_XMLDOM): Calling Sequence	8-21
PL/SQL DOM API for XMLType Examples	8-22
PL/SQL Parser API for XMLType (DBMS_XMLPARSER)	8-24
PL/SQL Parser API for XMLType: Features	8-24
PL/SQL Parser API for XMLType (DBMS_XMLPARSER): Calling Sequence	8-26
PL/SQL Parser API for XMLType Example.....	8-27
PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)	8-28
Enabling Transformations and Conversions with XSLT	8-28
PL/SQL XSLT Processor for XMLType: Features.....	8-28
PL/SQL XSLT Processor API (DBMS_XSLPROCESSOR): Methods	8-29
PL/SQL Parser API for XMLType (DBMS_XSLPROCESSOR): Calling Sequence	8-30
PL/SQL XSLT Processor for XMLType Example	8-31

9 Java and Java Bean APIs for XMLType

Introducing Java DOM and Java Bean APIs for XMLType	9-2
Java DOM API for XMLType	9-2
Accessing XML Documents Stored in Oracle9i Database (Java)	9-3
Using JDBC to Manipulate XML Documents Stored in a Database.....	9-6
Java DOM API for XMLType Features	9-16
Java DOM API for XMLType Classes	9-18
Java DOM API for XMLType: Calling Sequence	9-19
Java Bean API for XMLType	9-20
Guidelines for Using Java Bean API for XMLType	9-21

Part IV Viewing Existing Data as XML

10 Generating XML Data from the Database

Oracle XML DB Options for Generating XML Data From Oracle9i Database	10-2
Generating XML Using SQLX Functions	10-2
Generating XML Using Oracle Extensions to SQLX	10-2
Generating XML Using DBMS_XMLGEN.....	10-2
Generating XML Using SQL Functions.....	10-2
Generating XML with XSQL Pages Publishing Framework.....	10-3
Generating XML Using XML SQL Utility (XSU)	10-3
Generating XML from the Database Using SQLX Functions	10-5
XMLElement() Function	10-5
XMLForest() Function	10-9
XMLSEQUENCE() Function	10-11
XMLConcat() Function	10-15
XMLAgg() Function	10-17
Generating XML from the Database Using SQLX Functions	10-19
XMLColAttVal() Function	10-19
Generating XML from Oracle9i Database Using DBMS_XMLGEN	10-20
Sample DBMS_XMLGEN Query Result	10-20
DBMS_XMLGEN Calling Sequence	10-21
Generating XML Using Oracle-Provided SQL Functions	10-41

SYS_XMLGEN() Function	10-41
Using XMLFormat Object Type.....	10-43
SYS_XMLAGG() Function	10-50
Generating XML Using XSQL Pages Publishing Framework	10-51
Generating XML Using XML SQL Utility (XSU)	10-54

11 XMLType Views

What Are XMLType Views?	11-2
Creating Non-Schema-Based XMLType Views	11-2
Creating XML Schema-Based XMLType Views	11-4
Using Multiple Namespaces	11-9
Creating XMLType Views by Transforming XMLType Tables	11-14
Referencing XMLType View Objects Using REF()	11-15
DML (Data Manipulation Language) on XMLType Views	11-15
Query Rewrite on XMLType Views	11-17
Query Rewrite on XML Schema-Based Views	11-17
Query Rewrite on Non-Schema-Based Views	11-17
Ad-Hoc Generation of XML Schema-Based XML	11-19
Validating User-Specified Information	11-20

12 Creating and Accessing Data Through URLs

How Oracle9i Database Works with URLs and URIs	12-2
URI Concepts	12-4
What Is a URI?.....	12-4
Advantages of Using DBUri and XDBUri.....	12-5
UriTypes Store Uri-References	12-6
Advantages of Using UriTypes	12-6
UriType Functions	12-7
HttpUriType Functions	12-8
getContentType() Function	12-9
getXML() Function.....	12-9
DBUri, Intra-Database References	12-9
Formulating the DBUri	12-10
Notation for DBUriType Fragments	12-12
DBUri Syntax Guidelines.....	12-13

Some Common DBUri Scenarios	12-14
Identifying the Whole Table	12-14
Identifying a Particular Row of the Table	12-15
Identifying a Target Column	12-16
Retrieving the Text Value of a Column	12-16
How DBUris Differ from Object References.....	12-17
DBUri Applies to a Database and Session	12-17
Where Can DBUri Be Used?	12-17
DBUriType Functions	12-18
XDBUriType	12-20
How to Create an Instance of XDBUriType.....	12-21
Creating Oracle Text Indexes on UriType Columns	12-22
Using UriType Objects	12-22
Storing Pointers to Documents with UriType.....	12-23
Using HttpUriType and DBUriType	12-25
Creating Instances of UriType Objects with the UriFactory Package	12-25
Registering New UriType Subtypes with the UriFactory Package.....	12-26
Why Define New Subtypes of UriType?	12-28
SYS_DBURIGEN() SQL Function	12-29
Rules for Passing Columns or Object Attributes to SYS_DBURIGEN()	12-30
SYS_DBURIGEN Examples.....	12-31
Turning a URL into a Database Query with DBUri Servlet	12-34
DBUri Servlet Mechanism	12-34
Installing DBUri Servlet.....	12-36
DBUri Security	12-37
Configuring the UriFactory Package to Handle DBUris	12-38

Part V Oracle XML DB Repository: Foldering, Security, and Protocols

13 Oracle XML DB Foldering

Introducing Oracle XML DB Foldering	13-2
Oracle XML DB Repository	13-4
Repository Terminology.....	13-4
Oracle XML DB Resources	13-6
Where Exactly Is Repository Data Stored?	13-6

Pathname Resolution	13-7
Deleting Resources	13-7
Accessing Oracle XML DB Repository Resources	13-8
Navigational or Path Access	13-9
Accessing Oracle XML DB Resources Using Internet Protocols.....	13-10
Query-Based Access	13-12
Accessing Repository Data Using Servlets	13-13
Accessing Data Stored in Oracle XML DB Repository Resources	13-14
Managing and Controlling Access to Resources	13-17
Extending Resource Metadata Properties	13-17

14 Oracle XML DB Versioning

Introducing Oracle XML DB Versioning	14-2
Oracle XML DB Versioning Features.....	14-2
Oracle XML DB Versioning Terms Used in This Chapter	14-3
Oracle XML DB Resource ID and Pathname	14-3
Creating a Version-Controlled Resource (VCR)	14-4
Version Resource or VCR Version	14-4
Resource ID of a New Version.....	14-5
Accessing a Version-Controlled Resource (VCR).....	14-6
Updating a Version-Controlled Resource (VCR).....	14-6
Access Control and Security of VCR	14-8
Frequently Asked Questions: Oracle XML DB Versioning	14-12
Can I Switch a VCR to a Non-VCR?.....	14-12
How Do I Access the Old Copy of a VCR After Updating It?	14-12
Can We Use Version Control for Data Other Than Oracle XML DB Data?	14-13

15 RESOURCE_VIEW and PATH_VIEW

Oracle XML DB RESOURCE_VIEW and PATH_VIEW	15-2
RESOURCE_VIEW Definition and Structure	15-3
PATH_VIEW Definition and Structure	15-3
Understanding the Difference Between RESOURCE_VIEW and PATH_VIEW.....	15-4
Operations You Can Perform Using UNDER_PATH and EQUALS_PATH.....	15-5
Resource_View, Path_View API	15-6
UNDER_PATH	15-6

EQUALS_PATH	15-8
PATH	15-8
DEPTH	15-8
Using the Resource View and Path View API	15-9
Accessing Paths and Repository Resources: Examples.....	15-9
Inserting Data into a Repository Resource: Examples	15-10
Deleting Repository Resources: Examples.....	15-10
Updating Repository Resources: Examples.....	15-11
Working with Multiple Oracle XML DB Resources Simultaneously	15-12
16 Oracle XML DB Resource API for PL/SQL (DBMS_XDB)	
Introducing Oracle XML DB Resource API for PL/SQL	16-2
Overview of DBMS_XDB	16-2
DBMS_XDB: Oracle XML DB Resource Management	16-2
Using DBMS_XDB to Manage Resources, Calling Sequence.....	16-3
DBMS_XDB: Oracle XML DB ACL-Based Security Management	16-5
Using DBMS_XDB to Manage Security, Calling Sequence	16-6
DBMS_XDB: Oracle XML DB Configuration Management	16-8
Using DBMS_XDB for Configuration Management, Calling Sequence	16-9
DBMS_XDB: Rebuilding Oracle XML DB Hierarchical Indexes	16-11
Using DBMS_XDB to Rebuild Hierarchical Indexes, Calling Sequence.....	16-11
17 Oracle XML DB Resource API for Java/JNDI	
Introducing Oracle XML DB Resource API for Java/JNDI	17-2
What Is JNDI?.....	17-2
JNDI Support in Oracle XML DB	17-2
Oracle XML DB Resource API for Java/JNDI Features.....	17-3
Using Oracle XML DB Resource API for Java/JNDI	17-4
Calling Sequence for Oracle XML DB Resource API for Java/JNDI	17-5
Parameters for Oracle XML DB Resource API for Java/JNDI	17-5
Using JNDI Contexts - XDBCContextFactory()	17-6
How JNDI's Context Module Returns Objects.....	17-7
JNDI Context Module's Inputs.....	17-8
JNDI Context Processing.....	17-9
Oracle XML DB Resource API for Java/JNDI Examples	17-11

18 Oracle XML DB Resource Security

Introducing Oracle XML DB Resource Security and ACLs	18-2
How the ACL-Based Security Mechanism Works	18-2
Access Control List Terminology	18-2
Oracle XML DB ACL Features	18-5
ACL Interaction with Oracle XML DB Table/View Security.....	18-5
LDAP Integration and User IDs	18-5
Oracle XML DB Resource API for ACLs (PL/SQL)	18-5
How Concurrency Issues Are Resolved with Oracle XML DB ACLs.....	18-5
Access Control: User and Group Access	18-6
ACE Elements Specify Access Privileges for Principals	18-6
Oracle XML DB Supported Privileges	18-7
Atomic Privileges.....	18-7
Aggregate Privileges	18-8
ACL Evaluation Rules	18-9
Using Oracle XML DB ACLs	18-9
Updating the Default ACL on a Folder	18-10
ACL and Resource Management	18-11
How to Set Resource Property ACLs.....	18-11
Default Assignment of ACLs	18-12
Retrieving ACLs for a Resource	18-12
Changing Privileges on a Given Resource.....	18-12
Restrictions for Operations on ACLs.....	18-12
Using DBMS_XDB to Check Privileges	18-13
Row-Level Security for Access Control Security	18-13

19 Using FTP, HTTP, and WebDAV Protocols

Introducing Oracle XML DB Protocol Server	19-2
Session Pooling.....	19-2
Oracle XML DB Protocol Server Configuration Management	19-3
Configuring Protocol Server Parameters	19-4
Interaction with Oracle XML DB Filesystem Resources	19-6
Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents	19-7
Using FTP and Oracle XML DB Protocol Server	19-7
Oracle XML DB Protocol Server: FTP Features	19-7

Using HTTP and Oracle XML DB Protocol Server	19-8
Oracle XML DB Protocol Server: HTTP Features	19-8
Using WebDAV and Oracle XML DB	19-10
Oracle XML DB WebDav Features.....	19-10
Using Oracle XML DB and WebDAV: Creating a WebFolder in Windows 2000.....	19-11

20 Writing Oracle XML DB Applications in Java

Introducing Oracle XML DB Java Applications	20-2
Which Oracle XML DB APIs Are Available Inside and Outside the Database?	20-2
Design Guidelines: Java Inside or Outside the Database?.....	20-3
HTTP: Accessing Java Servlets or Directly Accessing XMLType Resources.....	20-3
Accessing Many XMLType Object Elements: Use JDBC XMLType Support.....	20-3
Use the Servlets to Manipulate and Write Out Data Quickly as XML	20-3
Writing Oracle XML DB HTTP Servlets in Java	20-4
Configuring Oracle XML DB Servlets	20-4
HTTP Request Processing for Oracle XML DB Servlets.....	20-8
The Session Pool and XML DB Servlets	20-9
Native XML Stream Support	20-9
Oracle XML DB Servlet APIs	20-10
Oracle XML DB Servlet Example.....	20-10
Installing the Oracle XML DB Example Servlet	20-11
Configuring the Oracle XML DB Example Servlet.....	20-12
Testing the Example Servlet.....	20-12

Part VI Oracle Tools that Support Oracle XML DB

21 Managing Oracle XML DB Using Oracle Enterprise Manager

Introducing Oracle XML DB and Oracle Enterprise Manager	21-2
Getting Started with Oracle Enterprise Manager and Oracle XML DB	21-2
Oracle Enterprise Manager Oracle XML DB Features.....	21-3
Configure Oracle XML DB	21-4
Create and Manage Resources.....	21-4
Manage XML Schema and Related Database Objects.....	21-4

The Enterprise Manager Console for Oracle XML DB	21-7
XML Database Management Window: Right-Hand Dialog Windows	21-7
Hierarchical Navigation Tree: Navigator	21-7
Configuring Oracle XML DB with Enterprise Manager	21-7
Viewing or Editing Oracle XML DB Configuration Parameters	21-11
Creating and Managing Oracle XML DB Resources with Enterprise Manager	21-12
Administering Individual Resources	21-15
Individual Resource Content Menu	21-17
Enterprise Manager and Oracle XML DB: ACL Security	21-22
Granting and Revoking User Privileges with User > XML Tab	21-23
XML Database Resource Privileges	21-25
Managing XML Schema and Related Database Objects	21-27
Navigating XML Schema in Enterprise Manager	21-28
Registering an XML Schema	21-31
Creating Structured Storage Infrastructure Based on XML Schema	21-34
Creating an XMLType Table	21-35
Creating Tables with XMLType Columns	21-37
Creating a View Based on XML Schema	21-39
Creating a Function-Based Index Based on XPath Expressions	21-42

22 Loading XML Data into Oracle XML DB

Loading XMLType Data into Oracle9i Database	22-2
Restoration	22-2
Using SQL*Loader to Load XMLType Columns	22-2

Part VII XML Data Exchange Using Advanced Queueing

23 Exchanging XML Data Using Advanced Queueing (AQ)

What Is AQ?	23-2
How Do AQ and XML Complement Each Other?	23-2
XMLType Attributes in Object Types	23-5
Internet Data Access Presentation (IDAP)	23-5
IDAP Architecture	23-6
XMLType Queue Payloads	23-6

Enqueue Using AQ XML Servlet	23-9
Dequeue Using AQ XML Servlet	23-11
IDAP and AQ XML Schemas	23-12
Frequently Asked Questions (FAQs): XML and Advanced Queuing	23-13
Can I Store AQ XML Messages with Many PDFs as One Record?	23-13
Do I Specify Payload Type as CLOB First, Then Enqueue and Store?	23-13
Can I Add New Recipients After Messages Are Enqueued?	23-14
How Does Oracle Enqueue and Dequeue and Process XML Messages?.....	23-14
How Can I Parse Messages with XML Content from AQ Queues?.....	23-15
Can I Prevent the Listener from Stopping Until the XML Document Is Processed?.....	23-15
How Can I Use HTTPS with AQ?	23-16
What Are the Options for Storing XML in AQ Message Payloads?	23-16
Can We Compare IDAP and SOAP?	23-16

A Installing and Configuring Oracle XML DB

Installing Oracle XML DB	A-2
Installing or Reinstalling Oracle XML DB from Scratch	A-2
Installing a New Oracle XML DB with DBCA	A-2
Installing a New Oracle XML DB Manually Without DBCA.....	A-3
Reinstalling Oracle XML DB	A-4
Upgrading an Existing Oracle XML DB Installation	A-4
Configuring Oracle XML DB	A-4
Oracle XML DB Configuration File, xdbconfig.xml	A-5
Top Level Tag <xdbconfig>	A-5
<sysconfig>.....	A-5
<userconfig>.....	A-6
<protocolconfig>	A-6
<httpconfig>	A-6
Oracle XML DB Configuration Example	A-6
Oracle XML DB Configuration API	A-9
Get Configuration, <code>cfg_get()</code>	A-9
Update Configuration, <code>cfg_update()</code>	A-9
Refresh Configuration, <code>cfg_refresh()</code>	A-10

B XML Schema Primer

Introducing XML Schema	B-2
Purchase Order Schema, po.xsd	B-4
XML Schema Components	B-6
Complex Type Definitions, Element and Attribute Declarations.....	B-6
Naming Conflicts	B-12
Simple Types	B-13
List Types	B-17
Union Types	B-19
Anonymous Type Definitions	B-20
Element Content	B-21
Complex Types from Simple Types.....	B-21
Mixed Content.....	B-22
Empty Content	B-23
AnyType.....	B-24
Annotations	B-25
Building Content Models	B-26
Attribute Groups	B-29
Nil Values	B-31
How DTDs and XML Schema Differ	B-31
DTD Limitations	B-33
XML Schema Features Compared to DTD Features.....	B-34
Converting Existing DTDs to XML Schema?.....	B-37
XML Schema Example, PurchaseOrder.xsd	B-37

C XPath and Namespace Primer

Introducing the W3C XML Path Language (XPath) 1.0 Recommendation	C-2
The XPath Expression	C-3
Evaluating Expressions with Respect to a Context.....	C-3
XPath Expressions Often Occur in XML Attributes	C-4
Location Paths	C-5
Location Path Syntax Abbreviations.....	C-5
Location Path Examples Using Unabbreviated Syntax.....	C-5
Location Path Examples Using Abbreviated Syntax	C-7

Relative and Absolute Location Paths.....	C-9
Location Path Syntax Summary	C-10
XPath 1.0 Data Model	C-10
Nodes.....	C-11
Introducing the W3C XML Path Language (XPath) 2.0 Working Draft	C-17
XPath 2.0 Expressions	C-17
Introducing the W3C Namespaces in XML Recommendation	C-18
What Is a Namespace?	C-19
Qualified Names.....	C-21
Using Qualified Names	C-21
Namespace Constraint: Prefix Declared	C-22
Applying Namespaces to Elements and Attributes	C-23
Namespace Scoping	C-23
Namespace Defaulting.....	C-24
Uniqueness of Attributes.....	C-25
Conformance of XML Documents	C-26
Introducing the W3C XML Information Set	C-26
Namespaces.....	C-27
End-of-Line Handling.....	C-28
Base URIs	C-28
Unknown and No Value.....	C-29
Synthetic Infosets.....	C-29

D XSLT Primer

Introducing XSL	D-2
The W3C XSL Transformation Recommendation Version 1.0.....	D-2
Namespaces in XML	D-4
XSL Stylesheet Architecture.....	D-5
XSL Transformation (XSLT)	D-5
XSLT 1.1 Specification.....	D-5
XML Path Language (XPath)	D-6
CSS Versus XSL	D-7
XSL Stylesheet Example, PurchaseOrder.xsl	D-7

E Java DOM and Java Bean API for XMLType, Resource API for Java/JNDI: Quick Reference

Java DOM API For XMLType	E-2
Java Bean API for XMLType	E-6
Oracle XML DB Resource API for Java/JNDI.....	E-7

F Oracle XML DB XMLType API, PL/SQL and Resource PL/SQL APIs: Quick Reference

XMLType API.....	F-2
PL/SQL DOM API for XMLType (DBMS_XMLDOM)	F-6
PL/SQL Parser for XMLType (DBMS_XMLPARSER).....	F-13
PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR).....	F-14
DBMS_XMLSCHEMA.....	F-15
Oracle XML DB XML Schema Catalog Views.....	F-18
Resource API for PL/SQL (DBMS_XDB)	F-19
DBMS_XMLGEN.....	F-22
RESOURCE_VIEW, PATH_VIEW	F-23
DBMS_XDB_VERSION	F-24
DBMS_XDBT.....	F-25

G Example Setup scripts. Oracle XML DB- Supplied XML Schemas

Example Setup Scripts	G-2
Chapter 3 Examples Set Up Script: Creating User and Directory	G-2
Chapter 3 Examples Set Up Script: Granting Privileges, Creating Table.....	G-3
Chapter 3 Examples Script: invoice.xml.....	G-8
Chapter 3 Examples Script: PurchaseOrder.xml.....	G-9
Chapter 3 Examples Script: FTP Script.....	G-10
Chapter 3 Examples Script: Configuring FTP and HTTP Ports.....	G-11
Resource View and Path View Database and XML Schema	G-12
Resource View Definition and Structure.....	G-12
Path View Definition and Structure.....	G-12
XDBResource.xsd: XML Schema for Representing Oracle XML DB Resources	G-12
XDBResource.xsd.....	G-12

acl.xsd: XML Schema for Representing Oracle XML DBACLs	G-15
ACL Representation XML Schema, acl.xsd	G-15
acl.xsd	G-15
xdbconfig.xsd: XML Schema for Configuring Oracle XML DB	G-18
xdbconfig.xsd	G-18

Glossary

Index

Send Us Your Comments

Oracle9i XML Database Developer's Guide - Oracle XML DB, Release 2 (9.2)

Part No. A96620-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
Oracle Corporation
Server Technologies Documentation
500 Oracle Parkway, Mailstop 4op11
Redwood Shores, CA 94065
USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This manual describes Oracle XML DB, the Oracle9i XML database. It describes how XML data can be stored, generated, manipulated, managed, and queried in the database using Oracle XML DB.

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 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 Advanced Queueing `XMLType` support.

The Preface contains the following sections:

- [Audience](#)
- [Organization](#)
- [Related Documentation](#)
- [Conventions](#)
- [Documentation Accessibility](#)

Audience

This manual is intended for developers building XML applications on Oracle9i database.

Prerequisite Knowledge

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

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

Organization

This document contains the following parts, chapters, and appendixes:

PART I. Introducing Oracle XML DB

Introduces you to the Oracle XML DB components and architecture, including XMLType and the Repository. It discusses some basic design issues and provides a comprehensive set of examples of where and how you can use Oracle XML DB.

Chapter 1, "Introducing Oracle XML DB"

Introduces you to the Oracle XML DB components and architecture. It includes a description of the benefits of using Oracle XML DB, the key features, standards supported, and requirements for running Oracle XML DB. It lists Oracle XML DB-related terms used throughout the manual.

Chapter 2, "Getting Started with Oracle XML DB"

Describes how to install Oracle XML DB, compatibility and migration, and some preliminary application planning issues.

Chapter 3, "Using Oracle XML DB"

Introduces you to where and how you can use Oracle XML DB. It provides examples of storing, accessing, updating, and validating your XML data using Oracle XML DB.

PART II. Storing and Retrieving XML Data

Describes the ways you can store, retrieve, validate, and transform XML data using Oracle9i database native XMLType API.

Chapter 4, "Using XMLType"

Describes how to create XMLType tables and manipulate and query XML data for non-schema-based XMLType tables and columns.

Chapter 5, "Structured Mapping of XMLType"

Describes how to use Oracle XML DB mapping from SQL to XML and back, provides an overview of how you must register your XML schema, how you can either use Oracle XML DBs default mapping or specify your own mapping. It also describes how to use Ordered Collections in Tables (OCTs) in Oracle XML DB.

Chapter 6, "Transforming and Validating XMLType Data"

Describes how you can use SQL functions to transform XML data stored in the database and being retrieved or generated from the database. It also describes how you can use SQL functions to validate XML data being input into the database.

Chapter 7, "Searching XML Data with Oracle Text"

Describes how you can create an Oracle Text index on DBUriType or Oracle XML DB UriType columns and search XML data using Oracle Text's CONTAINS() function and XMLType's existsNode() function. It includes how to use CTXXPATH index for XPath querying of XML data.

PART III. Using XMLType APIs to Manipulate XML Data

Describes the PL/SQL and Java XMLType APIs and how to use them.

Chapter 8, "PL/SQL API for XMLType"

Introduces the PL/SQL DOM API for XMLType, PL/SQL Parser API for XMLType, and PL/SQL XSLT Processor API for XMLType. It includes examples and calling sequence diagrams.

Chapter 9, "Java and Java Bean APIs for XMLType"

Describes how to use the Java (JDBC) and Java Bean API for XMLType. It includes examples and calling sequence diagrams.

PART IV. Viewing Existing Data as XML

Chapter 10, "Generating XML Data from the Database"

Discusses SQLX, Oracle SQLX extension functions, and SQL functions for generating XML. SQLX functions include XMLElement() and XMLForest().

Oracle SQLX extension functions include `XMLColAttValue()`. SQL functions include `SYS_XMLGEN()`, `XMLSEQUENCE()`, and `SYS_XMLAGG()`. It also describes how to use `DBMS_XMLGEN`, XSQL Pages Publishing Framework, and XML SQL Utility (XSU) to generate XML data from data stored in the database.

Chapter 11, "XMLType Views"

Describes how to create `XMLType` views based on XML generation functions, object types, or transforming `XMLType` tables. It also discusses how to manipulate XML data in `XMLType` views.

Chapter 12, "Creating and Accessing Data Through URLs"

Introduces you to how Oracle9i database works with URIs and URLs. It describes how to use `UriTypes` and associated sub-types: `DBUriType`, `HttpUriType`, and `XDBUriType` to create and access database data using URLs. It also describes how to create instances of `UriType` using the `UriFactory` package, how to use `SYS_DBURIGEN()` SQL function, and how to turn a URL into a database query using `DBUri Servlet`.

PART V. Oracle XML DB Repository: Foldering, Security, and Protocols

Describes Oracle XML DB Repository, the concepts behind it, how to use Versioning, ACL security, the Protocol Server, and the various associated Oracle XML DB Resource APIs.

Chapter 13, "Oracle XML DB Foldering"

Describes hierarchical indexing and foldering. Introduces you to the various Oracle XML DB Repository components such as Oracle XML DB Resource View API, Versioning, Oracle XML DB Resource API for PL/SQL and Java.

Chapter 14, "Oracle XML DB Versioning"

Describes how to create a version-controlled Oracle XML DB resource (VCR) and how to access and update a VCR.

Chapter 15, "RESOURCE_VIEW and PATH_VIEW"

Describes how you can use SQL to access data stored in Oracle XML DB Repository using Oracle XML DB Resource View API. This chapter also compares the functionality of the other Oracle XML DB Resource APIs.

Chapter 16, "Oracle XML DB Resource API for PL/SQL (DBMS_XDB)"

Describes DBMS_Oracle XML DB and the Oracle XML DB Resource API for PL/SQL.

Chapter 17, "Oracle XML DB Resource API for Java/JNDI"

Describes Oracle XML DB Resource API for Java/JNDI and how to use it to access Oracle XML DB Repository data.

Chapter 18, "Oracle XML DB Resource Security"

Describes how to use Oracle XML DB resources and ACL security, how to share ACL, and how to retrieve ACL information.

Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"

Introduces Oracle XML DB Protocol Server and how to use FTP, HTTP, and WebDAV with Oracle XML DB.

Chapter 20, "Writing Oracle XML DB Applications in Java"

Introduces you to writing Oracle XML DB applications in Java. It describes which Java APIs are available inside and outside the database, tips for writing Oracle XML DB HTTP servlets, which parameters to use to configure servlets in the configuration file `/xdbconfig.xml`, and HTTP request processing.

PART VI. Oracle Tools That Support Oracle XML DB Development

Includes chapters that describe the tools you can use to build and manage your Oracle XML DB application.

Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"

Describes how you can use Oracle Enterprise Manager to register your XML schema; create resources, `XMLType` tables, views, and columns; manage ACL security, configure Oracle XML DB; and create function-based indexes.

Chapter 22, "Loading XML Data into Oracle XML DB"

Describes ways you can load `XMLType` data using `SQL*Loader`.

PART VII. XML Data Exchange Using Advanced Queueing

Describes Oracle Advanced Queueing support for XML and `XMLType` messaging.

Chapter 23, "Exchanging XML Data Using Advanced Queueing (AQ)"

Introduces how you can use Advanced Queueing to exchange XML data. It describes Internet Data Access Presentation (IDAP), using AQ XML Servlet to enqueue and dequeue messages, using IDAP and AQ XML schemas.

Appendix A, "Installing and Configuring Oracle XML DB"

Describes how to install and configure Oracle XML DB.

Appendix B, "XML Schema Primer"

Provides a summary of the W3C XML Schema Recommendation.

Appendix C, "XPath and Namespace Primer"

Provides an introduction to W3C XPath Recommendation, Namespace Recommendation, and Information Sets.

Appendix D, "XSLT Primer"

Provides an introduction to the W3C XSL/XSLT Recommendation.

Appendix E, "Java DOM and Java Bean API for XMLType, Resource API for Java/JNDI: Quick Reference"

Provides a quick reference for the Oracle XML DB Java APIs.

Appendix F, "Oracle XML DB XMLType API, PL/SQL and Resource PL/SQL APIs: Quick Reference"

Provides a quick reference for the Oracle XML DB PL/SQL APIs.

Appendix G, "Example Setup scripts. Oracle XML DB- Supplied XML Schemas"

Provides a description of the setup scripts used for the examples in Chapter 3. It also describes the RESOURCE_VIEW and PATH_VIEW structures and lists the Oracle XML DB supplied sample resource XML schema.

Glossary

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i Database New Features* for information about the differences between Oracle9i and the Oracle9i Enterprise Edition and the available features and options. That book also describes all the features that are new in Oracle9i Release 2 (9.2).
- *Oracle9i XML Developer's Kits Guide - XDK*
- Oracle9i XML Case Studies and Applications (contains XDK examples, no Oracle XML DB examples for this release)
- *Oracle9i XML Developer's Kits Guide - XDK*
- *Oracle9i Database Error Messages*
- *Oracle Text Application Developer's Guide*
- *Oracle Text Reference*
- *Oracle9i Database Concepts*.
- *Oracle9i Java Developer's Guide*
- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle9i Application Developer's Guide - Advanced Queuing*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*

Some of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

<http://oraclestore.oracle.com/>

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

<http://www.oraclebookshop.com/>

Other customers can contact their Oracle representative to purchase printed documentation.

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

<http://otn.oracle.com/admin/account/membership.html>

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

<http://otn.oracle.com/docs/index.htm>

To access the database documentation search engine directly, please visit

<http://tahiti.oracle.com>

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- [Conventions in Text](#)
- [Conventions in Code Examples](#)

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an index-organized table .
<i>Italics</i>	Italic typeface indicates book titles or emphasis.	<i>Oracle9i Database Concepts</i> Ensure that the recovery catalog and target database do <i>not</i> reside on the same disk.

Convention	Meaning	Example
UPPERCASE monospace (fixed-width) font	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a NUMBER column. You can back up the database by using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES data dictionary view. Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter sqlplus to open SQL*Plus. The password is specified in the orapwd file. Back up the datafiles and control files in the /disk1/oracle/dbs directory. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user. The JRepUtil class implements these methods.
<i>lowercase italic monospace (fixed-width) font</i>	Lowercase italic monospace font represents placeholders or variables.	You can specify the <i>parallel_clause</i> . Run <i>Uold_release</i> .SQL where <i>old_release</i> refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[]	Brackets enclose one or more optional items. Do not enter the brackets.	DECIMAL (<i>digits</i> [, <i>precision</i>])
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	{ENABLE DISABLE}
	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE DISABLE} [COMPRESS NOCOMPRESS]
...	Horizontal ellipsis points indicate either: <ul style="list-style-type: none"> That we have omitted parts of the code that are not directly related to the example That you can repeat a portion of the code 	CREATE TABLE... AS <i>subquery</i> ; SELECT <i>col1, col2, ... , coln</i> FROM employees;
.	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
<i>Italics</i>	Italicized text indicates placeholders or variables for which you must supply particular values.	CONNECT SYSTEM/ <i>system_password</i> DB_NAME = <i>database_name</i>
UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files. Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	SELECT last_name, employee_id FROM employees; sqlplus hr/hr CREATE USER mjones IDENTIFIED BY ty3MU9;

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

<http://www.oracle.com/accessibility/>

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

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

What's New In Oracle XML DB?

This chapter describes the new features, enhancements, APIs, and product integration added through Oracle XML DB as a part of Oracle9i Release 2 (9.2).

Oracle XML DB: XMLType Enhancements

`XMLType` datatype was first introduced in Oracle9i. This datatype has been significantly enhanced in Oracle9i Release 2 (9.2). The following sections describe these enhancements.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

XMLType Tables

Datatype `XMLType` can now be used to create tables of `XMLType`. This gives you the flexibility to store XML either in a column, in a table, or as a whole table, much like objects.

XMLType Constructors

Additional `XMLType` constructors have been added. Besides the `createXML()` functions, `XMLType` can now also be constructed using user-defined constructors.

W3C XML Schema Support

Extensive XML schema support has been added in this release to Oracle XML DB. You can now perform the following:

- Construct an `XMLType` object based on an XML schema and have it be continuously validated.

- Create XML schema-based `XMLType` tables. This feature automatically creates appropriate storage structures for optimal storage of XML schema-based documents. Unlike SQL DDL, this process does not require knowledge of all column data types and their definitions.
- Register annotated XML schema using the `DBMS_XMLSCHEMA` package, to share storage and type definitions. Registered XML schema can be shared across all database users to allow for instance wide common document definition. The registration process optionally creates Java beans and default tables. With XML schema annotation you can specify various objects such as, SQL types, default storage tables, and so on.
- Pre-parse incoming XML documents and automatically direct them to default storage tables. This allows protocols such as FTP and WebDAV to accept structured XML documents and store them in object-relational tables.
- Automatically validate XML documents or instances against W3C XML Schema when the XML documents or instances are added to Oracle XML DB.
- Explicitly validate XML documents and instances against XML schema using `XMLIsValid()` method on `XMLType`.
- Use datatype-aware extraction of part of an XML document using the `extractValue` operator.

SQLX Functions and Oracle Extensions

This release includes support for SQLX operations for generating XML from existing relational and object relational tables. This is based on ISO-ANSI Working Draft for XML-Related Specifications (SQL/XML) [ISO/IEC 9075 Part 14 and ANSI] which defines ways in which the database language SQL can be used in conjunction with XML.

For example, the following functions defined by the SQLX standards body are supported: `XMLElement()`, `XMLForest()`, `XMLConcat()`, and `XMLAgg()`. Oracle XML DB also extends the SQLX operations with functions such as: `XMLColAttVal()`, `XMLSequence()`, `SYS_XMLGEN()`, and `SYS_XMLAGG()`.

W3C XPath Support for Extraction, Condition Checks, and Updates

Oracle9i Release 1 (9.0.1) provided the `extract()` and `existsNode()` functions on `XMLType` objects. These allowed XPath-based queries against XML documents. This release provides additional support as follows:

- `extract()`, `existsNode()`, and `extractValue()` now allow for a namespace-based operation.
- `extract()`, `existsNode()`, and `extractValue()` support the full XPath function set, including axis operators.
- `updateXML()` function (new) replaces part of the `XMLType` DOM by using XPath as a locator.

ToObject Method

`ToObject` method allows the caller to convert an `XMLType` object to a PL/SQL object type.

XMLType Views

This release supports `XMLType`-based views. These enable you to view any data in the database as XML. `XMLType` views can be XML schema-based or non-XML schema-based. See [Chapter 11, "XMLType Views"](#).

W3C XSLT Support

This release introduces a new function, `XMLTransform()` that allows for a database-based transformation of in-memory or on disk XML documents. See [Chapter 6, "Transforming and Validating XMLType Data"](#).

JDBC Support for XMLType

Oracle XML DB allows database clients to bind and define `XMLType`. JDBC support includes a function-rich `XMLType` class that allows for native (for thick JDBC) XML functionality support. See [Chapter 9, "Java and Java Bean APIs for XMLType"](#).

C-Based PL/SQL DOM, Parser, and XSLT APIs

This release includes native PL/SQL DOM, Parser, and XSLT APIs integrated in the database code. These PL/SQL APIs are compatible with the Java-based PLSQL APIs shipped as part of XDK for PL/SQL with Oracle9i Release 1 (9.0.1) and higher. See [Chapter 8, "PL/SQL API for XMLType"](#).

Oracle XML DB: Repository

In this release, Oracle XML DB Repository adds advanced foldering and security mechanisms to the database. Oracle XML DB Repository is a new feature that provides a novel file system-like access to all database data. The Repository allows the following actions:

- Viewing the database and its content as a file system containing resources, typically referred to as files and folders.
- Access and manipulation of resources through path name-based SQL and Java API.
- Access and manipulation of resources through built-in native Protocol Servers for FTP, HTTP, and WebDAV.
- ACL-based security for Oracle XML DB resources.

Oracle XML DB Resource API (PL/SQL): DBMS_XDB

DBMS_XDB package provides methods to access and manipulate Oracle XML DB resources. [Chapter 16, "Oracle XML DB Resource API for PL/SQL \(DBMS_XDB\)"](#).

Oracle XML DB Resource API (JNDI)

This uses JNDI (Java Naming and Directory Interface) to locate resources, and manage collections. It supports JNDI Service Provider Interface (SPI). This interface works only inside the database server on the JServer platform. See [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#).

Oracle XML DB Resource View API (SQL)

ResourceView is a public XMLType view that you can use to perform path name-based queries against all resources in a database instance. This view merges path-based queries with queries against relational and object-relational tables and views. See [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#).

Oracle XML DB Versioning: DBMS_XDB_VERSION

DBMS_XDB_VERSION package provides methods to version Oracle XML DB resources. See [Chapter 14, "Oracle XML DB Versioning"](#).

Oracle XML DB ACL Security

Methods that implement ACL-based security are a part of DBMS_XDB package. They allow you to create high-performance access control lists for any XMLType object. See [Chapter 18, "Oracle XML DB Resource Security"](#).

Oracle XML DB Protocol Servers

The Protocol Servers provide access to any foldered XMLType row through FTP, HTTP, and WebDAV. Note that XMLType can manage arbitrary binary data as well in any file format. See [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#).

XDBURIType

`URIType` now includes a new subtype, `XDBURIType`, that represents a path name within Oracle XML DB. See [Chapter 12, "Creating and Accessing Data Through URLs"](#).

Oracle Tools Enhancements for Oracle XML DB

Oracle Enterprise Manager

Oracle Enterprise Manager provides a graphical interface to manage, administer, and configure Oracle XML DB. See [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#).

Oracle Text Enhancements

This release offers the following Oracle Text enhancements:

- Columns of type `XMLType` can now be indexed natively in Oracle9i database using Oracle Text.
- `CONTAINS()` is a new function for use as `ora:contains` in an XPath query and as part of the `existsNode()` function.
- `CTXXPATH` is a new index type for use with `existsNode()` to speedup the performance of XPath searching.

See [Chapter 7, "Searching XML Data with Oracle Text"](#).

Oracle Advanced Queuing (AQ) Support

With this release, the Advanced Queueing (AQ) Internet Data Access Presentation (IDAP) has been enhanced. IDAP facilitates your using AQ over the Internet. You can now use AQ XML servlet to access Oracle9i AQ using HTTP and SOAP.

Also in this release, IDAP is the Simple Object Access Protocol (SOAP) implementation for AQ operations. IDAP now defines the XML message structure used in the body of the SOAP request.

You can now use `XMLType` as the AQ payload type instead of having to embed `XMLType` as an attribute in an Oracle object type.

See:

- [Chapter 23, "Exchanging XML Data Using Advanced Queuing \(AQ\)"](#)
- *Oracle9i Application Developer's Guide - Advanced Queuing*

Oracle XDK Support for XMLType

XDK for Java Support

XSQL Servlet and XML SQL Utility (XSU) for Java now support XMLType. Most methods on XMLType object, such as, `getClobVal()`, are now available in XSU for Java.

XDK for PLSQL Support

XML SQL Utility (XSU) for PLSQL now supports XMLType.

See:

- ["Generating XML Using XSQL Pages Publishing Framework"](#) on page 10-51. and ["Generating XML Using XML SQL Utility \(XSU\)"](#) on page 10-54
- *Oracle9i XML Developer's Kits Guide - XDK*

Part I

Introducing Oracle XML DB

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

- [Chapter 1, "Introducing Oracle XML DB"](#)
- [Chapter 2, "Getting Started with Oracle XML DB"](#)
- [Chapter 3, "Using Oracle XML DB"](#)

Introducing Oracle XML DB

This chapter introduces you to Oracle XML DB by describing the Oracle XML DB benefits, features, and architecture. This chapter contains the following sections:

- [Introducing Oracle XML DB](#)
- [Benefits of Oracle XML DB](#)
- [Key Features of Oracle XML DB](#)
- [Oracle XML DB and XML Schema](#)
- [Oracle XML DB Architecture](#)
- [XMLType Storage Architecture](#)
- [Why Use Oracle XML DB?](#)
- [Searching XML Data Stored in CLOBs Using Oracle Text](#)
- [Building Oracle XML DB XML Messaging Applications with Advanced Queuing](#)
- [Managing Oracle XML DB Applications with Oracle Enterprise Manager](#)
- [Requirements for Running Oracle XML DB](#)
- [Oracle XML DB Technical Support](#)
- [Terminology Used in This Manual](#)
- [Oracle XML DB Examples Used in This Manual](#)

Introducing Oracle XML DB

This chapter introduces you to Oracle XML DB. It discusses the features available for building XML applications on the Oracle9i database.

From its beginnings, XML's core characteristics of self-description and dynamic extensibility have provided the flexibility needed to transport messages between various applications, and loosely couple distributed business processes.

XML is also language-independent and platform-independent. As XML support has become standard in browsers, application servers, and databases, enterprises have wished to tie legacy applications to the Web using XML to transform various proprietary file- and document-exchange templates into XML.

More recently, a new generation of XML standards, such as XML Schema, has enabled a unified data model that can address both structured data and documents. XML Schema has emerged as a key innovation in managing document content with the same rigor as data by enabling documents marked up as XML to move into the database.

Oracle XML DB is a set of built-in high-performance storage and retrieval technologies geared to XML. Oracle XML DB fully absorbs the World Wide Web Consortium (W3C) XML data model into Oracle9i database and provides new standard access methods for navigating and querying XML. You get all the advantages of relational database technology and XML technology at the same time. Oracle XML DB can be used to store, query, update, transform, or otherwise process XML, while at the same time providing SQL access to the same XML data.

Not a Separate Database Server

Oracle XML DB is not some separate server but rather the name for a distinct group of technologies related to high-performance XML storage and retrieval that are available within the familiar Oracle database. Oracle XML DB can also be thought of as an evolution of the Oracle database that encompasses both SQL and XML data models in a highly interoperable manner, thus providing native XML support.

You use Oracle XML DB in conjunction with Oracle XML Developer's Kit (XDKs). XDKs provide common development-time utilities that can run in the middle tier in Oracle9iAS or in Oracle9i database.

See Also: *Oracle9i XML Developer's Kits Guide - XDK*. for more information about XDK

Benefits of Oracle XML DB

Applications often manage structured data as tables and unstructured data as files or Large Objects (LOBs). This subjects developers to different paradigms for managing different kinds of data. Systems channel application-development down either of the following paths:

- **Unstructured.** This typically makes document access transparent and table access complex.
- **Structured.** This typically makes document access complex and table access transparent.

Oracle XML DB provides the following benefits:

- The ability to store and manage both structured and unstructured data under the same standard W3C XML data model (XML Schema).
- Complete transparency and interchangeability between the XML and SQL data views.
- Valuable Repository functionality: foldering, access control, FTP, and WebDAV protocol support with versioning. This enables applications to retain the file abstraction when manipulating XML data brought into Oracle. As a result, you can store XML in the database (rendering it queryable) and at the same time access it through popular desktop tools.
- Better management of unstructured XML data by supporting
 - Piecewise updates
 - XML indexing
 - Integrated XML text search with Oracle Text
 - Multiple views on the data, including relational views for SQL access
 - Enforcement of intra-document and inter-document relationships in XML documents
- Users today face a performance barrier in storing and retrieving complex XML. Oracle XML DB provides high performance and scalability for XML operations with the help of a number of specific optimizations that relate to XML-specific data-caching and memory management, query optimization on XML, special hierarchical indexes on the XML Repository, and so on.
- Enables data and documents from disparate systems to be accessed, for example, through Oracle Gateway and External Tables, and combined into a

standard data model. This integrative aspect reduces the complexity of developing applications that must deal with data from different stores.

Key Features of Oracle XML DB

[Table 1–1](#) describes Oracle XML DB features. This list includes XML features available since Oracle9i Release 1 (9.0.1).

Table 1–1 Oracle XML DB Features

Oracle XML DB Features	Description
XMLType	<p>The native datatype <code>XMLType</code> helps store and manipulate XML. Multiple storage options (Character Large Object (CLOB), structured XML) are available with <code>XMLType</code>, and administrators can choose a storage that meets their requirements. CLOB storage is an un-decomposed storage that is like an image of the original XML.</p> <p>The native structured XML storage is a shredded decomposition of XML into underlying object-relational structures (automatically created and managed by Oracle) for better SQL queriability.</p> <p>With <code>XMLType</code>, you can perform SQL operations such as:</p> <ul style="list-style-type: none"> ■ Queries, OLAP function invocations, and so on, on XML data, as well as XML operations ■ XPath searches, XSL transformations, and so on, on SQL data <p>You can build regular SQL indexes or Oracle Text indexes on <code>XMLType</code> for high performance for a very broad spectrum of applications. See Chapter 4, "Using XMLType".</p>
DOM fidelity	<p>The Document Object Model (DOM) is a standard programmatic representation of XML documents. Oracle XML DB can shred XML documents while storing them (structured XML Storage) in a manner that maintains DOM fidelity: the DOM that you store is the DOM that you get back. DOM fidelity means that your programs can manipulate exactly the same XML data that you got, and the process of storage does not affect the order of elements, the presence of namespaces and so on. DOM fidelity does not, however, imply maintenance of whitespaces, and the like; if you want to preserve the exact layout of XML including whitespaces you can use CLOB storage. See Chapter 5, "Structured Mapping of XMLType".</p>
Document fidelity	<p>For applications that need to store XML while maintaining complete fidelity to the original, including whitespace characters, the CLOB storage option is available.</p>

Table 1–1 Oracle XML DB Features (Cont.)

Oracle XML DB Features	Description
XML Schema	Oracle XML DB gives you the ability to constrain XML documents to XML schemas. You can create tables and types automatically given a W3C standard XML Schema. You can also enforce that an XML document being stored is schema-valid. This means you have a standard data model for all your data (structured and unstructured) and can use the database to enforce this data model. See Chapter 5, "Structured Mapping of XMLType" .
XML Schema storage with DOM fidelity	Use structured storage (object-relational) columns, VARRAYs, nested tables, and LOBs to store any element or element-subtree in your XML schema and still maintain DOM fidelity (DOM stored == DOM retrieved). See Chapter 5, "Structured Mapping of XMLType" . Note: If you choose CLOB storage option, available with XMLType since Oracle9i Release 1 (9.0.1), you can keep whitespaces.
XML Schema validation	While storing XML documents in Oracle XML DB you can optionally ensure that their structure complies (is "valid" against) with specific XML Schema. See Chapter 6, "Transforming and Validating XMLType Data" .
XML piecewise update	You can use XPath 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 Chapter 5, "Structured Mapping of XMLType" .
XPath search	You can use XPath syntax (embedded in an SQL statement or as part of an HTTP request) to query XML content in the database. See Chapter 4, "Using XMLType" and Chapter 7, "Searching XML Data with Oracle Text" .
XML indexes	Use XPath to specify parts of your document to create indexes for XPath searches. Enables fast access to XML documents. See Chapter 4, "Using XMLType" .
SQLX operators	New SQL member functions tracking the emerging ANSI SQLX standard, such as, XMLElement (to create XML elements on the fly) and others, to make XML queries and on-the-fly XML generation easy. These render SQL and XML metaphors interoperable. See Chapter 10, "Generating XML Data from the Database" .
XSL transformations for XMLType	Use XSLT to transform XML documents through an SQL operator. Database-resident, high-performance XSL transformations. See Chapter 6, "Transforming and Validating XMLType Data" and Appendix D, "XSLT Primer" .
Lazy XML loading	Oracle XML DB provides a virtual DOM; it only loads rows of data as they are requested, throwing away previously referenced sections of the document if memory usage grows too large. You can use this to get high scalability when many concurrent users are dealing with large XML documents. The virtual DOM is available through Java interfaces running in a Java execution environment at the client or with the server. See Chapter 8, "PL/SQL API for XMLType" .

Table 1–1 Oracle XML DB Features (Cont.)

Oracle XML DB Features	Description
XML views	You can create XML views to create permanent aggregations of various XML document fragments or relational tables. You can also create views over heterogeneous data sources using Oracle Gateways. See Chapter 11, "XMLType Views" .
Java Bean interface	A Java Bean Interface for fast access to structured XML data has extensions that save only those parts that have been modified in memory. With this you get static access to XML as well as dynamic (DOM) access. See Chapter 9, "Java and Java Bean APIs for XMLType" .
PL/SQL and OCI interfaces	Use DOM and other APIs for accessing and manipulating XML data. You can get static and dynamic access to XML. See Chapter 8, "PL/SQL API for XMLType" .
Schema caching	Structural information (such as element tags, datatypes, and storage location) is kept in a special schema cache, to minimize access time and storage costs. See Chapter 5, "Structured Mapping of XMLType" .
XML generation	SQL operators such as <code>SYS_XMLGEN</code> and <code>SYS_XMLAGG</code> provide native, high-performance generation of XML from SQL queries. New operators such as <code>XMLElement()</code> , to create XML tables and elements on the fly, make XML generation more flexible. See Chapter 10, "Generating XML Data from the Database" . These operators track the emerging ANSI SQLX standard.
Oracle XML DB Repository	<p>A built-in XML Repository. This Repository can be used for <i>foldering</i> whereby you can view XML content stored in Oracle XML DB as a hierarchy of directory-like folders. See Chapter 13, "Oracle XML DB Foldering".</p> <ul style="list-style-type: none"> ■ The repository supports <i>access control lists</i> (ACLs) for any <code>XMLType</code> object, and lets you define your own privileges in addition to providing certain system-defined ones. See Chapter 18, "Oracle XML DB Resource Security". ■ You can use the Repository to view XML content as navigable directories through a number of popular clients and desktop tools. Items managed by the repository are called <i>resources</i>. ■ Hierarchical indexing is enabled on the Repository. Oracle XML DB provides a special hierarchical index to speed folder search. Additionally, you can automatically map hierarchical data in relational tables into folders (where the hierarchy is defined by existing relational information, such as with <code>CONNECT BY</code>).

Table 1–1 Oracle XML DB Features (Cont.)

Oracle XML DB Features	Description
SQL Repository search	You can search the XML Repository using SQL. Operators like UNDER_PATH and DEPTH allow applications to search folders, XML file metadata (such as owner and creation date), and XML file content. See Chapter 15, "RESOURCE_VIEW and PATH_VIEW" .
WebDav, HTTP, and FTP access	You can access any foldered XMLType row using WebDAV and FTP. Users manipulating XML data in the Oracle9i database can use the HTTP API. See Chapter 19, "Using FTP, HTTP, and WebDAV Protocols" .
Versioning	Oracle XML DB provides versioning and version-management capabilities over resources managed by the XML Repository. See Chapter 14, "Oracle XML DB Versioning" .

Oracle XML DB and XML Schema

XML schema unifies both *document* and *data* modeling. In Oracle XML DB, you can create tables and types automatically using XML Schema. In short, this means that you can develop and use a standard data model for *all* your data, structured, unstructured, and pseudo/semi-structured. You can now use Oracle9i database to enforce this data model for all your data.

You can create XML schema-based XMLType tables and columns and optionally specify, for example, that they:

- Conform to pre-registered XML schema
- Are stored in structured storage format specified by the XML schema maintaining DOM fidelity

You can also choose to wrap existing relational and object-relational data into XML format using XMLType views.

You can store an XMLType object as an XML schema-based object or a non-XML schema-based object:

- *XML Schema-based objects.* These are stored in Oracle XML DB as LOBs or in structured storage (object-relationally) in tables, columns, or views.
- *Non-XML schema-based objects.* These are stored in Oracle XML DB as Large Objects (LOBs).

You can map from XML instances to structured or LOB storage. The mapping can be specified in XML schema and the XML schema must be registered in Oracle XML

DB. This is a required step before storing XML schema-based instance documents. Once registered, the XML schema can be referenced using its URL.

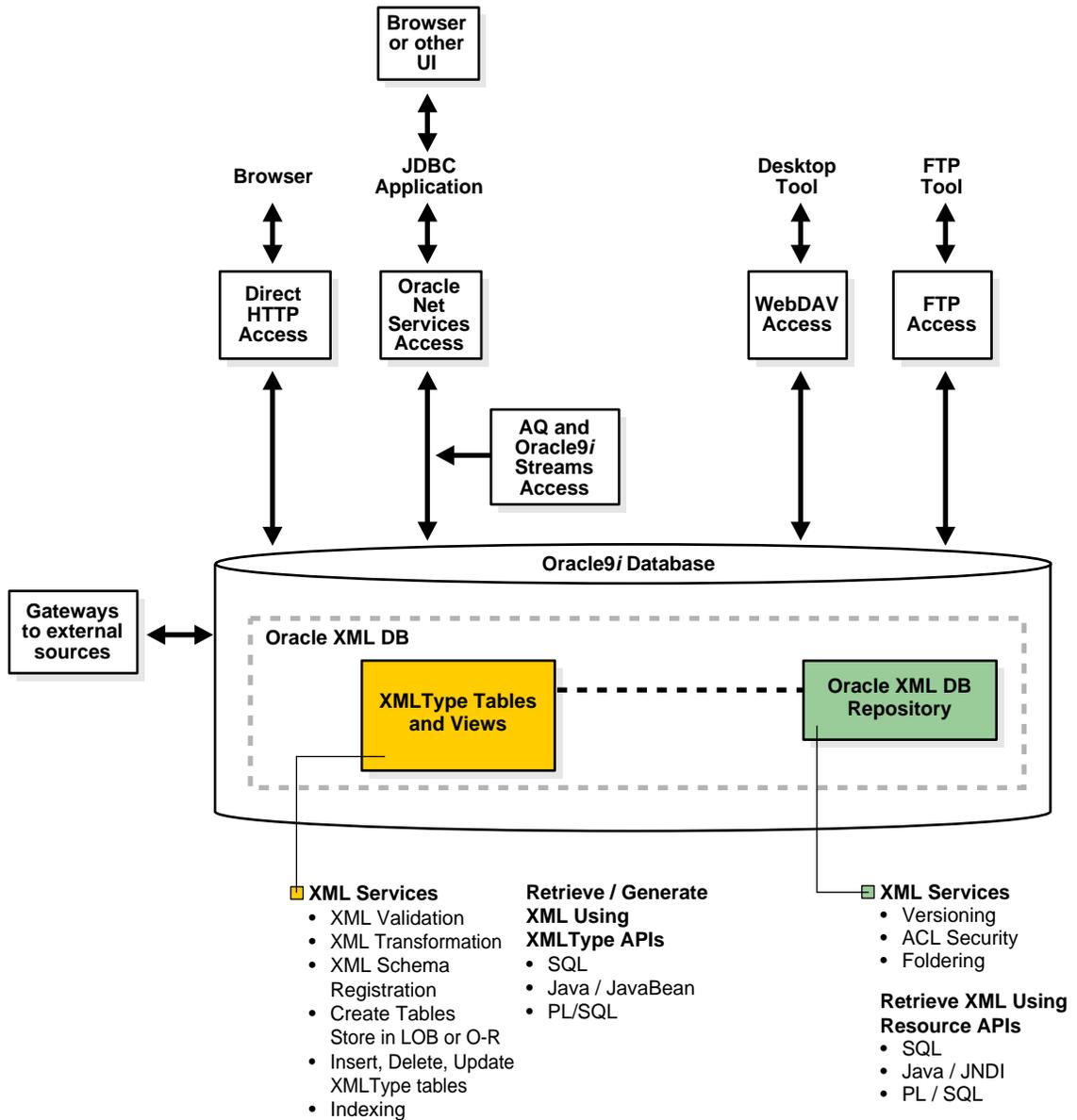
Oracle XML DB Architecture

[Figure 1-1](#) shows the Oracle XML DB architecture. The two main features in Oracle XML DB architecture are:

- **The `XMLType` tables and views storage**, which includes storage of `XMLType` tables and views
- **The Oracle XML DB Repository**, also referred to in this manual as "XML Repository" or "Repository"

The section following [Figure 1-1](#) describes the architecture in more detail.

Figure 1-1 Oracle XML DB Architecture: XMLType Storage and Repository



XMLType Tables and Views Storage

"XMLType tables and views storage" in Oracle XML DB provides a native XML storage and retrieval capability in the database, strongly integrated with SQL.

XML data, including XML schema definition files can be stored in LOBs, in structured storage (object-relationally), or using any hybrid combining both LOBs and structured storage. See [Chapter 3, "Using Oracle XML DB"](#) and [Chapter 4, "Using XMLType"](#).

Supported XML Access APIs

- PL/SQL and Java APIs for XMLType. Use these APIs to:
 - Create XMLType tables, columns, and views
 - Query and retrieve XML data
- SQL functions, such as `XMLElement()` and `XMLForest()`. Applications can query XML data in the database using standard SQL and SQL member functions that comply with the SQLX standard.

See Also: Part IV. [Viewing Existing Data as XML](#).

Supported XML Services

In Oracle XML DB, besides accessing or generating XML data, you can also perform various operations on the data:

- *PL/SQL and Java APIs for XMLType*. These enable you to manipulate XMLType data, such as update, delete, and insert XML data.
- *Indexing*. This speeds up data searches where XPath features are not critical. It is most suited for XML data stored in LOBs.
- *Transforming XML data* to other XML, HTML, and so on, using XMLType's `XMLTransform()` function, XDK's XSLT Processors, or XSQL Servlet Pages Publishing Framework. See [Chapter 6, "Transforming and Validating XMLType Data"](#) and [Chapter 10, "Generating XML Data from the Database"](#).
- *Validating XML data*. Validates XML data against XML schema when the XML data is stored in the database.

See Also: ["XMLType Storage Architecture"](#) on page 1-12.

Oracle XML DB Repository

Oracle XML DB Repository (XML Repository or Repository) is an XML data repository in the Oracle9i database optimized for handling XML data. At the heart of Oracle XML DB Repository is the Oracle XML DB foldering module.

See Also: [Chapter 13, "Oracle XML DB Foldering"](#).

The contents of Oracle XML DB Repository are referred to as *resources*. These can be either containers (or directories / folders) or files. All resources are identified by a path name and have a (extensible) set of (metadata) properties such as Owner, CreationDate, and so on, in addition to the actual contents defined by the user.

Supported XML Access APIs

[Figure 1–1](#) lists the following Oracle XML DB supported XML access and manipulation APIs:

- *Oracle XML DB Resource APIs.* Use these APIs to access the *foldered* XMLType and other data, that is, data accessed using the Oracle XML DB hierarchically indexed Repository. The APIs are available in the following languages:
 - SQL (through the RESOURCE_VIEW and PATH_VIEW APIs)
 - PL/SQL (DBMS_XDB) API
 - JNDI (Java/JNI) API

See Also: [Part V. Oracle XML DB Repository: Foldering, Security, and Protocols](#)

- *Oracle XML DB Protocol Server.* Oracle XML DB supports FTP, HTTP, and WebDav protocols, as well as JDBC, for fast access of XML data stored in the database in XMLType tables and columns. See [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#).

Supported XML Services

XML Repository, besides supporting APIs to access and manipulate XML and other data, also supports the following services:

- **Versioning.** Oracle XML DB provides support for versioning resources. The DBMS_XDB_VERSION PL/SQL package implements functions to make a resource version-controlled. Any subsequent updates to the resource results in

new versions being created while the data corresponding to the previous versions is retained.

- **ACL Security.** Security of accessing Oracle XML DB resources is based on the ACL (Access Control Lists) mechanism. Every resource in Oracle XML DB has an associated ACL that lists its privileges. Whenever resources are accessed or manipulated, these ACLs determine if the operation is legal.
- **Foldering.** XML Repository's foldering module manages a persistent hierarchy of containers, also known as folders or directories, and resources. Other Oracle XML DB modules, such as protocol servers, the schema manager, and the Oracle XML DB RESOURCE_VIEW API, use the foldering module to map path names to resources.

XMLType Storage Architecture

[Figure 1-2](#) describes the XMLType tables and views storage architecture in more detail.

For XMLType tables, tables with XMLType columns, and views, if XML schema-based and the XML schema is registered with Oracle XML DB, XML elements are mapped to database tables. These can be easily viewed and accessed in XML Repository.

Data in XMLType tables and tables containing XMLType columns can be stored in Character Large Objects (CLOBs) or natively in structured XML storage.

Data in XMLType views can be stored in local tables or remote tables that are accessed using DBLinks.

Both XMLType tables and views can be indexed using B*Tree, Oracle Text, function-based, or bitmap indexes.

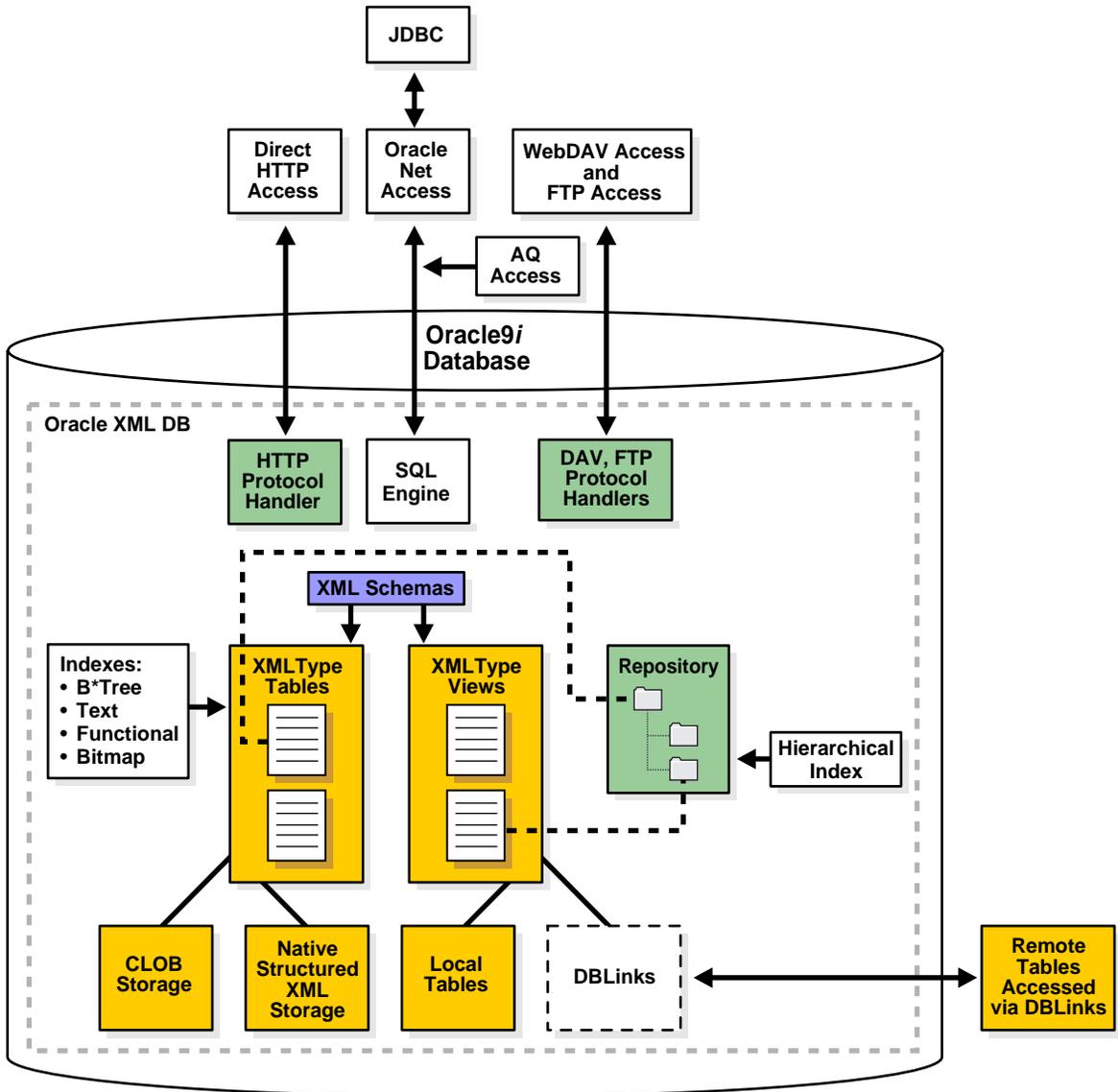
Options for accessing data in XML Repository include:

- HTTP, through the HTTP protocol handler.
- WebDav and FTP, through the WebDav and FTP protocol server.
- SQL, through Oracle Net Services including JDBC. Oracle XML DB also supports XML data messaging using Advanced Queueing (AQ) and SOAP.

See Also:

- [Part II. Storing and Retrieving XML Data in Oracle XML DB](#)
- [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#)
- [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)
- [Chapter 23, "Exchanging XML Data Using Advanced Queueing \(AQ\)"](#)

Figure 1-2 Oracle XML DB: XMLType Storage and Retrieval Architecture



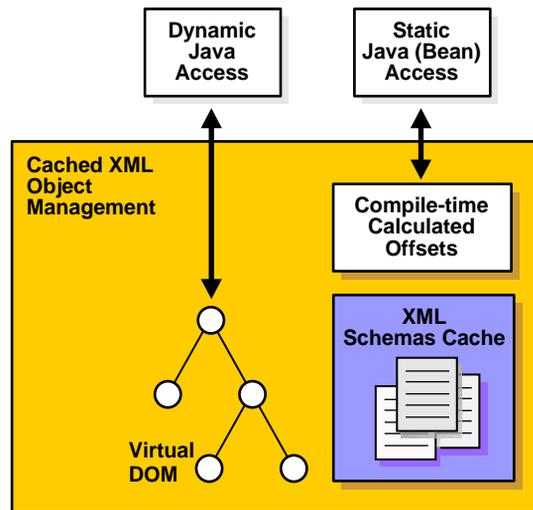
Cached XML Object Management Architecture

Figure 1–3 shows the Oracle XML DB Cached XML Object Management Architecture, relevant for programmatic access to `XMLType` instances. The Oracle XML DB cache can be deployed at the client (with Oracle JDBC OCI driver) or within the server. This cache provides:

- A lazily materialized virtual DOM from the stored `XMLType`, whose nodes are fetched on demand
- A cache for XML schemas

You can thus get dynamic access to XML without having to materialize an entire XML DOM in memory. Static (Java Bean) access is also available. This is accomplished by calculating offsets to the nodes in the DOM during compilation.

Figure 1–3 *Cached XML Object Management Architecture*



XML Repository Architecture

[Figure 1–4](#) describes the Oracle XML DB Repository (XML Repository) architecture.

A *resource* is any piece of content managed by Oracle XML DB, for which we desire to maintain or view the file/folder metaphor.

Each resource has a name, an associated access control list that determines who can see the resource, certain static properties, and some extra ones that are extensible by the application. The application using the Repository obtains a logical view of folders in parent-child arrangement. The Repository is available in the database (for example, for SQL access) using the `RESOURCE_VIEW`.

The `RESOURCE_VIEW` in Oracle9i database consists of a Resource (itself an `XMLType`), that contains the queryable name of the resource, its ACLs, and its properties, static or extensible.

- If the content comprising the resource is XML (stored somewhere in an `XMLType` table or view), the `RESOURCE_VIEW` points to that `XMLType` row that stores the content.
- If the content is not XML, then the `RESOURCE_VIEW` stores it as a LOB.

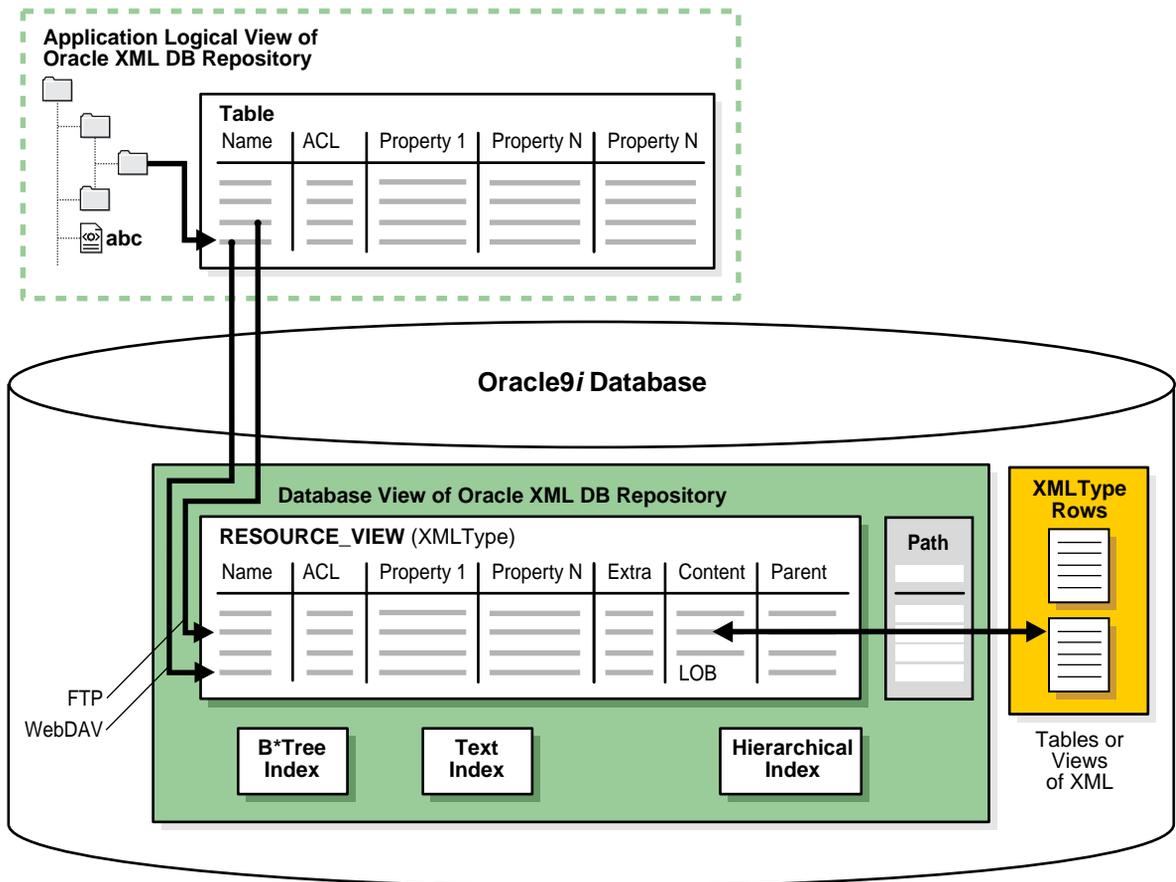
Parent-child relationships between folders (necessary to construct the hierarchy) are maintained and traversed efficiently using the hierarchical index. Text indexes are available to search the properties of a resource, and internal B*Tree indexes over Names and ACLs speed up access to these attributes of the Resource `XMLType`.

In addition to the resource information, the `RESOURCE_VIEW` also contains a Path column, which holds the paths to each resource.

See Also:

- [Chapter 13, "Oracle XML DB Foldering"](#)
- [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)

Figure 1-4 Oracle XML DB: Repository Architecture



Why Use Oracle XML DB?

The following section describes Oracle XML DB advantages for building XML database applications. The main advantages are:

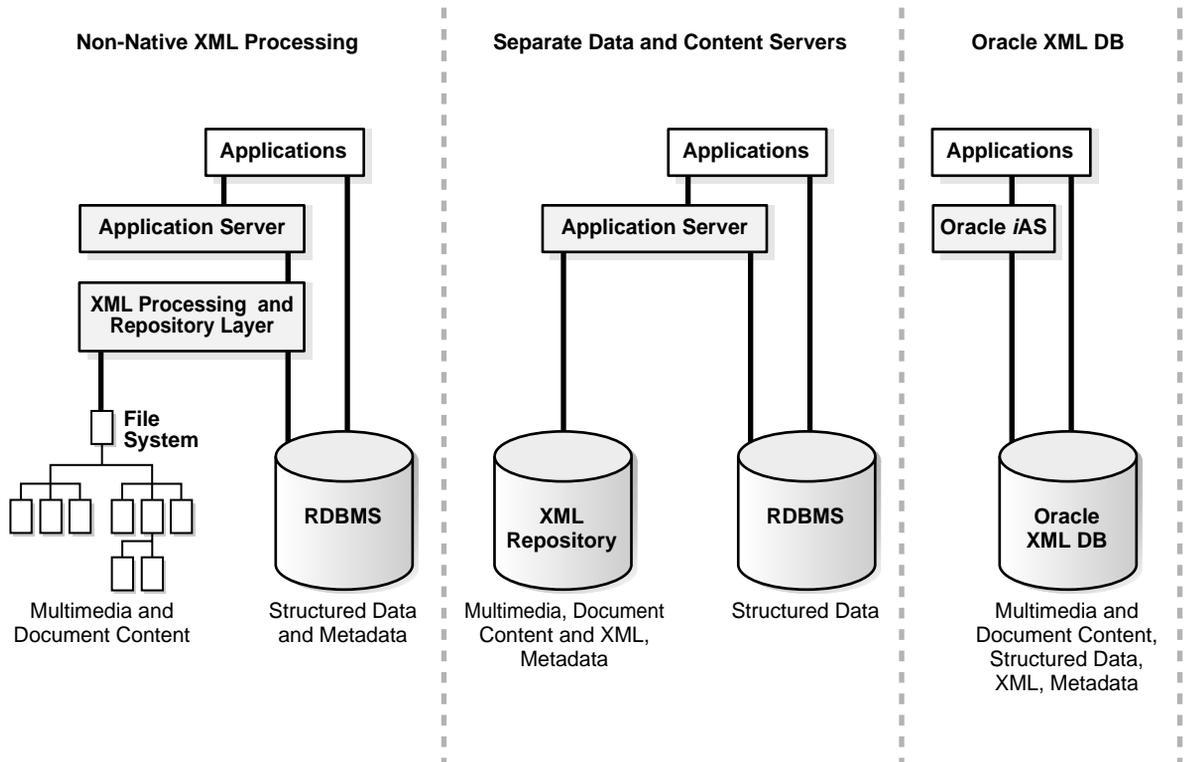
- [Unifying Data and Content with Oracle XML DB](#)
- [Oracle XML DB Offers Faster Storage and Retrieval of Complex XML Documents](#)
- [Oracle XML DB Helps You Integrate Applications](#)

- [When Your Data Is Not XML You Can Use XMLType Views](#)

Unifying Data and Content with Oracle XML DB

Most applications' data and Web content is stored in a relational database or a file system, or a combination of both. XML is used mostly for transport and is generated from a database or a file system. As the volume of XML transported grows, the cost of regenerating these XML documents grows and these storage methods become less effective at accommodating XML content. See [Figure 1-5](#). Oracle XML DB is effective at accommodating XML content. It provides enhanced native support for XML.

Figure 1-5 Unifying Data and Content: Some Common XML Architectures



Organizations today typically manage their structured data and unstructured data differently:

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

With Oracle XML DB you can store and manage both structured, unstructured, and pseudo or semi-structured data, using a standard data model, and standard SQL and XML.

Oracle XML DB provides complete transparency and interchangeability between XML and SQL. You can perform both the following:

- XML operations on object-relational (such as table) data
- SQL operations on XML documents

This makes the database much more accessible to XML-shaped data content.

Exploiting Database Capabilities

In previous releases, without strong database XML support, you most likely stored your XML data in files or in unstructured storage such as CLOBs. Whether you stored your XML data in files or CLOBs, you did not exploit several key capabilities of Oracle database:

- *Indexing and Search:* Applications use queries such as “find all the product definitions created between March and April 2002”, a query that is typically supported by a B*Tree index on a date column. Previously, content management vendors have had to build proprietary query APIs to handle this problem. Oracle XML DB can enable efficient structured searches on XML data. See [Chapter 4, "Using XMLType"](#), [Chapter 10, "Generating XML Data from the Database"](#), and [Chapter 7, "Searching XML Data with Oracle Text"](#).
- *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 [Chapter 4, "Using XMLType"](#).
- *Managing Relationships:* Data with any structure typically has foreign key constraints. Currently, XML data-stores 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 [Chapter 5, "Structured Mapping of XMLType"](#) and the purchase order case study/examples at the end of [Chapter 4, "Using XMLType"](#).

- *Multiple Views of Data:* Most enterprise applications need to group data together in different ways for different modules. This is why relational views are necessary—to allow for these multiple ways to combine data. By allowing views on XML, Oracle XML DB creates different logical abstractions on XML for, say, consumption by different types of applications. See [Chapter 11, "XMLType Views"](#).
- *Performance and Scalability:* Users expect data storage, retrieval, and query to be fast. Loading a file or CLOB and parsing is typically slower than relational data access. Oracle XML DB dramatically speeds up XML storage and retrieval. See [Chapter 2, "Getting Started with Oracle XML DB"](#) and [Chapter 3, "Using Oracle XML DB"](#).
- *Ease of Development:* Databases are foremost an application platform that provides standard, easy ways to manipulate, transform, and modify individual data elements. While typical XML parsers give standard read access to XML data they do not provide an easy way to modify and store individual XML elements. Oracle XML DB supports a number of standard ways to store, modify, and retrieve data: using XML Schema, XPath, DOM, Java Beans, and JNDI.

See Also:

- [Chapter 9, "Java and Java Bean APIs for XMLType"](#)
- [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)
- [Chapter 16, "Oracle XML DB Resource API for PL/SQL \(DBMS_XDB\)"](#)

Exploiting XML Capabilities

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

- *Structure Independence:* The open content model of XML cannot be captured easily in the pure tables-and-columns world. XML Schemas allow global element declarations, not just scoped to a container. Hence you can find a particular data item regardless of where in the XML document it moves to as your application evolves. See [Chapter 5, "Structured Mapping of XMLType"](#).

- *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 allows DBAs to map structured data to physical table and column storage. See [Chapter 5, "Structured Mapping of XMLType"](#) and [Chapter 13, "Oracle XML DB Foldering"](#).
- *Ease of Presentation*: XML is understood natively by browsers, many popular desktop applications, and most internet applications. Relational data is not generally accessible directly from applications, but requires programming to be made accessible to standard clients. Oracle XML DB stores data as XML and pump it out as XML, requiring no programming to display your database content. See:
 - [Chapter 6, "Transforming and Validating XMLType Data"](#).
 - [Chapter 10, "Generating XML Data from the Database"](#).
 - [Chapter 11, "XMLType Views"](#).
 - *Oracle9i XML Developer's Kits Guide - XDK*, in the chapter, "XSQL Pages Publishing Framework". It includes `XMLType` examples.
- *Ease of Interchange*: XML is the language of choice in Business-to-Business (B2B) data exchange. If you are forced to store XML in an arbitrary table structure, you are using some kind of proprietary translation. Whenever you translate a language, information is lost and interchange suffers. By natively understanding XML and providing DOM fidelity in the storage/retrieval process, Oracle XML DB enables a clean interchange. See:
 - [Chapter 6, "Transforming and Validating XMLType Data"](#)
 - [Chapter 11, "XMLType Views"](#)

Oracle XML DB Offers Faster Storage and Retrieval of Complex XML Documents

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

- Native `XMLType`. See [Chapter 4, "Using XMLType"](#).
- The lazily evaluated virtual DOM support. See [Chapter 8, "PL/SQL API for XMLType"](#).

- Database-integrated ad-hoc XPath and XSLT support. This support is described in several chapters, including [Chapter 4, "Using XMLType"](#) and [Chapter 6, "Transforming and Validating XMLType Data"](#).
- XML Schema-caching support. See [Chapter 5, "Structured Mapping of XMLType"](#).
- CTXPath Text indexing. See [Chapter 7, "Searching XML Data with Oracle Text"](#).
- The hierarchical index over the Repository. See [Chapter 13, "Oracle XML DB Foldering"](#).

Oracle XML DB Helps You Integrate Applications

Oracle XML DB enables data from disparate systems to be accessed through gateways and combined into one common data model. This reduces the complexity of developing applications that must deal with data from different stores.

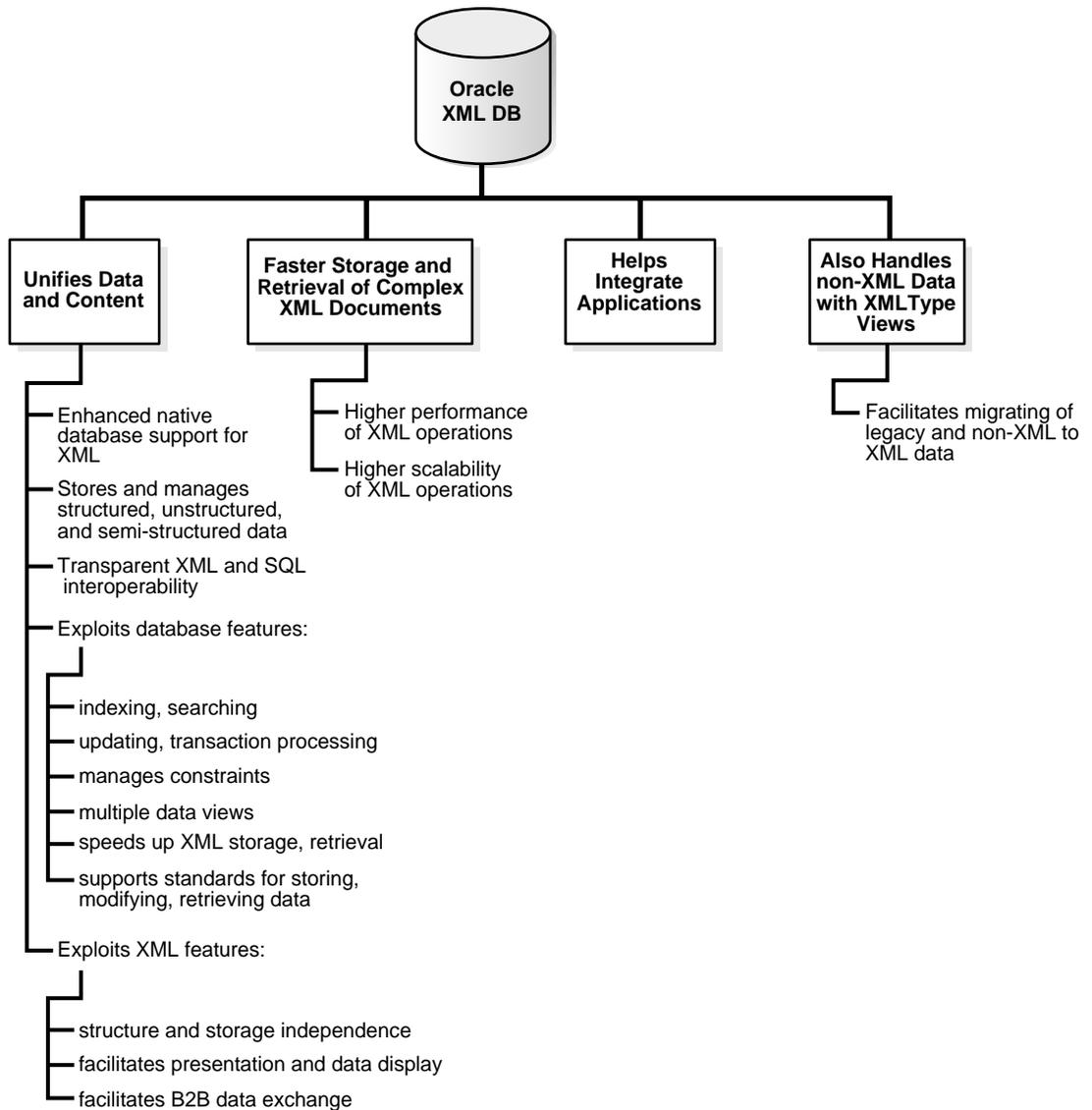
When Your Data Is Not XML You Can Use XMLType Views

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

See Also: [Chapter 11, "XMLType Views"](#).

To use XMLType views you must first register an XML schema with annotations that represent the bi-directional mapping from XML to SQL object types and back to XML. An XMLType view conforming to this schema (mapping) can then be created by providing an underlying query that constructs instances of the appropriate SQL object type. [Figure 1-6](#) summarizes the Oracle XML DB advantages.

Figure 1-6 Oracle XML DB Benefits



Searching XML Data Stored in CLOBs Using Oracle Text

Oracle enables special indexing on XML, including Oracle Text indexes for section searching, special operators to process XML, aggregation of XML, and special optimization of queries involving XML.

XML data stored in Character Large Objects (CLOBs) or stored in `XMLType` columns in structured storage (object-relationally), can be indexed using Oracle Text. `HASPATH()` and `INPATH()` operators are designed to optimize XML data searches where you can search within XML text for substring matches.

Oracle9i Release 2 (9.2) also provides:

- `CONTAINS()` function that can be used with `existsNode()` for XPath based searches. This is for use as the `ora:contains` function in an XPath query, as part of `existsNode()`.
- The ability to create indexes on `UriType` and `XDBUriType` columns.
- A new index type, `CTXXPATH`, that allows higher performance XPath searching in Oracle XML DB under `existsNode()`.

See Also:

- [Chapter 7, "Searching XML Data with Oracle Text"](#)
- *Oracle Text Application Developer's Guide*
- *Oracle Text Reference*

Building Oracle XML DB XML Messaging Applications with Advanced Queuing

Advanced Queuing now supports the use of:

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

See Also:

- *Oracle9i Application Developer's Guide - Advanced Queuing* for information about using `XMLType` with Oracle Advanced Queuing
- [Chapter 23, "Exchanging XML Data Using Advanced Queueing \(AQ\)"](#)

Managing Oracle XML DB Applications with Oracle Enterprise Manager

You can use Oracle Enterprise Manager (Enterprise Manager) to manage and administer your Oracle XML DB application. Enterprise Manager's graphical user interface facilitates your performing the following tasks:

- Configuration
 - Configuring Oracle XML DB, including protocol server configuration
 - Viewing and editing Oracle XML DB configuration parameters
 - Registering XML schema
- Create resources
 - Managing resource security, such as editing resource ACL definitions
 - Granting and revoking resource privileges
 - Creating and editing resource indexes
 - Viewing and navigating your Oracle XML DB hierarchical Repository
- Create XML schema-based tables and views
 - Creating your storage infrastructure based on XML schemas
 - Editing an XML schema
 - Creating an `XMLType` table and a table with `XMLType` columns
 - Creating a view based XML schema
 - Creating a function-based index based on XPath expressions

See Also: [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)

Requirements for Running Oracle XML DB

Oracle XML DB is available with Oracle9i Release 2 (9.2).

See:

- <http://otn.oracle.com/tech/xml> for the latest news and white papers on Oracle XML DB
- [Chapter 2, "Getting Started with Oracle XML DB"](#)

Standards Supported by Oracle XML DB

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

- W3C XML Schema 1.0 Recommendation. You can register XML schemas, validate stored XML content against XML schemas, or constrain XML stored in the server to XML schemas.
- W3C XPath 1.0 Recommendation. You can search or traverse XML stored inside the database using XPath, either from HTTP requests or from SQL.
- ISO-ANSI Working Draft for XML-Related Specifications (SQL/XML) [ISO/IEC 9075 Part 14 and ANSI]. You can use the emerging ANSI SQLX functions to query XML from SQL.
- Java Database Connectivity (JDBC) API. JDBC access to XML is available for Java programmers.
- W3C XSL 1.0 Recommendation. You can transform XML documents at the server using XSLT.
- W3C DOM Recommendation Levels 1.0 and 2.0 Core. You can retrieve XML stored in the server as an XML DOM, for dynamic access.
- Java Beans. You can access XML stored in the server through a Java Bean interface.
- Protocol support. You can store or retrieve XML data from Oracle XML DB using standard protocols such as HTTP, FTP, IETF WebDAV, as well as Oracle Net. See [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#).
- Java Naming and Directory Interface (JNDI). You can use JNDI for hierarchical access to XML resources.
- Java Servlet version 2.2, (except that the Servlet WAR file, `web.xml` is not supported in its entirety, and only one `ServletContext` and one `web-app` are

currently supported, and stateful servlets are not supported). See [Chapter 20, "Writing Oracle XML DB Applications in Java"](#).

- Simple Object Access Protocol (SOAP). You can access XML stored in the server from SOAP requests. You can build, publish, or find Web Services using Oracle XML DB and Oracle9iAS, using WSDL and UDDI. You can use Oracle Advanced Queuing IDAP, the SOAP specification for queuing operations, on XML stored in Oracle9i database. See [Chapter 23, "Exchanging XML Data Using Advanced Queuing \(AQ\)"](#) and *Oracle9i Application Developer's Guide - Advanced Queuing*.

Oracle XML DB Technical Support

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

<http://otn.oracle.com/tech/xml>

You do not need to be a registered user of OTN to post or reply to XML-related questions on the OTN technical discussion forum. To use the OTN technical forum follow these steps:

1. In the left-hand navigation bar of the OTN site, select Support > Discussions.
2. Click Enter a Technical Forum.
3. Scroll down to the Technologies section. Select XML.
4. Post any questions, comments, requests, or bug reports.

Terminology Used in This Manual

[Table 1-2](#) describes terms used in this manual.

See Also: ["Glossary"](#)

Table 1–2 Terminology Used in This Manual

Term Used in Manual	Description
XML Schema	<p>XML Schema is a schema definition language (also in XML) that can be used to describe the structure and various other semantics of conforming instance documents. See Appendix B, "XML Schema Primer".</p> <p>Oracle XML DB uses annotated XML schemas, that is, XML schemas that include additional attributes defined by Oracle XML DB. The Oracle XML DB attributes serve to specify metadata that in turn determines both the XML structuring and its mapping to a database schema. You can register XML schemas and then use the appropriate XML schema URLs while creating <code>XMLType</code> tables and columns and also to define <code>XMLType</code> views. See:</p> <ul style="list-style-type: none"> ■ Chapter 5, "Structured Mapping of XMLType" ■ Appendix B, "XML Schema Primer"
XPath	<p>A language for addressing parts of an XML document, for use by XSLT and XPointer. XPath uses the directory traversal syntax to traverse an XML document. It includes syntax for specifying predicate expressions on the nodes traversed. The result of a XPath traversal is an XML fragment. See Appendix C, "XPath and Namespace Primer".</p>
XSL	<p>A stylesheet language used for transforming XML documents to HTML, XML or any other formats. See Appendix D, "XSLT Primer".</p>
DOM	<p>Document Object Model (DOM) is an application program interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term “document” is used in the broad sense.</p> <p>XML is increasingly being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally have been seen as data rather than as documents. Nevertheless, XML presents this data as documents, and DOM can be used to manage this data.</p> <p>With DOM, you can build documents, navigate their structure, and add, modify, or delete elements and content. Anything in an HTML or XML document can be accessed, changed, deleted, or added using DOM, with a few exceptions. DOM is designed for use with any programming language.</p> <p>Oracle XML DB provides implementations of DOM APIs to operate on <code>XMLType</code> instances using various client APIs including PL/SQL DOM, Java DOM, and C DOM (for OCI clients). See</p> <ul style="list-style-type: none"> ■ Chapter 5, "Structured Mapping of XMLType" ■ Chapter 8, "PL/SQL API for XMLType"
Oracle XML DB Repository	See also Chapter 3, "Using Oracle XML DB"

Table 1–2 Terminology Used in This Manual (Cont.)

Term Used in Manual	Description
Resource	An object identified by a URL. In compliance with HTTP and WebDAV standards, it has a set of system properties, such as <code>displayname</code> , <code>creationdate</code> , and so on. In all cases, it maintains a reference count and destroys any associated data when the last URL binding to it is removed. It maintains an access control list (ACL) and owner. An Oracle XML DB resource is an <code>XMLType</code> mapped to a path name that contains these properties. See Chapter 15, "RESOURCE_VIEW and PATH_VIEW" .
Repository	<p>Oracle XML DB Repository is the set of all Oracle XML DB resources. The Repository is a hierarchically organized set of <code>XMLType</code> objects, each with a path name to identify them. Think of the Oracle XML DB Repository as a file system of objects rather than files. There is one root to this Repository (“/”), which contains a set of resources, each with a path name. Resources that contain (“contain” with respect to the hierarchical naming system) other resources are called <i>folders</i> (see “Folder” in the following).</p> <p>Oracle XML DB objects can have many path names (that is, a resource can be in more than one folder). In some sense, the database itself is the Repository, since any database object can be mapped to a path name. However, Oracle XML DB uses “Repository” to refer to the set of database objects, in any schema, that are mapped to path names. See Chapter 13, "Oracle XML DB Foldering".</p>
Folder	A non-leaf node object in Oracle XML DB Repository, or one with the potential to be such a node. Oracle XML DB has special storage semantics for collections for optimization reasons. It maintains a special kind of hierarchical index used to navigate the hierarchy of collections, and defines a property, called <i>name</i> that is used to form path names in the hierarchy. There are many names for collections, such as <i>folders</i> and <i>directories</i> . Any XML element type can be a folder by specifying the <code>isFolder</code> attribute in the Oracle XML DB schema. See Chapter 13, "Oracle XML DB Foldering" .
Pathname	A hierarchical name is composed of a root element (the first /), element separators (/), and various sub-elements (or path elements). A path element can be composed of any character in the database character set except the following (‘\’ ‘/’). In Oracle XML DB, a forward slash is the default name separator in a path name.
Resource Name	A resource here means any database object stored in Oracle XML DB Repository. Resource name is the name of a resource within its parent folder. Resource names are the path elements, that is, filenames within folders. Resource names must be unique (potentially subject to case-insensitivity) within a folder.
Content	The body of a resource is what you get when you treat the resource like a file and ask for its contents.
XDBBinary	An XML element defined by the Oracle XML DB schema that contains binary data. XDBBinary elements are stored in the Repository when completely unstructured binary data is uploaded into Oracle XML DB.

Table 1–2 Terminology Used in This Manual (Cont.)

Term Used in Manual	Description
ACL Terminology	See also Chapter 18, "Oracle XML DB Resource Security"
Access Control List (ACL)	Restricts access to an object. Oracle XML DB uses ACLs to restrict access to any Oracle XML DB resource, that is, any XMLType object that is mapped into the Oracle XML DB file system hierarchy.
Protocol Terminology	See also Chapter 19, "Using FTP, HTTP, and WebDAV Protocols" and Chapter 3, "Using Oracle XML DB"
FTP	"File Transfer Protocol". Defined as an Internet Standard (STD009) in RFC959. Oracle XML DB implements this standard. FTP is implemented by both dedicated clients at the operating system level, file system explorer clients, and browsers. FTP is commonly used for bulk file upload and download and for scripting of Repository maintenance. FTP can be used in a mode similar to HTTP, with frequent session establishment/destruction, by browsers in "passive" mode.
HTTP	"HyperText Transfer Protocol". Oracle XML DB implements HTTP 1.1 as defined in RFC2616. Oracle XML DB implements cookies, basic authentication, and HTTP/1.1 (RFC2616, 2109 & 2965) in this release.
WebDAV	Web Distributed Authoring and Versioning (WebDav). Oracle XML DB supports RFC2518 and access control in this release.
Servlets	<p>Sun developed a widely accepted standard for invoking Java code as the result of protocol requests and passing parameters to that request. Servlets are most commonly implemented with HTTP. The majority of Java services are implemented as servlets, through mechanisms (implemented in Java) such as JSPs (Java Server Pages) or SOAP (Simple Object Access Protocol). Servlets thus form the architectural basis for a large percentage of web application development.</p> <p>Oracle XML DB provides a method for invoking Java stored procedures over protocols other than Oracle Services (Net Services). Oracle XML DB implements most servlet standards. Chapter 20, "Writing Oracle XML DB Applications in Java".</p>

Oracle XML DB Examples Used in This Manual

This manual contains examples that illustrate the use of Oracle XML DB and XMLType. The examples are based on a number of database schema, sample XML documents, and sample XML schema. The infrastructure for the examples is described, in most cases, with the examples in each chapter. The examples provided have all been tested.

See Also: [Appendix G, "Example Setup scripts. Oracle XML DB-Supplied XML Schemas"](#)

Note: In the SQL clause for creating `XMLType` tables or columns, you no longer need to use the whole term, `sys.XMLType.createXML()`. You can use just, `XMLType`.

- For example, in the previous release, you used the following:

```
... sys.XMLType.createXML(' <Warehouse  
whNo="100">...)
```
- In Release 2 (9.2), you can use the following abbreviated version:

```
...XMLType(' <Warehouse whNo="100">...),
```

or you can use the long version shown in the first bullet.

Similarly, in the `CREATE TABLE` statement used in the previous release, you used:

```
CREATE TABLE warehouses(warehouse_id NUMBER(3),  
warehouse_spec SYS.XMLType,...)
```

You can now use “`XMLType`” without the “`SYS`” prefix, as follows:

```
CREATE TABLE warehouses(warehouse_id NUMBER(3),  
warehouse_spec XMLType,...)
```

Getting Started with Oracle XML DB

This chapter provides some preliminary design criteria for consideration when planning your Oracle XML DB solution. It contains the following sections:

- [Getting Started with Oracle XML DB](#)
- [When to Use the Oracle XML DB](#)
- [Designing Your XML Application](#)
- [Oracle XML DB Design Issues: Introduction](#)
- [Oracle XML DB Application Design: a. How Structured Is Your Data?](#)
- [Oracle XML DB Application Design: b. Access Models](#)
- [Oracle XML DB Application Design: c. Application Language](#)
- [Oracle XML DB Application Design: d. Processing Models](#)
- [Oracle XML DB Design: Storage Models](#)

Getting Started with Oracle XML DB

Installing Oracle XML DB

Oracle XML DB is installed as part of the General Purpose Database shipping with Oracle9i Release 2 (9.2) database. If you need to perform a manual installation or de-installation of the Oracle XML DB, see [Appendix A, "Installing and Configuring Oracle XML DB"](#) for further information.

When to Use the Oracle XML DB

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

The type of applications that Oracle XML DB is particularly suited to include:

- Business-to-Business (B2B) and Application-to-Application (A2A) integration
- Internet applications
- Content-management applications
- Messaging
- Web Services

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

- Large numbers of XML documents must be ingested or generated
- Large XML documents need to be processed or generated
- High performance searching, both within a document and across a large collections of documents
- High Levels of security. Fine grained control of security
- Data processing must be contained in XML documents and data contained in traditional relational tables
- Uses languages such as Java that support open standards such as SQL, XML, XPath, and XSLT

- Accesses information using standard Internet protocols such as FTP, HTTP/WebDav, or JDBC
- Full queriability from SQL and integration with analytic capabilities
- Validation of XML documents is critical

Designing Your XML Application

Oracle XML DB provides you with the ability to fine tune how XML documents will be stored and processed in Oracle9i database. Depending on the nature of the application being developed, XML storage must have at least one of the following features

- High performance ingestion and retrieval of XML documents
- High performance indexing and searching of XML documents
- Be able to update sections of an XML document
- Manage highly either or both structured and non-structured XML documents

Oracle XML DB Design Issues: Introduction

This section discusses the preliminary design criteria you can consider when planning your Oracle XML DB application. [Figure 2-1](#) provides an overview of your main design options for building Oracle XML DB applications.

a. Data

Will your data be highly structured (mostly XML), semi- structured (pseudo-structured), or mostly non-structured? If highly structured, will your table(s) be XML schema-based or non-schema-based? See "[Oracle XML DB Application Design: a. How Structured Is Your Data?](#)" on page 2-5 and [Chapter 3, "Using Oracle XML DB"](#).

b. Access

How will other applications and users access your XML and other data? How secure must the access be? Do you need versioning? See "[Oracle XML DB Application Design: b. Access Models](#)" on page 2-7.

c. Application Language

In which language(s) will you be programming your application? See ["Oracle XML DB Application Design: c. Application Language"](#) on page 2-8.

d. Processing

Will you need to generate XML? See [Chapter 10, "Generating XML Data from the Database"](#).

How often will XML documents be accessed, updated, and manipulated? Will you need to update fragments or the whole document?

Will you need to transform the XML to HTML, WML, or other languages, and how will your application transform the XML? See [Chapter 6, "Transforming and Validating XMLType Data"](#).

Does your application need to be primarily database resident or work in both database and middle tier?

Is your application data-centric, document- and content-centric, or *integrated* (is both data- and document-centric). See ["Oracle XML DB Application Design: d. Processing Models"](#) on page 2-9.

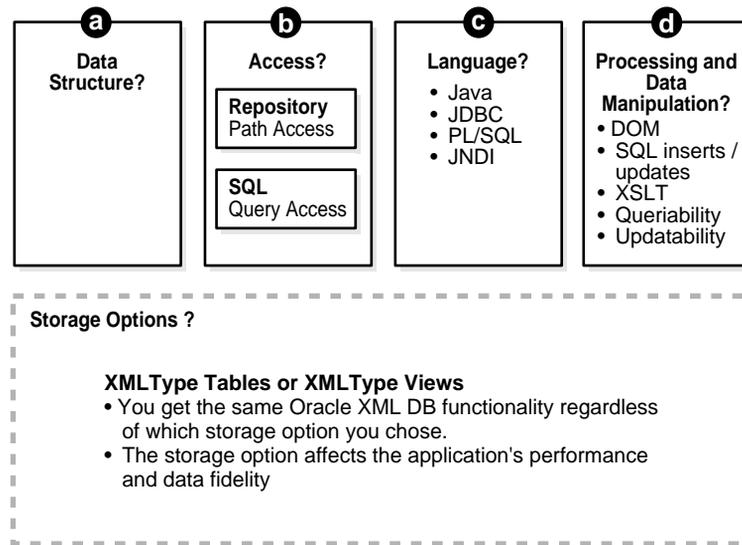
Will you be exchanging XML data with other applications, across gateways? Will you need Advanced Queueing (AQ) or SOAP compliance? See [Chapter 23, "Exchanging XML Data Using Advanced Queueing \(AQ\)"](#).

Storage

How and where will you store the data, XML data, XML schema, and so on? See ["Oracle XML DB Design: Storage Models"](#) on page 2-10.

Note: Your choice of which models to choose in the preceding four categories, a through d, are typically related to each other. However, the storage model you choose is *orthogonal* to the choices you make for the other design models. In other words, choices you make for the other design modeling options are not dependent on the storage model option you choose.

Figure 2–1 Oracle XML DB Design Options



Oracle XML DB Application Design: a. How Structured Is Your Data?

Figure 2–2 shows the following data structure categories and associated suggested storage options:

- **Structured data.** Is your data highly structured? In other words, is your data mostly XML data?
- **Semi/pseudo-structured data.** Is your data semi/pseudo-structured? In other words does your data include some XML data?
- **Unstructured data.** Is your data unstructured? In other words, is your data mostly non-XML data?

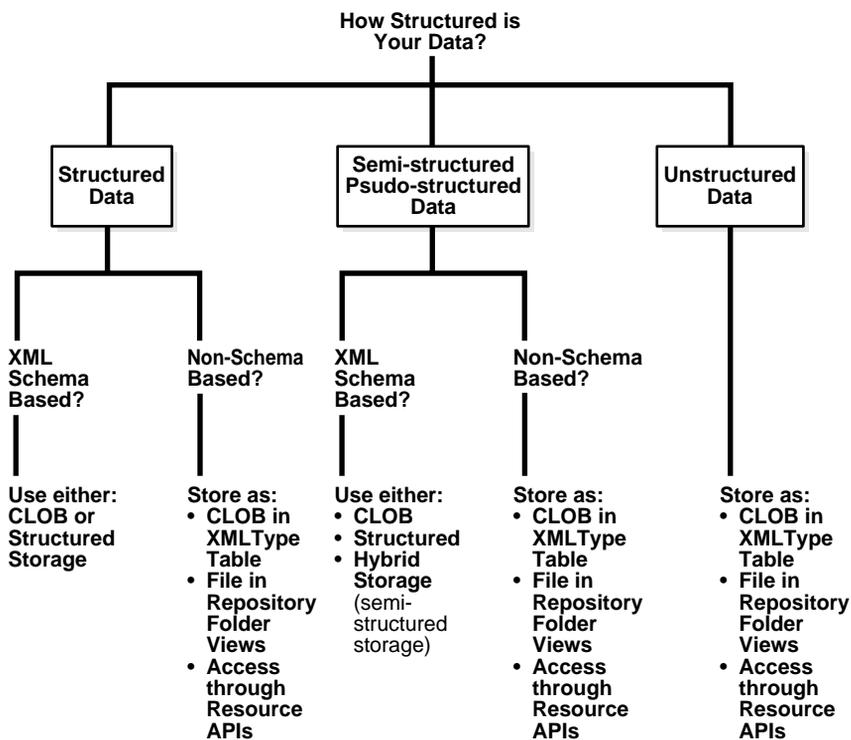
XML Schema-Based or Non-Schema-Based

Also consider the following data modeling questions:

- If your application is XML schema-based:
 - For structured data, you can use either Character Large Object (CLOB) or structured storage.

- For semi- or pseudo-structured data, you can use either CLOB, structured, or hybrid storage. Here your XML schema can be more loosely coupled. See also "[Oracle XML DB Design: Storage Models](#)" on page 2-10.
- For unstructured data, an XML schema design is not applicable.
- If your application is non-schema-based. For structured, semi/pseudo-structured, and unstructured data, you can store your data in either CLOBs in XMLType tables or views or in files in Repository folders. With this design you have many access options including path- and query-based access through Resource Views.

Figure 2-2 Data Storage Models: How Structured Is Your Data?



Oracle XML DB Application Design: b. Access Models

Figure 2-3 shows the two main data access modes to consider when designing your Oracle XML DB applications:

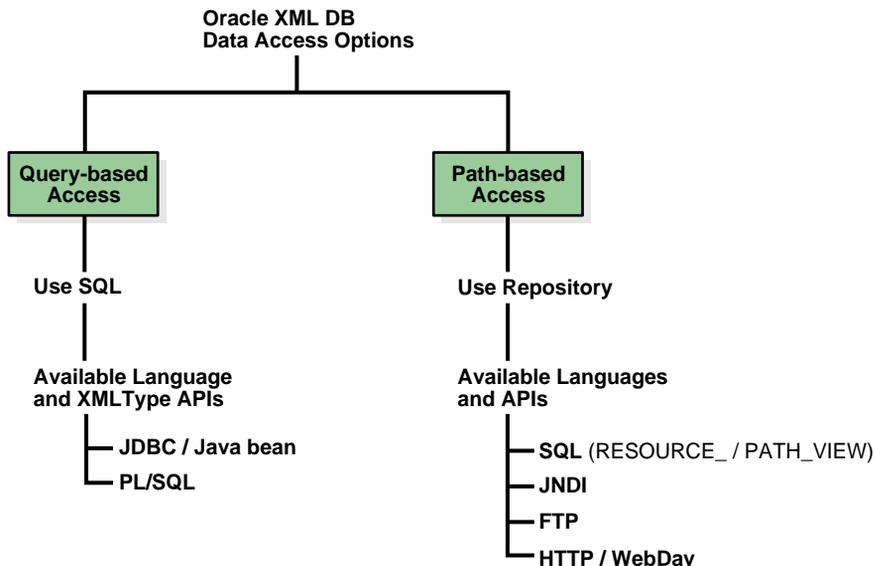
- *Navigation- or path-based access.* This is suitable for both content/document and data oriented applications. Oracle XML DB provides the following languages and access APIs:
 - SQL access through Resource/Path Views. See [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#).
 - PL/SQL access through `DBMS_XDB`. See [Chapter 16, "Oracle XML DB Resource API for PL/SQL \(DBMS_XDB\)"](#).
 - Java, JNDI access. See [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#).
 - Protocol-based access using HTTP/WebDAV or FTP, most suited to content-oriented applications. See [Chapter 19, "Using FTP, HTTP, and WebDAV 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 (through JDBC) access. See [Chapter 9, "Java and Java Bean APIs for XMLType"](#).
 - PL/SQL access. See [Chapter 8, "PL/SQL API for XMLType"](#).

These options for accessing Repository data are also discussed in [Chapter 13, "Oracle XML DB Foldering"](#).

You can also consider the following access model criteria:

- What level of security do you need? See [Chapter 18, "Oracle XML DB Resource Security"](#).
- What kind of indexing will best suit your application? Will you need to use Oracle Text indexing and querying? See [Chapter 4, "Using XMLType"](#) and [Chapter 7, "Searching XML Data with Oracle Text"](#).
- Do you need to version the data? If yes, see [Chapter 14, "Oracle XML DB Versioning"](#).

Figure 2–3 Data Access Models: How Will Users or Applications Access the Data?



Oracle XML DB Application Design: c. Application Language

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

- Java (JDBC, Java Servlets, Java Beans, JNDI)

See Also:

- [Chapter 9, "Java and Java Bean APIs for XMLType"](#)
 - [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#)
 - [Chapter 20, "Writing Oracle XML DB Applications in Java"](#)
 - [Appendix E, "Java DOM and Java Bean API for XMLType, Resource API for Java/JNDI: Quick Reference"](#)
- PLSQL

See Also:

- [Chapter 8, "PL/SQL API for XMLType"](#)
- [Chapter 16, "Oracle XML DB Resource API for PL/SQL \(DBMS_XDB\)"](#)
- [Appendix F, "Oracle XML DB XMLType API, PL/SQL and Resource PL/SQL APIs: Quick Reference"](#)

Oracle XML DB Application Design: d. Processing Models

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

- XSLT. Will you need to transform the XML to HTML, WML, or other languages, and how will your application transform the XML? While storing XML documents in Oracle XML DB you can optionally ensure that their structure complies (is "valid" against) with specific XML Schema. See [Chapter 6, "Transforming and Validating XMLType Data"](#).
- DOM. See [Chapter 8, "PL/SQL API for XMLType"](#). Use object-relational columns, VARRAYs, nested tables, as well as LOBs to store any element or Element-subtree in your XML Schema, and still maintain DOM fidelity (DOM stored == DOM retrieved). Note: If you choose the CLOB storage option, available with XMLType since Oracle9i Release 1 (9.0.1), you can keep whitespaces. If you are using XML schema, see the discussion on DOM fidelity in [Chapter 5, "Structured Mapping of XMLType"](#).
- XPath searching. You can use XPath syntax embedded in an SQL statement or as part of an HTTP request to query XML content in the database. See [Chapter 4, "Using XMLType"](#), [Chapter 7, "Searching XML Data with Oracle Text"](#), [Chapter 13, "Oracle XML DB Foldering"](#), and [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#).
- XML Generation and XMLType views. Will you need to generate or regenerate XML? If yes, see [Chapter 10, "Generating XML Data from the Database"](#).

How often will XML documents be accessed, updated, and manipulated? See [Chapter 4, "Using XMLType"](#) and [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#).

Will you need to update fragments or the whole document? You can use XPath to specify individual elements and attributes of your document during updates,

without rewriting the entire document. This is more efficient, especially for large XML documents. [Chapter 5, "Structured Mapping of XMLType"](#).

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

Messaging Options

Advanced Queueing (AQ) supports XML and XMLType applications. You can create queues with payloads that contain XMLType attributes. These can be used for transmitting and storing messages that contain XML documents. By defining Oracle 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 CLOBs.
- Selectively dequeue messages with XMLType attributes using the operators `existsNode()`, `extract()`, and so on.
- Define transformations to convert Oracle objects to XMLType.
- Define rule-based subscribers that query message content using XMLType operators such as `existsNode()` and `extract()`.

See Also:

- [Chapter 23, "Exchanging XML Data Using Advanced Queueing \(AQ\)"](#)
- *Oracle9i Application Developer's Guide - Advanced Queueing*

Oracle XML DB Design: Storage Models

[Figure 2–4](#) summarizes the Oracle XML DB storage options with regards to using XMLType tables or views. If you have existing or legacy relational data, use XMLType Views.

Regardless of which storage options you choose for your Oracle XML DB application, Oracle XML DB provides the same functionality. However, the option you choose will affect your application's performance and the data fidelity (data accuracy).

Currently, the three main storage options for Oracle XML DB applications are:

- **LOB-based storage?** LOB-based storage assures complete textual fidelity including whitespaces. This means that if you store your XML documents as

CLOBs, when the XML documents are retrieved there will be no data loss. Data integrity is high, and the cost of regeneration is low.

- **Structured storage?** Structured storage loses whitespace information but maintains fidelity to the XML DOM, namely DOM stored = DOM retrieved. This provides:
 - Better SQL 'queriability' with improved performance
 - Piece-wise updatability
- **Hybrid or semi-structured storage.** Hybrid storage is a special case of structured storage in which a portion of the XML data is broken up into a structured format and the remainder of the data is stored as a CLOB.

The storage options are totally independent of the following criteria:

- Data queryability and updatability, namely, how and how often the data is queried and updated.
- How your data is accessed. This is determined by your application processing requirements.
- What language(s) your application uses. This is also determined by your application processing requirements.

See Also:

- ["Storing XML: Structured or Unstructured Storage"](#), ["Structured Storage: XML Schema-Based Storage of XMLType"](#) and ["Storage Options for Resources"](#) in Chapter 3, ["Using Oracle XML DB"](#)
- [Chapter 4, "Using XMLType"](#), ["Storing XMLType Data in Oracle XML DB"](#) on page 4-4
- [Chapter 5, "Structured Mapping of XMLType"](#), ["DOM Fidelity"](#) on page 5-21

Using XMLType Tables

If you are using XMLType tables you can store your data in:

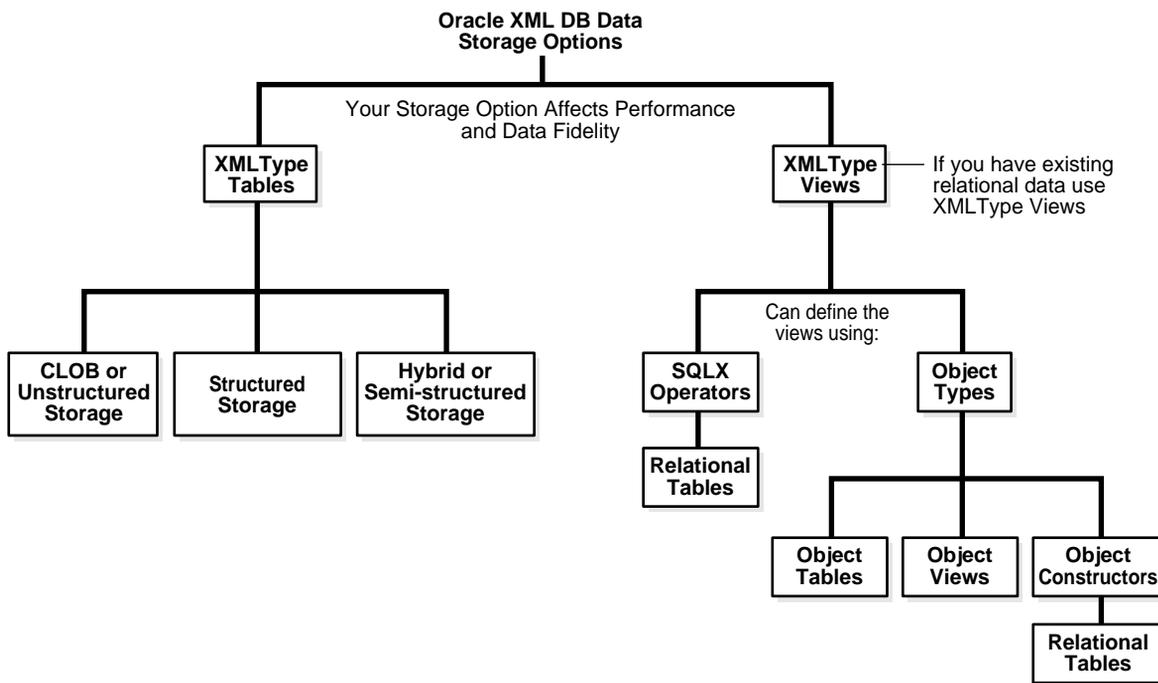
- CLOBs (unstructured) storage
- Structured storage
- Hybrid or semi-structured storage

Using XMLType Views

Use XMLType views if you have existing relational data. You can use the following options to define the XMLType views:

- SQLX operators. Using these operators you can store the data in relational tables and also generate/regenerate the XML . See [Chapter 10, "Generating XML Data from the Database"](#).
- Object Types:
 - Object tables
 - Object constructors. You can store the data in relational tables using object constructors.
 - Object views

Figure 2-4 Structured Storage Options



Using Oracle XML DB

This chapter describes where and how you can use Oracle XML DB. It discusses and includes examples on common Oracle XML DB usage scenarios including `XMLType` data storage and access, updating and validating your data, and why it helps to understand XPath and XML Schema. It provides you with ideas for how you can use the Repository to store, access, and manipulate database data using standard protocols from a variety of clients.

The chapter also discusses how you can define a default XML table for storing XML schema-based documents and using `XDBUriType` to access non-schema-based content.

It contains the following sections:

- [Storing Data in an XMLType Column or XMLType Table](#)
- [Accessing Data in XMLType Columns or XMLType Tables](#)
- [Using XPath with Oracle XML DB](#)
- [Updating XML Documents with `updateXML\(\)`](#)
- [Introducing the W3C XSLT Recommendation](#)
- [Using XSL/XSLT with Oracle XML DB](#)
- [Other XMLType Methods](#)
- [Introducing the W3C XML Schema Recommendation](#)
- [Validating an XML Document Using an XML Schema](#)
- [Storing XML: Structured or Unstructured Storage](#)
- [Structured Storage: XML Schema-Based Storage of XMLType](#)
- [Oracle XML DB Repository](#)

-
- [Query-Based Access to Oracle XML DB Repository](#)
 - [Storage Options for Resources](#)
 - [Defining Your Own Default Table Storage for XML Schema-Based Documents](#)
 - [Accessing XML Schema-Based Content](#)
 - [Accessing Non-Schema-Based Content With XDBUriType](#)
 - [Oracle XML DB Protocol Servers](#)

Storing Data in an XMLType Column or XMLType Table

When storing XML documents in Oracle9i database you can use a number of approaches, including:

- Parsing the XML document apart, outside Oracle9i database, and storing the data in the XML document as rows in one or more tables. In this scenario the database has no idea that is managing XML content.
- Storing the XML document in Oracle9i database using a CLOB or VARCHAR column. Again in this scenario the database has no idea that it is managing XML content, but you can programmatically use the XDK to perform XML operations.
- Storing the XML document in Oracle9i database using the XMLType datatype. Two options are available in this scenario.
 - The first is to store the XML document in an XMLType column.
 - The second is to store the XML document using an XMLType table.

Both these options mean that the database is aware that it is managing XML content. Selecting this approach provides you with a number of significant advantages, as the database provides a set of features that make it possible to process XML content efficiently.

Example 3–1 Creating a Table with an XMLType Column

```
CREATE TABLE Example1
(
  KEYVALUE varchar2(10) primary key,
  XMLCOLUMN xmltype
);
```

Example 3–2 Example 2: Creating a Table of XMLType

```
CREATE TABLE XMLTABLE OF XMLType;
```

Example 3–3 To Store an XML Document First Create an XMLType Instance Using getDocument()

To store an XML document in an XMLType table or column the XML document must first be converted into an XMLType instance. This is done using the different constructors provided by the XMLType datatype. For example, given a PL/SQL function called `getDocument()` with the following signature:

```
FUNCTION getDocument RETURNS CLOB
```

This returns the following:

Argument Name	Type	In/Out	Default?
-----	-----	----	-----
FILENAME	VARCHAR2	IN	

```

create or replace function getDocument(filename varchar2) return clob
  authid current_user is
  xbfile bfile;
  xclob clob;
begin
  xbfile := bfilename('XMLDIR',filename);
  dbms_lob.open(xbfile);

  dbms_lob.createtemporary(xclob,TRUE,dbms_lob.session);
  dbms_lob.loadfromfile(xclob,xbfile, dbms_lob.getlength(xbfile));
  dbms_lob.close(xbfile);
  return xclob;
end;
/

-- create XMLDIR directory
-- connect system/manager
-- create directory XMLDIR as '<location_of_xmlfiles_on_server>';
-- grant read on directory xmldir to public with grant option;

-- you can use getDocument() to generate a CLOB from a file containing an XML
-- document. For example, the following statement inserts a row into the
-- XMLType table Example2 created earlier:

INSERT INTO XMLTABLE
VALUES (XMLTYPE(getDocument('purchaseorder.xml')));

```

Accessing Data in XMLType Columns or XMLType Tables

Once a collection of XML documents have been stored as XMLType tables or columns the next step is to be able to retrieve them. When working with a collection of XML documents you have two fundamental tasks to perform:

- Decide how to select a subset of the available documents
- Determine how best to access some subset of the nodes contained within the documents

Oracle9i database and `XMLType` datatype provide a number of functions that make it easy to perform these tasks. These functions make use of the W3C XPath recommendation to navigate across and within a collection of XML documents.

See Also: [Appendix C, "XPath and Namespace Primer"](#) for an introduction to the W3C XPath Recommendation.

Using XPath with Oracle XML DB

A number of the functions provided by the Oracle XML DB are based on the W3C XPath recommendation. XPath traverses nested XML elements by your specifying the elements to navigate through with a slash-separated list of element and attribute names. By using XPath to define queries within and across XML documents. With Oracle XML DB you can express hierarchical queries against XML documents in a familiar, standards compliant manner.

The primary use of XPath in Oracle XML DB is in conjunction with the `extract()`, `extractValue()`, and `existsNode()` functions.

The `existsNode()` function evaluates whether or not a given document contains a node which matches a W3C XPath expression. The `existsNode()` function returns true (1) if the document contains the node specified by the XPath expression supplied to the function. The functionality provided by the `existsNode()` function is also available through the `XMLType` datatype `existNode()` method.

See Also:

- [Chapter 4, "Using XMLType"](#)
- [Chapter 10, "Generating XML Data from the Database"](#)

PurchaseOrder XML Document

Examples in this section are based on the following PurchaseOrder XML document:

```
<PurchaseOrder
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.oracle.com/xdb/po.xsd">
  <Reference>ADAMS-20011127121040988PST</Reference>
  <Actions>
    <Action>
      <User>SCOTT</User>
      <Date>2002-03-31</Date>
    </Action>
```

```

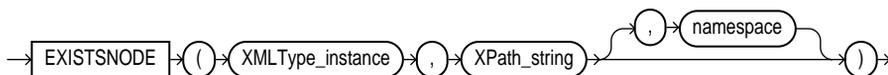
</Actions>
<Reject/>
<Requestor>Julie P. Adams</Requestor>
<User>ADAMS</User>
<CostCenter>R20</CostCenter>
<ShippingInstructions>
  <name>Julie P. Adams</name>
  <address>Redwood Shores, CA 94065</address>
  <telephone>650 506 7300</telephone>
</ShippingInstructions>
<SpecialInstructions>Ground</SpecialInstructions>
<LineItems>
  <LineItem ItemNumber="1">
    <Description>The Ruling Class</Description>
    <Part Id="715515012423" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>Diabolique</Description>
    <Part Id="037429135020" UnitPrice="29.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>8 1/2</Description>
    <Part Id="037429135624" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
</LineItems>
</PurchaseOrder>

```

Using existsNode()

The `existsNode()` syntax is shown in [Figure 3-1](#).

Figure 3-1 `existsNode()` Syntax



Example 3-4 `existsNode()` Examples That Find a Node to Match the XPath Expression

Given this sample XML document, the following `existsNode()` operators return true (1).

```

SELECT existsNode(value(X), '/PurchaseOrder/Reference')
FROM XMLTABLE X;

```

```

SELECT existsNode(value(X),
  '/PurchaseOrder[Reference="ADAMS-20011127121040988PST"]')
  FROM XMLTABLE X;

SELECT existsNode(value(X),
  '/PurchaseOrder/LineItems/LineItem[2]/Part[@Id="037429135020"]')
  FROM XMLTABLE X;

SELECT existsNode(value(X),
  '/PurchaseOrder/LineItems/LineItem[Description="8 1/2"]')
  FROM XMLTABLE X;

```

Example 3-5 existsNode() Examples That Do Not Find a Node that Matches the XPath Expression

The following `existsNode()` operations do not find a node that matches the XPath expression and all return `false(0)`:

```

SELECT existsNode(value(X), '/PurchaseOrder/UserName')
  FROM XMLTABLE X;

SELECT existsNode(value(X),
  '/PurchaseOrder[Reference="ADAMS-XXXXXXXXXXXXXXXXXXXXX"]')
  FROM XMLTABLE X;

SELECT existsNode(value(X),
  '/PurchaseOrder/LineItems/LineItem[3]/Part[@Id="037429135020"]')
  FROM XMLTABLE X;

SELECT existsNode(value(X),
  '/PurchaseOrder/LineItems/LineItem[Description="Snow White"]')
  FROM XMLTABLE X;

```

The most common use for `existsNode()` is in the `WHERE` clause of SQL `SELECT`, `UPDATE`, or `DELETE` statements. In this situation the XPath expression passed to the `existsNode()` function is used to determine which of the XML documents stored in the table will be processed by the SQL statement.

Example 3-6 Using existsNode() in the WHERE Clause

```

SELECT count(*)
  FROM XMLTABLE x
 WHERE existsNode(value(x), '/PurchaseOrder[User="ADAMS"]') = 1;

```

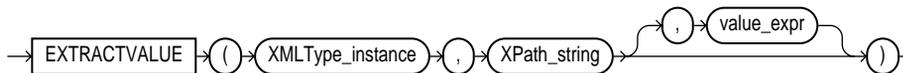
```
DELETE FROM XMLTABLE x
  WHERE existsNode(value(x), '/PurchaseOrder[User="ADAMS"]') = 1;
commit;
```

The `extractValue()` function is used to return the value of a text node or attribute associated with an XPath Expression from an XML document stored as an `XMLType`. It returns a scalar data type.

Using extractValue()

The `extractValue()` syntax is shown in [Figure 3-2](#).

Figure 3-2 *extractValue() Syntax*



The following are examples of `extractValue()`:

Example 3-7 *Valid Uses of extractValue()*

```
SELECT extractValue(value(x), '/PurchaseOrder/Reference')
  FROM XMLTABLE X;
```

Returns the following:

```
EXTRACTVALUE(VALUE(X), '/PURCHASEORDER/REFERENCE')
```

```
-----
ADAMS-20011127121040988PST
```

```
SELECT extractValue(value(x),
  '/PurchaseOrder/LineItems/LineItem[2]/Part/@Id')
  FROM XMLTABLE X;
```

Returns the following:

```
EXTRACTVALUE(VALUE(X), '/PURCHASEORDER/LINEITEMS/LINEITEM[2]/PART/@ID')
```

```
-----
037429135020
```

`extractValue()` can only return the value of a single node or attribute value. For instance the following example shows an invalid use of `extractValue()`. In the first example the XPath expression matches three nodes in the document, in the

second example the XPath expression identifies a nodetree, not a text node or attribute value.

Example 3-8 Non-Valid Uses of extractValue()

```
SELECT extractValue(value(X),
                    '/PurchaseOrder/LineItems/LineItem/Description')
FROM XMLTABLE X;

-- FROM XMLTABLE X;
--      *
-- ERROR at line 3:
-- ORA-19025: EXTRACTVALUE returns value of only one node

SELECT extractValue(value(X),
                    '/PurchaseOrder/LineItems/LineItem[1]')
FROM XMLTABLE X;

-- FROM XMLTABLE X
--      *
-- ERROR at line 3:
-- ORA-19025: EXTRACTVALUE returns value of only one node
```

Example 3-9 Using extractValue() in the WHERE Clause

extractValue() can also be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. This makes it possible to perform joins between XMLType tables or tables containing XMLType columns and other relational tables or XMLType tables. The following query shows you how to use extractValue() in both the SELECT list and the WHERE clause:

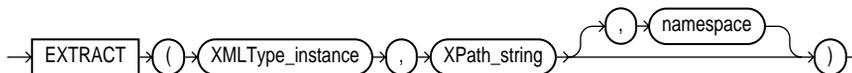
```
SELECT extractValue(value(x), '/PurchaseOrder/Reference')
FROM XMLTABLE X, SCOTT.EMP
WHERE extractValue(value(x), '/PurchaseOrder/User') = EMP.ENAME
AND EMP.EMPNO = 7876;

-- This returns:
-- EXTRACTVALUE(VALUE(X), '/PURCHASEORDER/REFERENCE')
-- -----
-- ADAMS-20011127121040988PST
```

Using extract()

The `extract()` syntax is shown in [Figure 3-3](#).

Figure 3-3 `extract()` Syntax



`extract()` is used when the XPath expression will result in a collection of nodes being returned. The nodes are returned as an instance of `XMLType`. The results of `extract()` can be either a `Document` or a `DocumentFragment`. The functionality of `extract` is also available through the `XMLType` datatype's `extract()` method.

Example 3-10 Using `extract()` to Return an XML Fragment

The following `extract()` statement returns an `XMLType` that contains an XML document fragment containing occurrences of the `Description` node. These match the specified XPath expression shown.

Note: In this case the XML is not well formed as it contains more than one root node.

```
set long 20000
```

```
SELECT extract(value(X),
              '/PurchaseOrder/LineItems/LineItem/Description')
       FROM XMLTABLE X;
```

```
-- This returns:
```

```
-- EXTRACT(VALUE(X), '/PURCHASEORDER/LINEITEMS/LINEITEM/DESCRIPTION')
```

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

```
-- <Description>The Ruling Class</Description>
```

```
-- <Description>Diabolique</Description>
```

```
-- <Description>8 1/2</Description>
```

Example 3-11 Using `extract()` to Return a Node Tree that Matches an XPath Expression

In this example `extract()` returns the node tree that matches the specified XPath expression:

```
SELECT extract(value(X),
              '/PurchaseOrder/LineItems/LineItem[1]')
FROM XMLTABLE X;
```

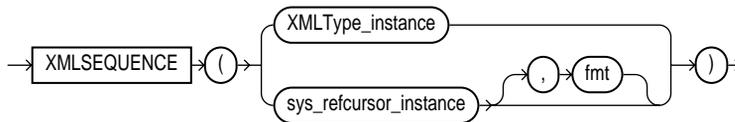
This returns:

```
EXTRACT(VALUE(X), '/PURCHASEORDER/LINEITEMS/LINEITEM[1]')
-----
<LineItem ItemNumber="1">
  <Description>The Ruling Class</Description>
  <Part Id="715515012423" UnitPrice="39.95" Quantity="2"/>
</LineItem>
```

Using XMLSequence()

The XMLSequence() syntax is shown in [Figure 3-4](#).

Figure 3-4 XMLSequence() Syntax



An XML document fragment can be converted into a set of XMLTypes using the XMLSequence() function. XMLSequence() takes an XMLType containing a document fragment and returns a collection of XMLType objects. The collection will contain one XMLType for each root level node in the fragment. The collection can then be converted into a set of rows using the SQL TABLE function.

Example 3-12 Using XMLSequence() and TABLE() to Extract Description Nodes from an XML Document

The following example shows how to use XMLSequence() and Table() to extract the set of Description nodes from the purchaseorder document.

```
set long 10000
set feedback on
SELECT extractValue(value(t), '/Description')
FROM XMLTABLE X,
     TABLE ( xmlsequence (
              extract(value(X),
                    '/PurchaseOrder/LineItems/LineItem/Description')
```

```

        )
    ) t;

```

This returns:

```
EXTRACTVALUE(VALUE(T), '/DESCRIPTION')
```

```

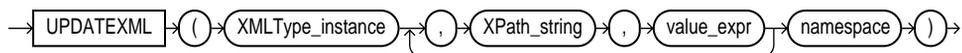
The Ruling Class
Diabolique
8 1/2

```

Updating XML Documents with updateXML()

The updateXML() syntax is shown in [Figure 3-5](#).

Figure 3-5 updateXML() Syntax



You can update XML documents using the updateXML() function. updateXML() updates an attribute value, node, text node, or node tree. The target for the update operation is identified using an XPath expression. The following examples show how you can use updateXML() to modify the contents of an XML Document stored as an XMLType.

Example 3-13 Using updateXML() to Update a Text Node Value Identified by an XPath Expression

This example uses updateXML() to update the value of the text node identified by the XPath expression '/PurchaseOrder/Reference':

```

UPDATE XMLTABLE t
  SET value(t) = updateXML(value(t),
                          '/PurchaseOrder/Reference/text()',
                          'MILLER-200203311200000000PST')
  WHERE existsNode(value(t),
                  '/PurchaseOrder[Reference="ADAMS-20011127121040988PST"']') = 1;

```

This returns:

```
1 row updated.
```

```
SELECT value(t)
FROM XMLTABLE t;
```

This returns:

```
VALUE(T)
-----
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.oracle.com/xdb/po.xsd">
  <Reference>MILLER-200203311200000000PST</Reference>
  ...
</PurchaseOrder>
```

Example 3–14 Using updateXML() to Replace Contents of a Node Tree Associated with XPath Elements

In this example `updateXML()` replaces the contents of the node tree associated with the element identified by the XPath expression `'/PurchaseOrders/LineItems/LineItem[2]'`.

Note: In this example, since the replacement value is a Node tree, the third argument to the `updateXML()` function is supplied as an instance of the `XMLType` datatype.

```
UPDATE XMLTABLE t
SET value(t) =
  updateXML(value(t),
    '/PurchaseOrder/LineItems/LineItem[2]',
    xmltype('<LineItem ItemNumber="4">
      <Description>Andrei Rublev</Description>
      <Part Id="715515009928" UnitPrice="39.95"
        Quantity="2"/>
    </LineItem>')
  )
WHERE existsNode(value(t),
  '/PurchaseOrder[Reference="MILLER-200203311200000000PST"]')
) = 1;
```

This returns:

```
1 row updated.
```

```
SELECT value(t)
FROM XMLTABLE t;
```

And this returns:

```
VALUE(T)
-----
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNames
paceSchemaLocation="http://www.oracle.com/xdm/po.xsd">
  <Reference>MILLER-200203311200000000PST</Reference>
  ...
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>The Ruling Class</Description>
      <Part Id="715515012423" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="4">
      <Description>Andrei Rublev</Description>
      <Part Id="715515009928" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>8 1/2</Description>
      <Part Id="037429135624" UnitPrice="39.95" Quantity="4"/>
    </LineItem>
  </LineItems>
</PurchaseOrder>
```

Introducing the W3C XSLT Recommendation

The W3C XSLT Recommendation defines an XML language for specifying how to transform XML documents from one form to another. Transformation can include mapping from one XML schema to another or mapping from XML to some other format such as HTML or WML.

See Also: [Appendix D, "XSLT Primer"](#) for an introduction to the W3C XSL and XSLT recommendations.

Example 3–15 XSL Stylesheet Example: PurchaseOrder.xsl

The following example, `PurchaseOrder.xsl`, is an example fragment of an XSL 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="#FFFFFF" 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/>
          ...
          <FONT FACE="Arial, Helvetica, sans-serif"
            COLOR="#000000">
            <xsl:for-each select="Part">
              <xsl:value-of select="@Quantity*@UnitPrice"/>
            </xsl:for-each>
          </FONT>
        </td>
      </tr>
    </tbody>
  </xsl:for-each>
</xsl:for-each>
</table>
</xsl:for-each>
</FONT>
</body>
</html>
</xsl:template>
</xsl:stylesheet>

```

See Also: [Appendix D, "XSLT Primer"](#) for the full listing of this XSL stylesheet.

Using XSL/XSLT with Oracle XML DB

Oracle XML DB complies with the W3C XSL/XSLT recommendation by supporting XSLT transformations in the database. In Oracle XML DB, XSLT transformations can be performed using either of the following:

- `XMLTransform()` function
- `XMLType` datatype's `transform()` method

Since XSL stylesheets are valid XML documents both approaches apply when the XSL stylesheets are provided as instances of the `XMLType` datatype. The results of the XSL transformation are also returned as an `XMLType`.

Because the transformation takes place close to the data, Oracle XML DB can optimize features such as memory usage, I/O operations, and network traffic required to perform the transformation.

See Also: [Chapter 6, "Transforming and Validating XMLType Data"](#)

Example 3-16 Using transform() to Transform an XSL

The following example shows how `transform()` can apply XSLT to an XSL stylesheet, `PurchaseOrder.xml`, to transform the `PurchaseOrder.xml` document:

```
SELECT value(t).transform(xmltype(getDocument('purchaseOrder.xml')))
   from XMLTABLE t
   where existsNode(value(t),
                    '/PurchaseOrder[Reference="MILLER-200203311200000000PST"]'
                    ) = 1;
```

This returns:

```
VALUE(T).TRANSFORM(XMLTYPE(GETDOCUMENT('PURCHASEORDER.XML')))
-----
<html>
  <head/>
  <body bgcolor="#003333" text="#FFFFCC" link="#FFCC00" vlink="#66CC99" alink="#
669999">
    <FONT FACE="Arial, Helvetica, sans-serif">
      <center>
    ...
      </FONT>
    </body>
  </html>
```

Since the transformed document using XSLT is expected as an instance of `XMLType`, the source could easily be a database table.

Other XMLType Methods

The following describes additional `XMLType` methods:

- `createXML()`. A static method for creating an `XMLType` instance. Different signatures allow the `XMLType` to be created from a number of different sources containing an XML document. Largely replaced by the `XMLType` constructor in Oracle 9i Release 2 (9.2).
- `isFragment()`. Returns true (1) if the `XMLType` contains a document fragment. A document fragment is an XML document without a Root Node. Document fragments are typically generated using the `extract()` function and method.
- `getClobVal()`. Returns a CLOB containing an XML document based on the contents of the `XMLType`.
- `getRootElement()`. Returns the name of the root element of the XML document contained in the `XMLType`.
- `getNamespace()`. Returns the name of the root element of the XML document contained in the `XMLType`.

Introducing the W3C XML Schema Recommendation

XML Schema provides a standardized way of defining what the expected contents of a set of XML documents should be. An XML schema is an XML document that defines metadata. This metadata specifies what the member contents of the document class should be. The members of a document class can be referred to as instance documents.

Since an XML schema definition is simply an XML document that conforms to the class defined by the XML Schema <http://www.w3.org/2001/XMLSchema>, XML schemas can be authored using a simple text editor, such as Notepad, vi, a schema-aware editor, such as the XML editor included with the Oracle9i JDeveloper tool, or an explicit XML schema authoring tool, such as XMLSpy from Altova Corporation. The advantage of using a tool such as XMLSpy, is that these tools allow the XML schema to be developed using an intuitive, graphical editor which hides much of the details of the XML schema definition from the developer.

Example 3–17 XML Schema Example, PurchaseOrder.xsd

The following example `PurchaseOrder.xsd`, is a standard W3C XML Schema example fragment, in its native form, as an XML Document:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="ActionTypes" >
    <xs:sequence>
      <xs:element name="Action" maxOccurs="4" >
        <xs:complexType >
          <xs:sequence>
            <xs:element ref="User"/>
            <xs:element ref="Date"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="RejectType" >
    <xs:all>
      <xs:element ref="User" minOccurs="0"/>
      <xs:element ref="Date" minOccurs="0"/>
      <xs:element ref="Comments" minOccurs="0"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="ShippingInstructionsType" >
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="address"/>
      <xs:element ref="telephone"/>
    </xs:sequence>
    ...
    ....
  <xs:complexType>
    <xs:attribute name="Id" >
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:minLength value="12"/>
          <xs:maxLength value="14"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="Quantity" type="money"/>
    <xs:attribute name="UnitPrice" type="quantity"/>
  </xs:complexType>
</xs:element>
```

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

See Also: [Appendix B, "XML Schema Primer"](#) for the detailed listing of `PurchaseOrder.xsd`.

Using XML Schema with Oracle XML DB

Oracle XML DB supports the use of the W3C XML Schema in two ways.

- Automatic Validation of instance documents
- Definition of Storage models

To use a W3C XML Schema with Oracle XML DB, the XML schema document has to be registered with the database. Once an XML schema has been registered XMLType tables and columns can be created which are bound to the schema.

To register an XML schema you must provide two items. The first is the XMLSchema document, the second is the URL which will be used by XML documents which claim to conform to this Schema. This URL will be provided in the root element of the instance document using either the `noNamespaceSchemaLocation` attribute or `schemaLocation` attribute as defined in the W3C XML Schema recommendation

XML schemas are registered using methods provided by PL/SQL package `DBMS_XMLSCHEMA`. Schemas can be registered as global or local schemas. See [Chapter 5, "Structured Mapping of XMLType"](#) for a discussion of the differences between Global and Local Schemas.

Oracle XML DB provides a number of options for automatically generating default database objects and Java classes as part of the schema registration process. Some of these options are discussed later in this section.

Example 3–18 Registering PurchaseOrder.xsd as a Local XML Schema Using registerSchema()

The following example shows how to register the preceding `PurchaseOrder.xsd` XML schema as a local XML schema using the `registerSchema()` method.

```
begin
  dbms_xmlschema.registerSchema(
    'http://www.oracle.com/xsd/purchaseOrder.xsd',
    getDocument('PurchaseOrder.xsd'),
    TRUE, TRUE, FALSE, FALSE
  );
end;
```

```

/
--This returns:
-- PL/SQL procedure successfully completed.

```

The `registerSchema()` procedure causes Oracle XML DB to perform the following operations:

- Parse and validate the XML schema
- Create a set of entries in Oracle Data Dictionary that describe the XML schema
- Create a set of SQL object definitions, based on the complexTypes defined in the XML schema

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

Example 3–19 Creating an XMLType Table that Conforms to an XML Schema

This example shows how to create an `XMLType` table which can only contain XML Documents that conform to the definition of the `PurchaseOrder` element in the XML schema registered at

```
'http://www.oracle.com/xsd/purchaseorder.xsd'.
```

```

CREATE TABLE XML_PURCHASEORDER of XMLType
  XMLSCHEMA "http://www.oracle.com/xsd/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder";

```

This results in:

Table created.

```
DESCRIBE XML_PURCHASEORDER
```

Returns the following:

Name	Null?	Type

TABLE of SYS.XMLTYPE(XMLSchema "http://www.oracle.com/xsd/purchaseOrder.xsd"		
Element "PurchaseOrder")	STORAGE	Object-relational TYPE "PurchaseOrder538_T"

XMLSchema-Instance Namespace

Oracle XML DB must recognize that the XML document inserted into an XML schema-based table or column is a valid member of the class of documents defined by the XML schema. The XML document must correctly identify the XML schema or XML schemas it is associated with.

This means that XML schema, for each namespace used in the document, must be identified by adding the appropriate attributes to the opening tag for the root element of the document. These attributes are defined by W3C XML Schema recommendation and are part of the W3C XMLSchema-Instance namespace. Consequently in order to define these attributes the document must first declare the XMLSchema-instance namespace. This namespace is declared as follows:

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

Once the XMLSchema-instance namespace has been declared and given a namespace prefix the attributes that identify the XML schema can be added to the root element of the instance document. A given document can be associated with one or more XML schemas. In the preceding example, the namespace prefix for the XMLSchema-instance namespace was defined as `xsi`.

noNameSpaceSchemaLocation Attribute. The XML schema associated with the unqualified elements is defined using the attribute `noNameSpaceSchemaLocation`. In the case of the `PurchaseOrder.xsd` XML schema, the correct definition would be as follows:

```
<PurchaseOrder
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:noNameSpaceSchemaLocation="http://www.oracle.com/xsd/purchaseOrder.xsd">
```

Using Multiple Namespaces: schemaLocation Attribute. If the XML document uses multiple namespaces then each namespace needs to be identified by a `schemaLocation` attribute. For example, assuming that the `PurchaseOrder` document used the namespace `PurchaseOrder`, and the `PurchaseOrder` namespace is given the prefix `po`. The definition of the root element of a `PurchaseOrder` document would then be as follows:

```
<po:PurchaseOrder
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:po="PurchaseOrder"
  xsi:schemaLocation="PurchaseOrder
http://www.oracle.com/xsd/purchaseOrder.xsd">
```

Validating an XML Document Using an XML Schema

By default Oracle XML DB performs a minimum amount of validation when a storing an instance document. This minimal validation ensures that the structure of the XML document conforms to the structure specified in the XML schema.

Example 3–20 Attempting to Insert an Invoice XML Document Into an XMLType Table Conforming to PurchaseOrder XML Schema

The following example shows what happens when an attempt is made to insert an XML Document containing an invoice into a XMLType table that is defined as storing documents which conform to the PurchaseOrder Schema

```
INSERT INTO XML_PURCHASEORDER
  values (xmltype(getDocument('Invoice.xml')))
  values (xmltype(getDocument('Invoice.xml')))
  *
```

This returns:

```
ERROR at line 2:
ORA-19007: Schema and element do not match
```

The reason for not performing full instance validation automatically is based on the assumption that, in the majority of cases it is likely that schema based validation will have been performed prior to attempting to insert the XML document into the database.

In situations where this is not the case, full instance validation can be enabled using one of the following approaches:

- A table level CHECK constraint
- A PL/SQL BEFORE INSERT trigger

Example 3–21 Using CHECK Constraints in XMLType Tables

This example shows how to use a CHECK constraint to an XMLType table and the result of attempting to insert an invalid document into the table:

```
ALTER TABLE XML_PURCHASEORDER
  add constraint VALID_PURCHASEORDER
  check (XMLIsValid(sys_nc_rowinfo$)=1);

-- This returns:
-- Table altered

INSERT INTO XML_PURCHASEORDER
```

```

        values (xmltype(getDocument('InvalidPurchaseOrder.xml')));
INSERT INTO XML_PURCHASEORDER;
*
-- This returns:
-- ERROR at line 1:
-- ORA-02290: check constraint (DOC92.VALID_PURCHASEORDER) violated

```

Example 3–22 Using BEFORE INSERT Trigger to Validate Data Inserted Into XMLType Tables

The next example shows how to use a BEFORE INSERT trigger to validate that the data being inserted into the XMLType table conforms to the specified schema

```

CREATE TRIGGER VALIDATE_PURCHASEORDER
  before insert on XML_PURCHASEORDER
  for each row
  declare
    XMLDATA xmltype;
  begin
    XMLDATA := :new.sys_nc_rowinfo$;
    xmltype.schemavalidate(XMLDATA);
  end;
/

-- This returns:
-- Trigger created.

insert into XML_PURCHASEORDER
  values (xmltype(getDocument('InvalidPurchaseOrder.xml')));

-- values (xmltype(getDocument('InvalidPurchaseOrder.xml')))
-- *
-- ERROR at line 2:
-- ORA-31154: invalid XML document
-- ORA-19202: Error occurred in XML processing
-- LSX-00213: only 0 occurrences of particle "User", minimum is 1
-- ORA-06512: at "SYS.XMLTYPE", line 0
-- ORA-06512: at "DOC92.VALIDATE_PURCHASEORDER", line 5
-- ORA-04088: error during execution of trigger 'DOC92.VALIDATE_PURCHASEORDER'

```

As can be seen both approaches ensure that only valid XML documents can be stored in the XMLType table:

- Table CHECK Constraint.** The TABLE constraint approach's advantage is that it is simpler to code. Its disadvantage is that, since it is based on the

`isSchemaValid()` method, it can only indicate whether or not the instance document is valid. When the instance document is not valid it cannot give any information as to *why* a document is invalid.

- BEFORE INSERT Trigger.** The BEFORE INSERT trigger requires a little more coding. Its advantage is that it is based on the `schemaValidate()` method. This means that when the instance document is not valid it can provide information about what was wrong with the instance document. It also has the advantage of allowing the trigger to take corrective action when appropriate.

Storing XML: Structured or Unstructured Storage

When designing an Oracle XML DB application you must first decide whether the `XMLType` columns and table will be stored using structured or unstructured storage techniques.

Table 3–1 compares using structured and structured storage to store XML.

Table 3–1 Comparing Structured and Unstructured XML Storage

Feature	Unstructured XML Storage	Structured XML Storage
Storage technique	Contents of <code>XMLType</code> columns and tables are stored using the CLOB data type.	Contents of <code>XMLType</code> columns and tables are stored as a collection of SQL objects. By default, the underlying storage model for XML schema-based <code>XMLType</code> columns and tables is structured storage.
Can store non-XML schema-based tables?	Only option available for <code>XMLType</code> tables and columns that are not associated with an XML schema.	Can only be used when the <code>XMLType</code> column or table is based on an XML schema. This means that the instance documents must conform to the underlying XML schema.
Performance: Storage and retrieval speed	It allows for higher rates of ingestion and retrieval, as it avoids the overhead associated with parsing and recomposition during storage and retrieval operations.	Results in a slight overhead during ingestion and retrieval operations in that the document has to be shredded during ingestion and re-constituted prior to retrieval.

Table 3–1 Comparing Structured and Unstructured XML Storage (Cont.)

Feature	Unstructured XML Storage	Structured XML Storage
Performance: operation speed	Slower than for structured storage.	<p>When an XML schema is registered, Oracle XML DB generates a set of SQL objects that correspond to <code>complexType</code>s defined in the XML schema. XPath expressions sent to Oracle XML DB functions are translated to SQL statements that operate directly against the underlying objects.</p> <p>This re-writing of <code>XMLType</code> operations into object-relational SQL statements results in significant performance improvements compared with performing the same operations against XML documents stored using unstructured storage.</p>
Flexible. Can easily process varied content?	Allows for a great deal of flexibility in the documents being processed making it an appropriate choice when the XML documents contain highly variable content.	Leverages the object-relational capabilities of the Oracle9i database.
Memory usage: Do the XML documents need parsing?	Oracle XML DB must parse the entire XML document and load it into an in-memory DOM structure before any validation, XSL Transformation, or XPath operations can be performed on it.	<p>Allows Oracle XML DB to minimize memory usage and optimize performance of DOM-based operations on <code>XMLType</code> table and columns by using:</p> <ul style="list-style-type: none"> ■ Lazy Manifestation (LM): Occurs when Oracle XML DB constructs a DOM structure based on an XML document. With LM, instead of constructing the whole DOM when the document is accessed, Oracle XML DB only instantiates the nodes required to perform the immediate operation. As other parts of the document are required the appropriate node trees are dynamically loaded into the DOM. ■ Least Recently Used (LRU): Strategy to discard nodes in the DOM that have not been accessed recently.

Table 3–1 Comparing Structured and Unstructured XML Storage (Cont.)

Feature	Unstructured XML Storage	Structured XML Storage
Update processing	<p>When stored, any update operations on the document will result in the entire CLOB being re-written.</p> <p>If any part of the document is updated using <code>updateXML()</code> then the entire document has to be fetched from the CLOB, updated, and written back to the CLOB.</p>	<p>Can update individual elements, attributes, or nodes in an XML document without rewriting the entire document.</p> <p>Possible to re-write the <code>updateXML()</code> operation to an SQL UPDATE statement that operates on columns or objects referenced by the XPATH expression.</p>
Indexing	<p>You can use B*Tree indexes based on the functional evaluation of XPath expressions or Oracle Text inverted list indexes.</p> <p>Unstructured storage make it impossible to create B*TREE indexes based on the values of elements or attributes that appear within collections.</p>	<p>You can use B*Tree indexes and Oracle Text inverted list indexes.</p> <p>By tuning the way in which collections are managed, indexes can be created on any element or attribute in the document, including elements or attributes that appear with collections.</p>
Space needed	Can be large.	Since based on XML schema, it is not necessary for Oracle XML DB to store XML tag names when storing the contents of XML documents. This can significantly reduce the storage space required.
Data integrity	--	Makes it possible to use a set of database integrity constraints that allow the contents of an XML document to be validated against information held elsewhere in the database.
Tuning: Fine-grained object control	None	<p>You can annotate XML schema, for fine grain control over sets of SQL objects generated from XML schema and how these objects are stored in the database.</p> <p>You can control how collections are managed, define tablespace usage, and partitioning of table or tables used to store and manage the SQL objects. This makes it possible to fine tune the performance of the Oracle XML DB to meet the needs of the application.</p> <p>Other annotations control how Simple elements and attributes are mapped to SQL columns</p>

Data Manipulation Language (DML) Independence

Oracle XML DB ensures that all Data Manipulation Language (DML) operations based on Oracle XML DB functions return consistent results. By abstracting the storage model through the use of the `XMLType` datatype, and providing a set of operators that use XPath to perform operations against XML documents, Oracle XML DB makes it possible for you to *switch between structured and unstructured storage*, and to experiment with different forms of structured storage without affecting the application.

DOM Fidelity in Structured and Unstructured Storage

To preserve DOM fidelity a system must ensure that a DOM generated from the stored representation of an XML Document is identical to a DOM generated from the original XML document. Preserving DOM integrity ensures that none of the information contained in the XML Document is lost as a result of storing it.

The problem with maintaining DOM integrity is that an XML document can contain a lot of information in addition to the data contained in element and attribute values. Some of this information is explicitly provided, using Comments and Processing Instructions. Other information can be implicitly provided, such as:

- Ordering of the elements in a collection
- Ordering of child elements within the parent
- Relative position of Comments and Processing Instructions

One of the common problems application developers face when using a traditional relational model to manage the contents of XML documents is how to preserve this information. [Table 3-2](#) compares DOM fidelity in structured and unstructured storage:

Table 3–2 DOM Fidelity: Unstructured and Structured Storage

DOM Fidelity with Unstructured Storage	DOM Fidelity with Structured Storage
Relational systems do not provide any implicit ordering, nor do they provide the flexibility to make it easy to preserve out of band data such as comments and processing instructions. With a typical relational database, the only way to preserve DOM Fidelity is to store the source document using unstructured storage techniques	Oracle XML DB can preserve DOM Fidelity even with structured storage. When an XML Document is shredded and stored using structured storage techniques, the Comments, Processing Instructions, and any ordering information implicit in the source document is preserved as part of the SQL objects that are created when the document is shredded. When the document is retrieved this information is incorporated back into the generated XML document.

Structured Storage: XML Schema-Based Storage of XMLType

Logically, an XML document consists of a collection of elements and attributes. Elements can be either of the following:

- `complexType`s, containing child elements and attributes
- `simpleTypes`, containing scalar values

An XML schema defines the set of elements and attributes that can exist in a particular class of XML document and defines the relationships between them.

During XML schema registration, Oracle XML DB generates an SQL Object Type for each `complexType` defined in the XML schema. The definition of the SQL object mirrors the definition of the `complexType`.

Each child element and attribute defined by the `complexType` maps to an attribute of the SQL object type.

- If a child element in the `complexType` is itself a `complexType`, the datatype of the corresponding SQL attribute will be the appropriate SQL type.
- If the child element is a `simpleType` or attribute, based on one of the scalar datatypes defined by the W3C XML Schema recommendation, then the datatype of the corresponding SQL attribute will be the appropriate primitive SQL data type.

XML Schema Names and Defining Oracle XML DB Namespace

By default SQL Objects generated when an XML schema is registered are given system-generated names. However, with Oracle XML DB you can *specify* the names

of SQL objects by annotating the schema. To annotate an XML schema, you must first include the Oracle XML DB namespace in the XMLSchema tag, defined as:

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

Hence an XML schema using Oracle XML DB annotations, must contain the following attributes in the XMLSchema tag:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb" >
...
</xs:schema>
```

Once Oracle XML DB namespace has been defined, the annotations defined by Oracle XML DB can be used.

Example 3–23 Defining the Name of SQL Objects Generated from complexTypes

This example uses `xdb:SQLType` to define the name of the SQL object generated from complexType `PurchaseOrder`, as `XML_PURCHASEORDER_TYPE`.

```
<xs:element name="PurchaseOrder">
  <xs:complexType type="PurchaseOrderType"
                 xdb:SQLType="XML_PURCHASEORDER_TYPE">
    <xs:sequence>
      <xs:element ref="Reference"/>
      <xs:element name="Actions" type="ActionsType"/>
      <xs:element name="Reject" type="RejectType" minOccurs="0"/>
      <xs:element ref="Requestor"/>
      <xs:element ref="User"/>
      <xs:element ref="CostCenter"/>
      <xs:element name="ShippingInstructions"
                  type="ShippingInstructionsType"/>
      <xs:element ref="SpecialInstructions"/>
      <xs:element name="LineItems" type="LineItemsType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

So executing the following statement:

```
DESCRIBE XML_PURCHASEORDER_TYPE
XML_PURCHASEORDER_TYPE is NOT FINAL;
```

Returns the following structure:

Name	Null?	Type
SYS_XDBPD\$		XDB.XDB\$RAW_LIST_T
Reference		VARCHAR2(26)
Actions		XML_ACTIONS_TYPE
Reject		XML_REJECTION_TYPE
Requestor		VARCHAR2(128)
User		VARCHAR2(10)
CostCenter		VARCHAR2(4)
ShippingInstructions		XML_SHIPPINGINSTRUCTIONS_TYPE
SpecialInstructions		VARCHAR2(2048)
LineItems		XML_LINEITEMS_TYPE

Note: In the preceding example, `xdb:SQLType` annotation was also used to assign names to the SQL types that correspond to the `complexTypees:ActionsType`, `ShippingInstructionsType` and `LineItemsType`.

Using `xdb:SQLName` to Override Default Names

Oracle XML DB uses a predefined algorithm to generate valid SQL names from the names of the XML elements, attributes, and types defined in the XML schema. The `xdb:SQLName` annotation can be used to override the default algorithm and supply explicit names for these items.

Using `xdb:SQLType` to Override Default Mapping

Oracle XML DB also provides a default mapping between scalar datatypes defined by the XML Schema recommendation and the primitive datatype defined by SQL. Where possible the size of the SQL datatype is derived from restrictions defined for the XML datatype. If required, the `xdb:SQLType` annotation can be used to override this default mapping:

Example 3–24 Using `xdb:SQLType` and `xdb:SQLName` to Specify the Name and Mapping of Objects Generated from `complexTypees`

This example shows how to override the name and type used for the `SpecialInstructions` element and the effect these changes have on the generated SQL Object type.

Note: The override for the name of the SpecialInstructions element is applied where the element is used, inside the PurchaseOrderType, not where it is defined

```

<xs:element name="SpecialInstructions" xdb:SQLType="CLOB" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="0"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="PurchaseOrder">
  <xs:complexType type="PurchaseOrderType"
    xdb:SQLType="XML_PURCHASEORDER_TYPE">
    <xs:sequence>
      <xs:element ref="Reference" />
      <xs:element name="Actions" type="ActionsType" />
      <xs:element name="Reject" type="RejectType" minOccurs="0"/>
      <xs:element ref="Requestor" />
      <xs:element ref="User" />
      <xs:element ref="CostCenter" />
      <xs:element name="ShippingInstructions"
        type="ShippingInstructionsType" />
      <xs:element ref="SpecialInstructions"
        xdb:SQLName="SPECINST" />
      <xs:element name="LineItems" type="LineItemsType" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

On executing the following statement:

```

DESCRIBE XML_PURCHASEORDER_TYPE
XML_PURCHASEORDER_TYPE is NOT FINAL

```

The following structure is returned:

Name	Null?	Type
SYS_XDBPD\$		XDB.XDB\$RAW_LIST_T
Reference		VARCHAR2(26)

Actions	XML_ACTIONS_TYPE
Reject	XML_REJECTION_TYPE
Requestor	VARCHAR2(128)
User	VARCHAR2(10)
CostCenter	VARCHAR2(4)
ShippingInstructions	XML_SHIPPINGINSTRUCTIONS_TYPE
SPECINST	CLOB
LineItems	XML_LINEITEMS_TYPE

Structured Storage: Storing complexType Collections

One issue you must consider when selecting structured storage, is what techniques to use to manage Collections. Different approaches are available and each approach offers different benefits. Generally, you can handle Collections in five ways:

- **CLOBs.** If a `complexType` is defined with `xdb:SQLType="CLOB"` then the type, and all child elements are stored using *unstructured* storage techniques
- **Inline VARRAYS.** If no other information is given for a `complexType` which occurs more than once, the members of the collection are stored as a set of serialized objects in-line as part of the SQL object for the parent element. You cannot create B*Tree indexes on elements or attributes which are part of collection
- **Nested Object Tables.** The members of the collection are stored in a nested object table. The SQL object, as in previous option, contains an attribute of type VARRAY, but is stored as a table. The parent row contains a unique `setid` (set identifier) value which is used to associate with the corresponding nested table rows.
- **Separate XMLType Table.** The members of the collection are stored as a separate XMLType table. Each member of the collection is stored as a row in the table. The Parent SQL object contains an array of refs which point to the rows in the child table which belong to this parent. All data is XMLType.
 - Creating multiple XMLType columns based on the XML schema
 - Linking from a child to the corresponding parent
- **Separate XMLType Table with Link Table.** The members of the collection are stored as a separate XMLType table. An link table is created which cross references which member in the child table are linked to which members of the parent. Table. All data is visible as XMLTypes. Possible to link from the child back to the parent. Problems with creating multiple XMLType columns based on the Schema.

See Also: [Chapter 5, "Structured Mapping of XMLType"](#)

Structured Storage: Data Integrity and Constraint Checking

In addition to schema-validation, structured storage makes it possible to introduce traditional relational constraints on to XMLType columns and Tables. With database integrity checking you can perform instance validation beyond what is achievable with XML Schema-based validation.

The W3C XML Schema Recommendation only allows for validation based on cross-referencing of values with an instance document. With database integrity checking you can enforce other kinds of validation, such as enforcing the uniqueness of a element or attribute across a collection of documents, or validating the value of a element or attribute against information stored elsewhere in the database.

Note: In Oracle9i Release 2 (9.2) constraints have to be specified using object-relational syntax.

Example 3–25 Adding a Unique and Referential Constraint to Table Purchaseorder

The following example shows how you can introduce a Unique and Referential Constraint on the PurchaseOrder table.

```
XMLDATA.SQLAttributeName
alter table XML_PURCHASEORDER
add constraint REFERENCE_IS_UNIQUE
-- unique(extractValue('/PurchaseOrder/Reference'))
unique (xmldata."Reference");

alter table XML_PURCHASEORDER
add constraint USER_IS_VALID
-- foreign key extractValue('/PurchaseOrder/User') references
SCOTT.EMP(ENAME)
foreign key (xmldata."User") references SCOTT.EMP(ENAME);
```

As can be seen, when an attempt is made to insert an XML Document that contains a duplicate value for the element `/PurchaseOrder/Reference` into the table, the database detects that the insert would violate the unique constraint, and raises the appropriate error.

```
insert into xml_purchaseorder values (
  xmltype(getDocument('ADAMS-20011127121040988PST.xml'))
```

```
);
```

This returns:

```
1 row created.
```

```
insert into xml_purchaseorder values (  
  xmltype(getDocument('ADAMS-20011127121040988PST.xml'))  
);
```

```
insert into xml_purchaseorder values (  
*
```

This returns:

```
ERROR at line 1:  
ORA-00001: unique constraint (DOC92.REFERENCE_IS_UNQIUE) violated
```

Example 3–26 How Oracle9i Database Enforces Referential Constraint User_Is_Valid

The following example shows how the database will enforce the referential constraint `USER_IS_VALID`, which states that the value of the element `/PurchaseOrder/User`, that translates to the `SQLAttribute xmldata.user`, must match one of the values of `ENAME` in `SCOTT.EMP`.

```
insert into xml_purchaseorder values (  
  xmltype(getDocument('HACKER-20011127121040988PST.xml'))  
);
```

```
insert into xml_purchaseorder values (  
*
```

This returns:

```
ERROR at line 1:  
ORA-02291: integrity constraint (SCOTT.USER_IS_VALID)  
violated - parent key notfound
```

Oracle XML DB Repository

XML documents are by nature hierarchical animals. The information they contain is represented by a *hierarchy* of elements, child elements, and attributes. XML documents also view the world around them as a hierarchy. When an XML document refers to another XML document, or any other kind of document, it does so using a URL. URLs can be either relative or absolute. In either case, the URL

defines a path to the target document. The path is expressed in terms of a *folder hierarchy*.

Oracle XML DB Repository makes it possible to view all of XML content stored in the database using a File / Folder metaphor. The Repository provides support for basic operations such as creating files and folders as well as more advanced features such as version and access control.

The Repository is fully accessible, queryable, and updatable through SQL. It can also be directly accessed through industry standard protocols such as HTTP, WebDAV, and FTP.

See Also: [Chapter 13, "Oracle XML DB Foldering"](#)

Introducing the IETF WebDAV Standard

WebDAV is an Internet Engineering Task Force (IETF) Standard for Distributed Authoring and Versioning of content. The standard is implemented by extending the HTTP protocol allowing a Web Server to act as a File Server in a distributed environment.

Oracle XML DB Repository is Based on WebDAV

Oracle XML DB Repository is based on the model defined by the WebDAV standard. It uses the WebDAV resource model to define the basic metadata that is maintained for each document stored in the Repository. The WebDAV protocol uses XML to transport metadata between the client and the server.

Hence, you can easily create, edit, and access documents stored in Oracle XML DB Repository using standard tools. For example, you can use:

- Microsoft Web Folders
- Other WebDAV-enabled products, such as Microsoft Office, Macromedia, and the Adobe range of authoring tools.

WebDAV uses the term *Resource* to define a file or folder. It defines a set of basic operations that can be performed on a Resource. These operations require a WebDAV server to maintain a set of basic metadata for each Resource. Oracle XML DB exposes this metadata as a set of XML Documents in the following form:

Example 3–27 Oracle XML DB Exposes WebDAV Resource Metadata as XML Documents

```
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
  Hidden="false" Invalid="false" Container="false"
```

```
      CustomRslv="false">
<CreationDate> 2002-02-14T16:01:01.066324000</CreationDate>
<ModificationDate> 2002-02-14T16:01:01.066324000</ModificationDate>
<DisplayName>testFile.xml</DisplayName>
<Language>us english</Language>
<CharacterSet>utf-8</CharacterSet>
<ContentType>text/xml</ContentType>
<RefCount>1</RefCount>
<ACL>
  <acl description="/sys/acls/all_all_acl.xml"
        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">
    <ace>
      <grant>true</grant>
      <privilege>
        <all/>
      </privilege>
      <principal>PUBLIC</principal>
    </ace>
  </acl>
</ACL>
<Owner>DOC92</Owner>
<Creator>DOC92</Creator>
<LastModifier>DOC92</LastModifier>
<SchemaElement>
  http://xmlns.oracle.com/xdb/XDBSchema.xsd#binary
</SchemaElement>
<Contents>
  <binary>02C7003802C77B7000081000838B1C24000000002C71E7C</binary>
</Contents>
</Resource>
```

Query-Based Access to Oracle XML DB Repository

The Oracle XML DB exposes the Repository to the SQL developer as two views. The two views are:

- RESOURCE_VIEW
- PATH_VIEW

It also provides a set of SQL functions and PL/SQL packages for performing Repository operations.

See Also: [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)

Using RESOURCE_VIEW

RESOURCE_VIEW is the primary way for querying Oracle XML DB Repository. There is one entry in the RESOURCE_VIEW for each document stored in the Repository. The RES column contains the resource entry for the document, the ANY_PATH entry provides a valid folder path from the root to the resource.

The definition of the RESOURCE_VIEW is as follows:

```
SQL> describe RESOURCE_VIEW
```

Name	Null?	Type
RES		SYS.XMLTYPE
ANY_PATH		VARCHAR2(4000)

Using PATH_VIEW

PATH_VIEW contains an entry for each Path in the Repository. Since a Resource can be linked into more than one folder, PATH_VIEW shows all possible Paths in the Repository and the resources they point to. The definition of the PATH_VIEW is as follows:

```
SQL> describe PATH_VIEW
```

Name	Null?	Type
PATH		VARCHAR2(1024)
RES		SYS.XMLTYPE
LINK		SYS.XMLTYPE

Creating New Folders and Documents

You can create new folders and documents using methods provided by the DBMS_XDB package. For instance a new folder can be created using the procedure createFolder() and a file can be uploaded into that folder using createResource(). The following examples show you how to do this:

Example 3–28 *Creating a Repository Resource and Folder*

```
SQL> declare
  2   result boolean;
```

```
3 begin
4   result := dbms_xdb.createFolder('/public/testFolder');
5 end;
6 /
```

PL/SQL procedure successfully completed.

```
SQL> declare
2   result boolean;
3 begin
4   result := dbms_xdb.createResource(
5           '/public/testFolder/testFile.xml',
6           getDocument('testFile.xml')
7   );
8 end;
9 /
```

PL/SQL procedure successfully completed.

Querying Resource Documents

`RESOURCE_VIEW` can be queried just like any other view. Oracle XML DB provides a new operator, `UNDER_PATH`, that provides a way for you to restrict queries to a particular folder tree within the `RESOURCE_VIEW`.

`extractValue()` and `existsNode()` can be used on the Resource documents when querying the `RESOURCE_VIEW` and `PATH_VIEW` Resource documents.

Updating Resources

You can update Resources using `updateXML()`.

Example 3–29 Updating Repository Resources

For instance, the following query updates the `OWNER` and `NAME` of the document created in the previous example.

```
update RESOURCE_VIEW
  set RES=updateXML(RES,
                    '/Resource/DisplayName/text()', 'RenamedFile',
                    '/Resource/Owner/text()', 'SCOTT'
  )
where any_path = '/public/testFolder/testFile.xml';

-- 1 row updated.
```

```

select r.res.getClobVal()
  from RESOURCE_VIEW r
  where ANY_PATH = '/public/testFolder/testFile.xml'
/

-- Results in:
-- R.RES.GETCLOBVAL()
-- -----
-- <Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
--           Hidden="false" Invalid="false" Container="false"
--           CustomRslv="false">
--   <CreationDate> 2002-02-14T16:01:01.066324000</CreationDate>
--   <ModificationDate> 2002-02-14T21:36:39.579663000</ModificationDate>
--   <DisplayName>RenamedFile</DisplayName>
--   <Language>us english</Language>
--   <CharacterSet>utf-8</CharacterSet>
--   <ContentType>text/xml</ContentType>
-- <RefCount>1</RefCount>
-- <ACL>
-- ...
-- </ACL>
--   <Owner>SCOTT</Owner>
--   <Creator>DOC92</Creator>
--   <LastModifier>DOC92</LastModifier>
-- </Resource>

```

Deleting Resources

Resource can be deleted using `deleteResource()`. If the resource is a folder then the folder must be empty before it can be deleted.

Example 3-30 *Deleting Repository Resources*

The following examples show the use of the `deleteResource()` procedure.

```

call dbms_xdb.deleteResource('/public/testFolder')
/
call dbms_xdb.deleteResource('/public/testFolder')
*

ERROR at line 1:
ORA-31007: Attempted to delete non-empty container /public//testFolder
ORA-06512: at "XDB.DBMS_XDB", line 151
ORA-06512: at line 1

```

```
call dbms_xdb.deleteResource('/public/testFolder/testFile.xml')  
/
```

Call completed.

```
call dbms_xdb.deleteResource('/public/testFolder')  
/
```

Call completed.

Storage Options for Resources

RESOURCE_VIEW and PATH_VIEW are based on tables stored in the Oracle XML DB database schema. The metadata exposed through the RESOURCE_VIEW and PATH_VIEW is stored and managed using a set of tables in the Oracle XML DB supplied XML schema, XDBSchema.xsd. The contents of the files are stored as BLOB or CLOB columns in this XML schema.

See Also: [Appendix G, "Example Setup scripts. Oracle XML DB-Supplied XML Schemas", "xdbconfig.xsd: XML Schema for Configuring Oracle XML DB"](#) on page G-18

Defining Your Own Default Table Storage for XML Schema-Based Documents

There is an exception to this storage paradigm when storing XML schema-based XML documents. When an XML schema is registered with Oracle XML DB you can *define a default storage table for each root element* defined in the XML schema.

You can define your own default storage tables by adding an `xdb:defaultTable` attribute to the definition of the top level element. When the schema is registered, Oracle XML DB establishes a link between the Repository and the default tables defined by your XML schema. You can choose to generate the default tables as part of the XML schema registration.

Your Default Table is an XMLType Table and Hierarchically Enabled

A default table is an XMLType table, that is, it is an object table based on the XMLType datatype. When an XML document, with a root element and XML schema *that match your default table's root element and XML schema*, is inserted into the Repository, the XML content is stored as a row in the specified default table. A

resource is created that contains a reference to the appropriate row in the default table.

One of the special features of an XMLType table is that it can be *hierarchically enabled*. Default Tables, created as part of XML schema registration are automatically hierarchically enabled. When a table is hierarchically enabled DML operations on the default table may cause corresponding operations on the Oracle XML DB Repository. For example, when a row is deleted from the default table, any entries in the Repository which reference that row are deleted.

Example 3-31 Adding the xdb:defaultTable Attribute to the XML Schema's Element Definition

The following example shows the result of adding an xdb:defaultTable attribute to the XML schema definition's PurchaseOrder element and then registering the XML schema with the Create Table option set to TRUE:

```
<xs:element name="PurchaseOrder" xdb:defaultTable="XML_PURCHASEORDER">
  <xs:complexType type="PurchaseOrderType"
    xdb:SQLType="XML_PURCHASEORDER_TYPE">
    <xs:sequence>
      <xs:element ref="Reference"/>
      <xs:element name="Actions" type="ActionsType"/>
      <xs:element name="Reject" type="RejectType" minOccurs="0"/>
      <xs:element ref="Requestor"/>
      <xs:element ref="User"/>
      <xs:element ref="CostCenter"/>
      <xs:element name="ShippingInstructions"
        type="ShippingInstructionsType"/>
      <xs:element ref="SpecialInstructions"
        xdb:SQLName="SPECINST"/>
      <xs:element name="LineItems" type="LineItemsType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
SQL> begin
2   dbms_xmlschema.registerSchema(
3     'http://www.oracle.com/xsd/purchaseOrder.xsd',
4     getDocument('purchaseOrder3.xsd'),
5     TRUE, TRUE, FALSE, TRUE
6   );
7
8 end;
9 /
```

PL/SQL procedure successfully completed.

SQL> describe XML_PURCHASEORDER

```

Name                                Null?    Type
-----
TABLE of SYS.XMLTYPE(XMLSchema
http://www.oracle.com/xsd/purchaseOrder.xsd Element "PurchaseOrder")
STORAGE Object-relational TYPE "XML_PURCHASEORDER_TYPE"

```

Example 3-32 Inserting an XML Document into Oracle XML DB Repository Causes a Insertion of a Row into the Table

The following example shows how, once the XML schema is registered, and the default table created, when inserting an XML document into Oracle XML DB Repository causes a row to be inserted into the designated default table:

```
select count(*) from XML_PURCHASEORDER;
```

Results in:

```

COUNT(*)
-----
          0

-- create testFolder
declare
  result boolean;
begin
  result := dbms_xdb.createFolder('/public/testFolder');
end;
/

declare
  result boolean;
begin
  result := dbms_xdb.createResource(
              '/public/testFolder/purchaseOrder1.xml',
              getDocument('purchaseOrder1.xml')
            );
end;
/

-- PL/SQL procedure successfully completed.

```

```

commit;

-- Commit complete.

select count(*) from XML_PURCHASEORDER;

```

Results in:

```

COUNT(*)
-----
1

```

Example 3–33 Deleting a Row Causes Deletion of Corresponding Entry from the Repository

This example shows when deleting a row from the hierarchy enabled default table, the corresponding entry is deleted from the hierarchy.

```

select extractValue(res, 'Resource/DisplayName') "Filename"
       from RESOURCE_VIEW where under_path(res, '/public/testFolder') = 1;
/

```

Results in:

```

Filename
-----
purchaseOrder1.xml

```

```

delete from XML_PURCHASEORDER;
1 row deleted.

```

```

SQL> commit;
Commit complete.

```

```

select extractValue(res, 'Resource/DisplayName') "Filename"
       from RESOURCE_VIEW where under_path(res, '/public/testFolder') = 1
/

```

Results in:

```

no rows selected

```

Accessing XML Schema-Based Content

When a resource describes XML content that has been stored in a default table the resource entry itself simply contains a reference to the appropriate row in the

default table. This reference can be used to perform join operations between the resource and its content. This can be seen in the following example.

Accessing Non-Schema-Based Content With XDBUriType

XDBUriType can be used to access the contents of a file stored in the Repository using a logical path. The following example shows how to access a resource associated with a JPEG file. The JPEG file has been inserted into the Repository. The example uses Oracle *interMedia* `ordsys.ordimage` class to extract the metadata associated with the JPEG file.

```
create or replace function getImageMetaData (uri varchar2)
return xmltype deterministic
is
    resType xmltype;
    resObject xdb.xdb$resource_t;
    attributes CLOB;
    xmlAttributes xmltype;
begin
    DBMS_LOB.CREATETEMPORARY(attributes, FALSE, DBMS_LOB.CALL);
    -- ordsys.ordimage.getProperties(xdburitype(uri).getBlob(),
    --                               attributes);
    select res into resType from resource_view where any_path = uri;
    resType.toObject(resObject);
    ordsys.ordimage.getProperties(resObject.XMLLOB,attributes);
    xmlAttributes := xmltype(attributes);
    DBMS_LOB.FREETEMPORARY(attributes);
    return xmlAttributes;
end;
/
```

Oracle XML DB Protocol Servers

Oracle XML DB includes three protocol servers through which you can access the Repository directly from standard file-based applications.

See Also: [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#)

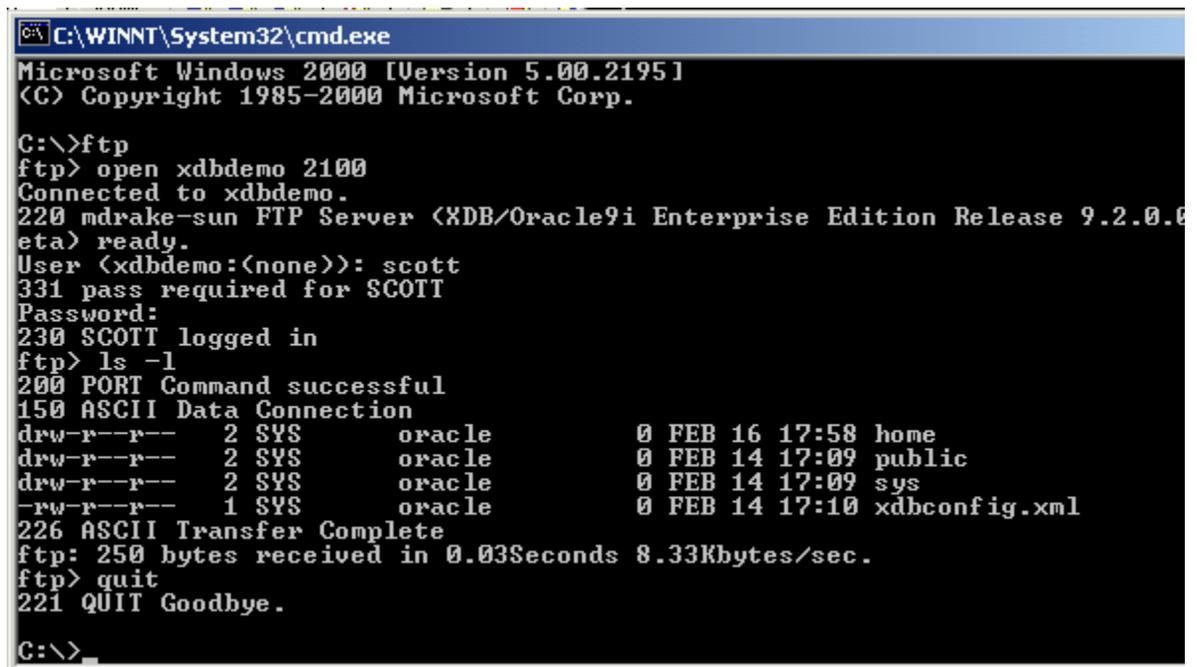
Using FTP Protocol Server

The FTP Protocol Server allows standard FTP clients to access content stored in the Repository as if it were content behind a regular FTP server. FTP Protocol Server works with standard FTP clients, including:

- Command line clients, such as the command line clients supplied with Unix and Windows Command Prompt
- Graphical clients, such as WS-FTP
- Web Browsers that support the FTP protocol

Figure 3-6, Figure 3-7, Figure 3-8, and Figure 3-9 show examples of how you can access the root level of the Repository using various of standard FTP clients.

Figure 3-6 Accessing the Repository Root Level from the Command Prompt Command Line



```
C:\WINNT\System32\cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>ftp
ftp> open xdbdemo 2100
Connected to xdbdemo.
220 mdrake-sun FTP Server (XDB/Oracle9i Enterprise Edition Release 9.2.0.0
eta) ready.
User (xdbdemo:(none)): scott
331 pass required for SCOTT
Password:
230 SCOTT logged in
ftp> ls -l
200 PORT Command successful
150 ASCII Data Connection
drw-r--r--  2 SYS      oracle      0 FEB 16 17:58 home
drw-r--r--  2 SYS      oracle      0 FEB 14 17:09 public
drw-r--r--  2 SYS      oracle      0 FEB 14 17:09 sys
-rw-r--r--  1 SYS      oracle      0 FEB 14 17:10 xdbconfig.xml
226 ASCII Transfer Complete
ftp: 250 bytes received in 0.033seconds 8.33Kbytes/sec.
ftp> quit
221 QUIT Goodbye.

C:\>
```

Figure 3-7 Accessing the Repository Root Level from m IE Browser Web Folder Menu

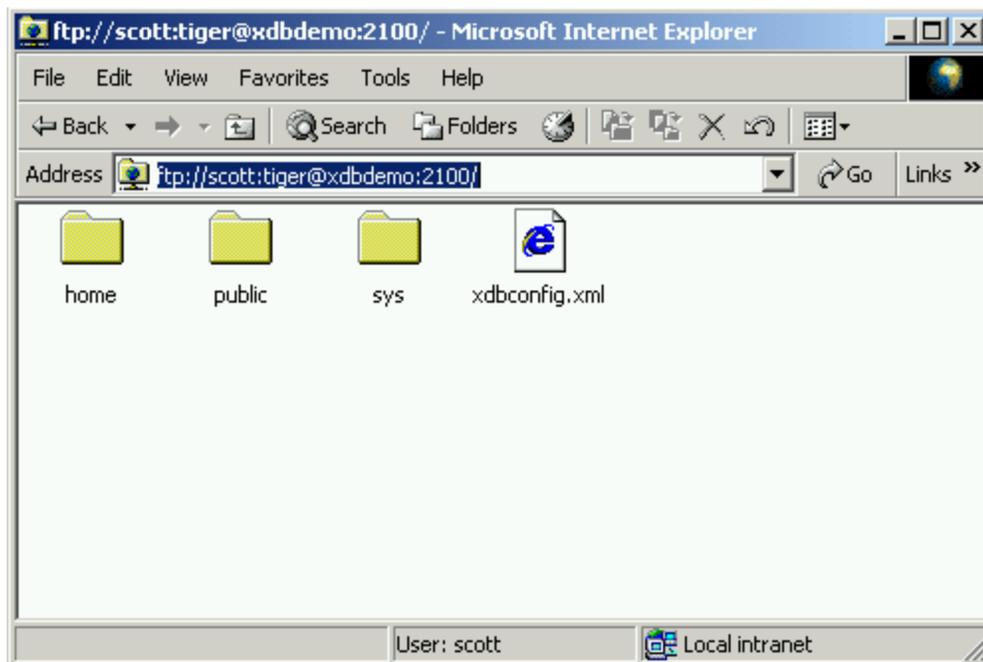


Figure 3–8 Accessing the Repository Root Level from m WS_FTP95LE FTP Interface Program

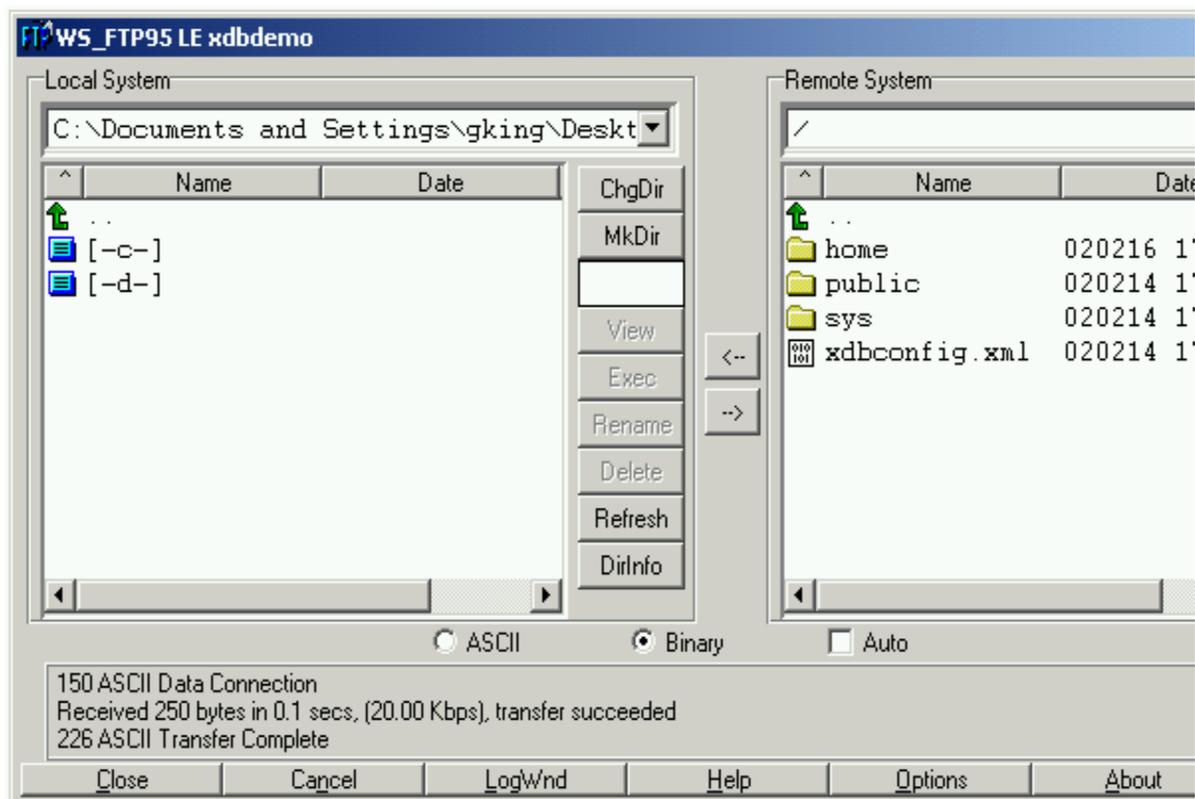
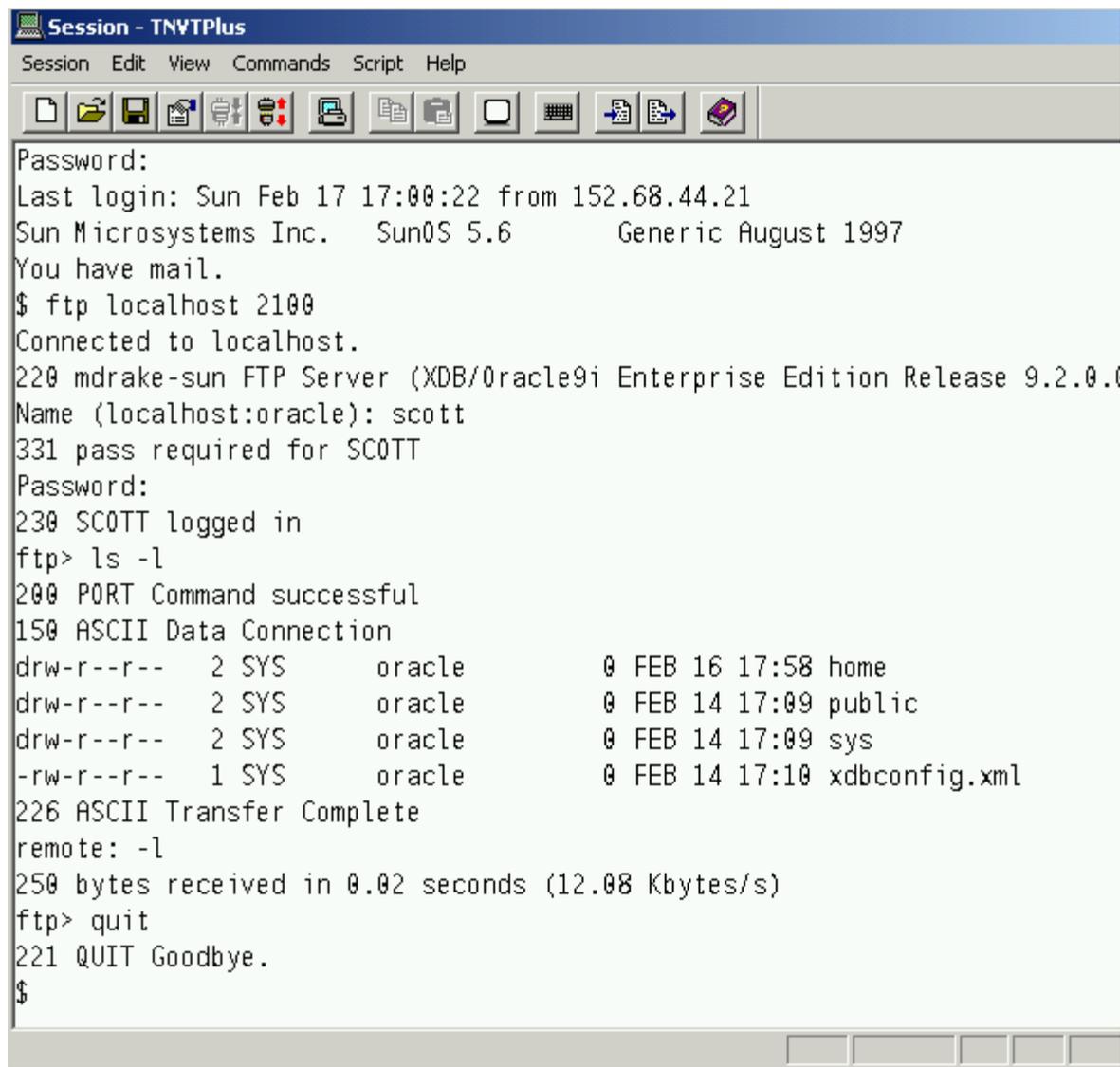


Figure 3–9 Accessing the Repository Root Level from a Telnet Session

```
Session - TNVTPlus
Session Edit View Commands Script Help
[Icons]
Password:
Last login: Sun Feb 17 17:00:22 from 152.68.44.21
Sun Microsystems Inc. SunOS 5.6 Generic August 1997
You have mail.
$ ftp localhost 2100
Connected to localhost.
220 mdrake-sun FTP Server (XDB/Oracle9i Enterprise Edition Release 9.2.0.0)
Name (localhost:oracle): scott
331 pass required for SCOTT
Password:
230 SCOTT logged in
ftp> ls -l
200 PORT Command successful
150 ASCII Data Connection
drw-r--r-- 2 SYS oracle 0 FEB 16 17:58 home
drw-r--r-- 2 SYS oracle 0 FEB 14 17:09 public
drw-r--r-- 2 SYS oracle 0 FEB 14 17:09 sys
-rw-r--r-- 1 SYS oracle 0 FEB 14 17:10 xdbconfig.xml
226 ASCII Transfer Complete
remote: -l
250 bytes received in 0.02 seconds (12.08 Kbytes/s)
ftp> quit
221 QUIT Goodbye.
$
```

Using HTTP/WebDAV Protocol Server

Oracle XML DB Repository can also be accessed using HTTP and WebDAV. WebDAV support allows applications such as a Microsoft's Web Folders client, Microsoft Office, and Macromedia's Dream weaver to directly access Oracle XML DB Repository. [Figure 3-10](#) and [Figure 3-11](#) are examples of using HTTP and WebDAV to access the Repository.

Figure 3-10 Accessing the Repository Using HTTP/WebDAV from Microsoft Windows Explorer

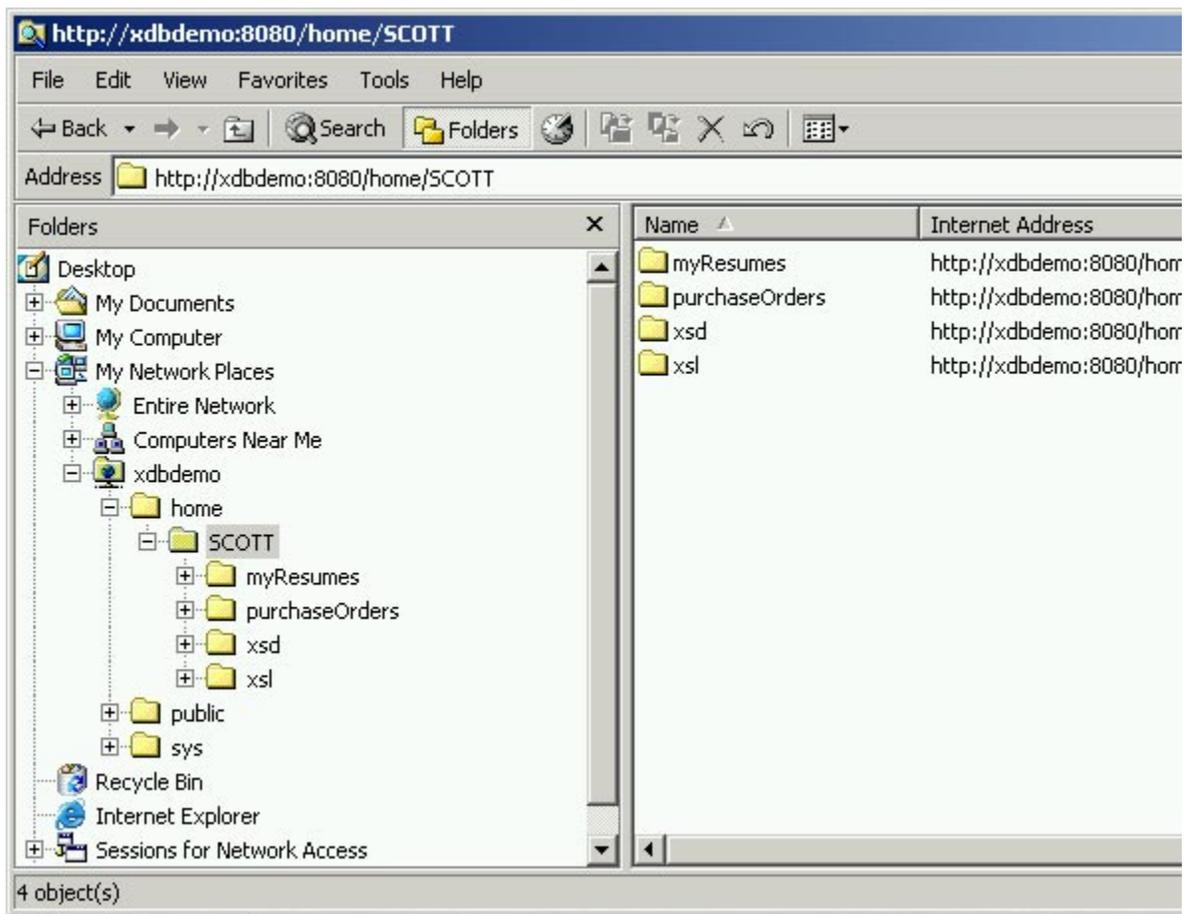
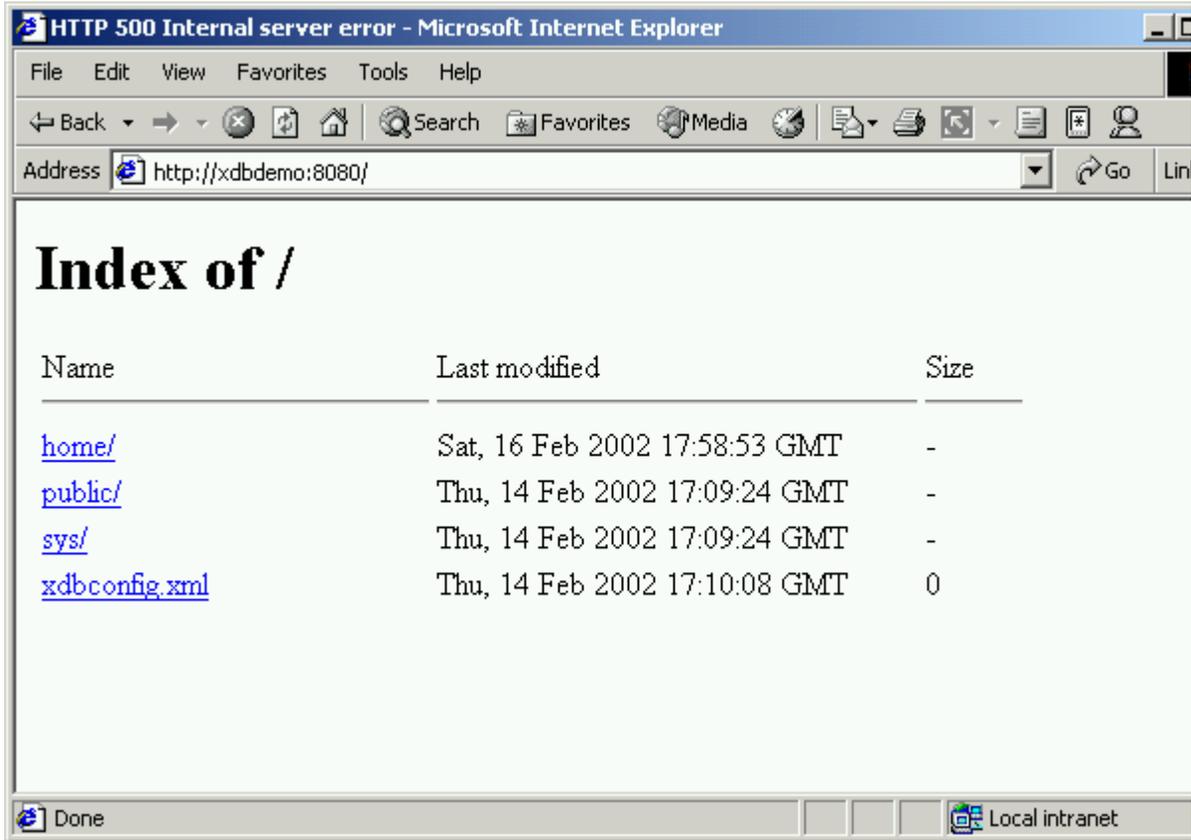


Figure 3–11 Accessing the Repository Using HTTP/WebDAV Protocol Server from Microsoft Web Folders Client



By providing support for standard industry protocols, Oracle XML DB makes it possible to upload and access data and documents stored in Oracle9i database using standard, familiar interfaces.

Part II

Storing and Retrieving XML Data in Oracle XML DB

Part II of this manual introduces you to ways you can store, retrieve, validate, and transform XML data using Oracle XML DB. It contains the following chapters:

- [Chapter 4, "Using XMLType"](#)
- [Chapter 5, "Structured Mapping of XMLType"](#)
- [Chapter 6, "Transforming and Validating XMLType Data"](#)
- [Chapter 7, "Searching XML Data with Oracle Text"](#)

Using XMLType

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

- [What Is XMLType?](#)
- [When to Use XMLType](#)
- [Storing XMLType Data in Oracle XML DB](#)
- [XMLType Member Functions](#)
- [How to Use the XMLType API](#)
- [Guidelines for Using XMLType Tables and Columns](#)
- [Manipulating XML Data in XMLType Columns/Tables](#)
- [Inserting XML Data into XMLType Columns/Tables](#)
- [Selecting and Querying XML Data](#)
- [Updating XML Instances and Data in Tables and Columns](#)
- [Deleting XML Data](#)
- [Using XMLType In Triggers](#)
- [Indexing XMLType Columns](#)

Note:

- *Non-schema-based:* XMLType tables and columns described in this chapter are not based on XML schema. You can, however, use the techniques and examples provided in this chapter regardless of which storage option you choose for your XMLType tables and columns. See [Chapter 3, "Using Oracle XML DB"](#) for further storage recommendations.
 - *XML schema-based:* [Appendix B, "XML Schema Primer"](#) and [Chapter 5, "Structured Mapping of XMLType"](#) describe how to work with XML schema-based XMLType tables and columns.
-
-

What Is XMLType?

Oracle9i Release 1 (9.0.1) introduced a new datatype, XMLType, to facilitate native handling of XML data in the database. The following summarizes XMLType:

- XMLType can be used in PL/SQL stored procedures as parameters, return values, and variables.
- XMLType can represent an XML document as an instance (of XMLType) in SQL.
- XMLType has built-in member functions that operate on XML content. For example, you can use XMLType functions to create, extract, and index XML data stored in Oracle9i database.
- Functionality is also available through a set of Application Program Interfaces (APIs) provided in PL/SQL and Java.

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

XMLType datatype can be used as the datatype of columns in tables and views. Variables of XMLType can be used in PL/SQL stored procedures as parameters, return values, and so on. You can also use XMLType in SQL, PL/SQL, and Java (through JDBC).

Note: In Oracle9i Release 1 (9.0.1), XMLType was only supported in the server in SQL, PL/SQL, and Java. In Oracle9i Release 2 (9.2), XMLType is also supported on the client side through SQL, Java, and protocols such as FTP and HTTP/WebDav.

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

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

See Also:

- ["Oracle XML DB Offers Faster Storage and Retrieval of Complex XML Documents"](#) on page 1-21
- *Oracle9i SQL Reference* Appendix D, "Using XML in SQL Statements"

Benefits of the XMLType Data Type and API

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

- *Versatile API.* XMLType has a versatile API for application development, as it includes built-in functions, indexing support, navigation, and so on.
- *XMLType and SQL.* You can use XMLType in SQL statements combined with other columns and datatypes. For example, you can query XMLType columns and join the result of the extraction with a relational column, and then Oracle can determine an optimal way to execute these queries.
- *Optimized evaluation using XMLType.* XMLType is optimized to not materialize the XML data into a tree structure unless needed. Therefore when SQL selects XMLType instances inside queries, only a serialized form is exchanged across function boundaries. These are exploded into tree format only when operations such as `extract()` and `existsNode()` are performed. The internal structure of XMLType is also an optimized DOM-like tree structure.

- *Indexing.* Oracle Text index has been enhanced to support XMLType columns. You can also create function-based indexes on `existsNode()` and `extract()` functions to speed up query evaluation.

See Also: [Chapter 10, "Generating XML Data from the Database"](#)

When to Use XMLType

Use XMLType when you need to perform the following:

- SQL queries on part of or the whole XML document: The functions `existsNode()` and `extract()` provide the necessary SQL query functions over XML documents.
- Strong typing inside SQL statements and PL/SQL functions: Strong typing implies that you ensure that the values passed in are XML values and not any arbitrary text string.
- XPath functionality provided by `extract()` and `existsNode()` functions: Note that XMLType uses the built-in C XML parser and processor and hence provides better performance and scalability when used inside the server.
- Indexing on XPath searches on documents: XMLType has member functions that you can use to create function-based indexes to optimize searches.
- To shield applications from storage models. Using XMLType instead of CLOBs or relational storage allows applications to gracefully move to various storage alternatives later without affecting any of the query or DML statements in the application.
- To prepare for future optimizations. New XML functionality will support XMLType. Since Oracle9i database is natively aware that XMLType can store XML data, better optimizations and indexing techniques can be done. By writing applications to use XMLType, these optimizations and enhancements can be easily achieved and preserved in future releases without your needing to rewrite applications.

Storing XMLType Data in Oracle XML DB

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

- *In Large Objects (LOBs).* LOB storage maintains content accuracy to the original XML (whitespaces and all). Here the XML documents are stored composed as whole documents like files. In this release, for non-schema-based storage, XMLType offers a CLOB storage option. In future releases, Oracle may

provide other storage options, such as BLOBs, NCLOBs, and so on. You can also create a CLOB-based storage for XML schema-based storage.

When you create an `XMLType` column without any XML schema specification, a hidden CLOB column is automatically created to store the XML data. The `XMLType` column itself becomes a virtual column over this hidden CLOB column. It is not possible to directly access the CLOB column; however, you can set the storage characteristics for the column using the `XMLType` storage clause.

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

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

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

Pros and Cons of XML Storage Options in Oracle XML DB

[Table 4-1](#) summarizes some advantages and disadvantages to consider when selecting your Oracle XML DB storage option.

Table 4–1 XML Storage Options in Oracle XML DB

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

When to Use CLOB Storage for XMLType

Use CLOB storage for XMLType in the following cases:

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

Note: XMLType and Varray:

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

See Also:

- [Chapter 2, "Getting Started with Oracle XML DB"](#)
- [Chapter 3, "Using Oracle XML DB", "Storing XML: Structured or Unstructured Storage"](#) on page 3-24
- [Chapter 10, "Generating XML Data from the Database"](#), for information on how to generate XMLType data.

XMLType Member Functions

Oracle9i Release 1 (9.0.1) introduced several SQL functions and XMLType member functions that operate on XMLType values. Oracle9i Release 2 (9.2) has expanded functionality. It provides several new SQL functions and XMLType member functions.

See Also:

- [Appendix F, "Oracle XML DB XMLType API, PL/SQL and Resource PL/SQL APIs: Quick Reference"](#)
- *Oracle9i XML API Reference - XDK and Oracle XML DB* for a list of all XMLType and member functions, their syntax, and descriptions.

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

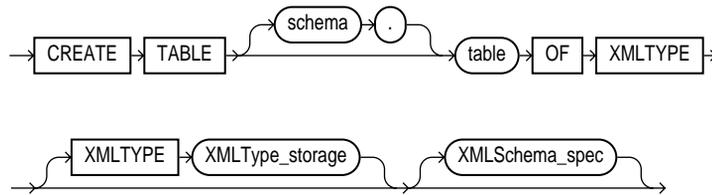
See Also: [Appendix C, "XPath and Namespace Primer"](#)

How to Use the XMLType API

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

[Figure 4-1](#) shows the syntax for creating an XMLType table:

```
CREATE TABLE [schema.] table OF XMLTYPE
  [XMLTYPE XMLType_storage] [XMLSchema_spec];
```

Figure 4–1 Creating an XMLType Table

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

Creating, Adding, and Dropping XMLType Columns

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

Example 4–1 Creating XMLType: Creating XMLType Columns

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

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

Example 4–2 Creating XMLType: Creating XMLType Columns

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

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

```
CREATE TABLE po_xtab of XMLType; -- this creates a table of XMLType. The default
-- is CLOB based storage.
```

Example 4–3 Adding XMLType Columns

You can alter tables to add XMLType columns as well. This is similar to any other datatype. The following statement adds a new customer document column to the table:

```
ALTER TABLE po_xml_tab add (custDoc XMLType);
```

Example 4–4 Dropping XMLType Columns

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

```
ALTER TABLE po_xml_tab drop (custDoc);
```

Inserting Values into an XMLType Column

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

Example 4–5 Inserting into XMLType Using the XMLType() Constructor

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

```
INSERT INTO warehouses VALUES
  (      100, XMLType(
    '<Warehouse whNo="100">
      <Building>Owned</Building>
    </Warehouse>'), 'Tower Records', 1003);
```

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

Using XMLType in an SQL Statement

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

Example 4–6 Using XMLType and in a SELECT Statement

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

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

```
Building
-----
```

Owned

See Also: ["How to Use the XMLType API"](#) on page 4-7.

Updating an XMLType Column

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

Example 4-7 Updating XMLType

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

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

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

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

Deleting a Row Containing an XMLType Column

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

Example 4-8 Deleting an XMLType Column Row

You can use extract() and existsNode() functions to identify rows to delete as well. For example to delete all warehouse rows for which the warehouse building is leased, you can write a statement such as:

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

Note: In this release, Oracle supports `XMLType` as a public synonym for `sys.XMLType`. `XMLType` now also supports a set of user-defined constructors (mirroring the `createXML` static functions). For example:

- In Oracle9i Release 1 (9.0.1), you could use the following syntax: `sys.XMLType.createXML(' <Warehouse whNo="100">...')`
 - In Oracle9i Release 2 (9.2), you can use the following abbreviated version: `XMLType(' <Warehouse whNo="100">...')`.
-
-

Guidelines for Using XMLType Tables and Columns

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

Define table/column of XMLType

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

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

Create an XMLType Instance

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

See Also: ["SYS_XMLGEN\(\): Converting an XMLType Instance"](#) on page 10-47, for an example.

Select or Extract a Particular XMLType Instance

You can select out the `XMLType` instance from the column. `XMLType` also offers a choice of member functions, such as `extract()` and `existsNode()`, to extract a particular node and to check to see if a node exists respectively. See the table of `XMLType` member functions in *Oracle9i XML API Reference - XDK and Oracle XML DB*.

See Also:

- ["Selecting XMLType Columns using getClobVal\(\)"](#) on page 4-18
- ["Extracting Fragments from XMLType Using extract\(\)"](#) on page 4-30

You can Define an Oracle Text Index

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

You Can Define XPath Index, CTXXPATH

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

See Also:

- ["Indexing XMLType Columns"](#) on page 4-38
- [Chapter 7, "Searching XML Data with Oracle Text"](#)
- [Chapter 10, "Generating XML Data from the Database"](#)
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)*

Specifying Storage Characteristics on XMLType Columns

XML data in an XMLType column can be stored as a CLOB column. Hence you can also specify LOB storage characteristics for that column. In example, ["Creating XMLType: Creating XMLType Columns"](#) on page 4-8, the warehouse_spec column is an XMLType column.

Example 4–9 Specifying Storage When Creating an XMLType Table

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

```
CREATE TABLE po_xml_tab(  
    poid NUMBER(10),  
    poDoc XMLTYPE  
)  
XMLType COLUMN poDoc
```

```

STORE AS CLOB (
    TABLESPACE lob_seg_ts
    STORAGE (INITIAL 4096 NEXT 4096)
    CHUNK 4096 NOCACHE LOGGING
);

```

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

Example 4–10 Adding an XMLType Columns and Specifying Storage

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

```

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

```

Changing Storage Options on an XMLType Column Using XMLData

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

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

For example, consider table `foo_tab`:

```
CREATE TABLE foo_tab (a xmltype);
```

To change the storage characteristics of LOB column `a` in `foo_tab`, you can use the following statement:

```
ALTER TABLE foo_tab MODIFY LOB (a.xmldata) (storage (next 5K) cache);
```

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

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

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

See also: *Oracle9i Application Developer's Guide - Large Objects (LOBs)* f and *Oracle9i SQL Reference* for more information about LOB storage options

Specifying Constraints on XMLType Columns

You can specify NOT NULL constraint on an XMLType column.

Example 4–12 *Specifying Constraints on XMLType Columns*

```
CREATE TABLE po_xml_tab (  
  poId number(10),  
  poDoc XMLType NOT NULL  
);
```

prevents inserts such as:

```
INSERT INTO po_xml_tab (poDoc) VALUES (null);
```

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

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

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

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

Manipulating XML Data in XMLType Columns/Tables

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

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

- [Inserting XML Data into XMLType Columns/Tables](#)
- [Selecting and Querying XML Data](#)
- [Updating XML Instances and Data in Tables and Columns](#)
- [Deleting XML Data](#)

Inserting XML Data into XMLType Columns/Tables

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

- By using the `INSERT` statement (in SQL, PL/SQL, and Java)
- By using `SQL*Loader`. See [Chapter 22, "Loading XML Data into Oracle XML DB"](#)

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

Using INSERT Statements

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

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

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

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

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

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

Example 4–15 Inserting XML Data Using an XMLType Instance

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

```
INSERT INTO po_xml_tab
SELECT poid, XMLType(poClob)
FROM po_clob_tab;
```

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

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

This example inserts a purchase order into table `po_tab` using the `XMLType` constructor:

```
INSERT INTO po_xml_tab
VALUES(100, XMLType('<?xml version="1.0"?>
<PO pono="1">
  <PNAME>Po_1</PNAME>
  <CUSTNAME>John</CUSTNAME>
  <SHIPADDR>
    <STREET>1033, Main Street</STREET>
    <CITY>Sunnyvalue</CITY>
    <STATE>CA</STATE>
  </SHIPADDR>
</PO>'));)
```

Example 4–17 Inserting XML Data Using XMLElement()

This example inserts a purchase order into table `po_xml_tab` by generating it using the `XMLElement()` SQL function. Assume that the purchase order is an object view that contains a purchase order object. The whole definition of the purchase order view is given in ["DBMS_XMLGEN: Generating a Purchase Order from the Database in XML Format"](#) on page 10-33.

```
INSERT INTO po_xml_tab
  SELECT XMLElement("po", value(p))
         FROM po p
         WHERE p.pono=2001;
```

`XMLElement()` creates an `XMLType` from the purchase order object, which is then inserted into table `po_xml_tab`. You can also use `SYS_XMLGEN()` in the `INSERT` statement.

Selecting and Querying XML Data

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

- By selecting `XMLType` columns through SQL, PL/SQL, or Java
- By querying `XMLType` columns directly and using `extract()` and `existsNode()`
- By using Oracle Text operators to query the XML content. See ["Indexing XMLType Columns"](#) on page 4-38 and [Chapter 7, "Searching XML Data with Oracle Text"](#).

SQL Functions for Manipulating XML data

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

Selecting XML Data

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

Example 4–18 Selecting XMLType Columns using getClobVal()

This example shows how to select an XMLType column using SQL*Plus:

```
SET long 2000

SELECT e.poDoc.getClobval() AS poXML
FROM po_xml_tab e;
```

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

Querying XML Data

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

Using XPath Expressions for Searching XML Documents

XPath is a W3C recommendation for navigating XML documents. XPath models the XML document as a tree of nodes. It provides a rich set of operations to “walk” the tree and to apply predicates and node test functions. Applying an XPath expression to an XML document can result in a set of nodes. For instance, `/PO/PONO` selects out all “PONO” child elements under the “PO” root element of the document.

[Table 4–2](#) lists some common constructs used in XPath.

Table 4–2 Some Common XPath Constructs

XPath Construct	Description
“/”	Denotes the root of the tree in an XPath expression. For example, /PO refers to the child of the root node whose name is “PO”.
“/”	Also used as a path separator to identify the children node of any given node. For example, /PO/PNAME identifies the purchase order name element, a child of the root element.
“//”	Used to identify all descendants of the current node. For example, PO//ZIP matches any zip code element under the “PO” element.
“*”	Used as a wildcard to match any child node. For example, /PO/*/STREET matches any street element that is a grandchild of the “PO” element.
[]	Used to denote predicate expressions. XPath supports a rich list of binary operators such as OR, AND, and NOT. For example, /PO[PONO=20 and PNAME="PO_2"]/SHIPADDR select out the shipping address element of all purchase orders whose purchase order number is 20 and whose purchase order name is “PO_2”. [] is also used for denoting an index into a list. For example, /PO/PONO[2] identifies the second purchase order number element under the “PO” root element.

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

See Also: [Appendix C, "XPath and Namespace Primer"](#)

Querying XML Data Using XMLType Member Functions

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

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

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

```
set long 2000

SELECT e.poDoc.getClobval() AS poXML
FROM po_xml_tab e
WHERE e.poDoc.existsNode('/PO[PNAME = "po_2"]') = 1;
```

```

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

```

existsNode Function

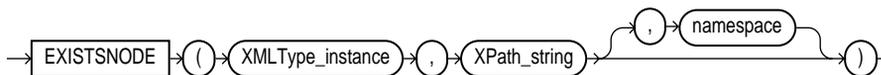
The syntax for the `existsNode()` function is described in [Figure 4-2](#) and also as follows:

```

existsNode(XMLType_instance IN XMLType,
           XPath_string IN VARCHAR2, namespace_string IN varchar2 := null)
RETURN NUMBER

```

Figure 4-2 *existsNode()* Syntax



`existsNode()` function on `XMLType` checks if the given XPath evaluation results in at least a single XML element or text node. If so, it returns the numeric value 1, otherwise, it returns a 0. Namespace can be used to identify the mapping of prefix(es) specified in the `XPath_string` to the corresponding namespace(s).

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

For example, consider an XML document such as:

```

<PO>
  <PONO>100</PONO>
  <PNAME>Po_1</PNAME>
  <CUSTOMER CUSTNAME="John" />
  <SHIPADDR>
    <STREET>1033, Main Street</STREET>
    <CITY>Sunnyvalue</CITY>

```

```

    <STATE>CA</STATE>
  </SHIPADDR>
</PO>

```

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

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

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

Example 4–21 Using existsNode() to Find a node

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

```

SELECT warehouse_id, EXISTSNODE(warehouse_spec, '/Warehouse/Docks')
   "Loading Docks"
FROM warehouses
WHERE warehouse_spec IS NOT NULL;

```

WAREHOUSE_ID	Loading Docks
1	1
2	1
3	0
4	1

Using Indexes to Evaluate existsNode()

You can create functional indexes using `existsNode()` to speed up the execution. You can also create a `CTXXPATH` index to help speed up arbitrary XPath searching.

See Also: ["Creating XPath Indexes on XMLType Columns: CTXXPATH Index"](#) on page 4-41

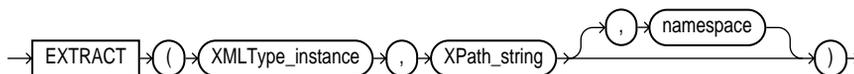
extract () Function

The `extract()` function is similar to the `existsNode()` function. It applies a `VARCHAR2` XPath string with an optional namespace parameter and returns an

XMLType instance containing an XML fragment. The syntax is described in Figure 4-3 and as follows:

```
extract(XMLType_instance IN XMLType, XPath_string IN VARCHAR2,
        namespace_string In varchar2 := null) RETURN XMLType;
```

Figure 4-3 *extract()* Syntax



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

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

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

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

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

For example, XPath expressions:

- `/PO/PNAME` identifies the fragment `<PNAME>PO_1</PNAME>`
- `/PO/PNAME/text()` identifies the text value “PO_1”

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

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

you can use:

- //PONO[1] to identify the first “PONO” element (with value 100).
- //PONO[2] to identify the second “PONO” element (with value 200).

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

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

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

Example 4–22 Using `extract()` to Extract the Value of a Node

This example extracts the value of node, `/Warehouse/Docks`, of column, `warehouse_spec` in table `oe.warehouses`:

```
SELECT warehouse_name,
       extract(warehouse_spec, '/Warehouse/Docks').getStringVal()
       "Number of Docks"
FROM warehouses
WHERE warehouse_spec IS NOT NULL;
```

WAREHOUSE_NAME	Number of Docks
-----	-----
Southlake, Texas	<Docks>2</Docks>
San Francisco	<Docks>1</Docks>
New Jersey	<Docks/>
Seattle, Washington	<Docks>3</Docks>

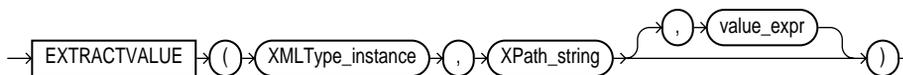
extractValue() Function

The `extractValue()` function takes as arguments an `XMLType` instance and an XPath expression. It returns a scalar value corresponding to the result of the XPath evaluation on the `XMLType` instance. `extractValue()` syntax is also described in Figure 4–4.

- **XML schema-based documents.** For documents based on XML schema, if Oracle9i can infer the type of the return value, then a scalar value of the appropriate type is returned. Otherwise, the result is of type `VARCHAR2`.
- **Non- schema-based documents.** For documents not based on XML schemas, the return type is always `VARCHAR2`.

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

Figure 4–4 `extractValue()` Syntax



A Shortcut Function

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

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

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

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

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

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

extractValue() Characteristics

`extractValue()` has the following characteristics:

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

```
extractValue(xmlinstance, '/PO/PNAME')
```

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

```
extract(xmlinstance, '/PO/PNAME/text()').getStringval()
```

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

The following example takes as input the same arguments as the example for [extract \(\) Function](#) on page 4-21. Instead of returning an XML fragment, as `extract()` does, it returns the scalar value of the XML fragment:

```
SELECT warehouse_name,
       extractValue(e.warehouse_spec, '/Warehouse/Docks')
       "Docks"
FROM warehouses e
WHERE warehouse_spec IS NOT NULL;
```

WAREHOUSE_NAME	Docks
-----	-----
Southlake, Texas	2
San Francisco	1
New Jersey	
Seattle, Washington	3

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

```
extract(e.warehouse_spec, '/Warehouse/Docks/text()').getStringval()
```

More SQL Examples That Query XML

The following SQL examples illustrate ways you can query XML.

Example 4–24 Querying XMLType Using extract() and existsNode()

Assume the `po_xml_tab` table, which contains the purchase order identification and the purchase order XML columns, and assume that the following values are inserted into the table:

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

```
INSERT INTO po_xml_tab values (200,
    xmltype('<?xml version="1.0"?>
        <PO>
            <PONAME>PO_1</PONAME>
        </PO>'));
```

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

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

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

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

See Also: ["XMLType Member Functions"](#) on page 4-7

Example 4–25 Querying Transient XMLType Data

The following example shows how you can select out the XML data and query it inside PL/SQL: create a transient instance from the purchase order table and then perform some extraction on it. Assume `po_xml_tab` contains the data shown in [Example 4–16, "Inserting XML Data Using XMLType\(\) with String"](#), modified:

```
set serverout on
declare
    poxml XMLType;
    cust XMLType;
    val VARCHAR2(200);
begin

    -- select the adt instance
    select poDoc into poxml
        from po_xml_tab p where p.poid = 100;

    -- do some traversals and print the output
    cust := poxml.extract('//SHIPADDR');

    -- do something with the customer XML fragment
    val := cust.getStringVal();
    dbms_output.put_line(' The customer XML value is ' || val);

end;
/
```

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

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

```
CREATE TABLE cust_tab
(
    custid number primary key,
    custname varchar2(20)
);

INSERT INTO cust_tab values (1001, 'John Nike');
```

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

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

```
<?xml version = '1.0'?>
<PO>
  <PONO>2001</PONO>
  <PNAME>Po_1</PNAME>
  <CUSTOMER CUSTNAME="John Nike"/>
  <SHIPADDR>
    <STREET>323 College Drive</STREET>
    <CITY>Edison</CITY>
    <STATE>NJ</STATE>
    <ZIP>08820</ZIP>
  </SHIPADDR>
</PO>
```

into the relational tables, using `extract()`.

Here is an SQL example assuming that the XML described in the previous example is present in the `po_xml_tab`:

```
INSERT INTO po_rel_tab
SELECT p.poDoc.extract('/PO/PONO/text()').getnumberval() as pono,
       p.poDoc.extract('/PO/PNAME/text()').getstringval() as pname,
       -- get the customer id corresponding to the customer name
       ( SELECT c.custid
         FROM cust_tab c
         WHERE c.custname = p.poDoc.extract('/PO/CUSTOMER/@CUSTNAME').getstringval()
       ) as custid,
       p.poDoc.extract('/PO/SHIPADDR/STREET/text()').getstringval() as shipstreetr,
       p.poDoc.extract('/PO/CITY/text()').getstringval() as shipcity,
       p.poDoc.extract('/PO/ZIP/text()').getstringval() as shipzip
FROM po_xml_tab p;
```

Table `po_rel_tab` should now have the following values:

PONO	PNAME	CUSTID	SHIPSTREET	SHIPCITY	SHIPZIP
2001	Po_1	1001	323 College Drive	Edison	08820

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

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

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

```

DECLARE
    poxml XMLType;
    cname varchar2(200);
    pono number;
    pname varchar2(100);
    shipstreet varchar2(100);
    shipcity varchar2(30);
    shipzip varchar2(20);

BEGIN

    -- select the adt instance
    SELECT poDoc INTO poxml FROM po_xml_tab p;

    cname := poxml.extract('//CUSTOMER/@CUSTNAME').getstringval();

    pono := poxml.extract('/PO/PONO/text()').getnumberval();
    pname := poxml.extract('/PO/PNAME/text()').getstringval();
    shipstreet := poxml.extract('/PO/SHIPADDR/STREET/text()').getstringval();
    shipcity := poxml.extract('//CITY/text()').getstringval();
    shipzip := poxml.extract('//ZIP/text()').getstringval();

    INSERT INTO po_rel_tab
        VALUES (pono, pname,
                (SELECT custid FROM cust_tab c WHERE custname = cname),
                shipstreet, shipcity, shipzip);

END;
/

```

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

Using `extract()` and `existsNode()` functions, you can perform a variety of search operations on the column, as follows:

```
SELECT e.poDoc.extract('/PO/PNAME/text()').getStringVal() PNAME
FROM po_xml_tab e
WHERE e.poDoc.existsNode('/PO/SHIPADDR') = 1 AND
      e.poDoc.extract('///PONO/text()').getNumberVal() = 300 AND
      e.poDoc.extract('///@CUSTNAME').getStringVal() like '%John%';
```

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

Example 4–29 Searching XML Data with extractValue()

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

```
SELECT extractvalue(e.poDoc, '/PO/PNAME') PNAME
FROM po_xml_tab e
WHERE e.poDoc.existsNode('/PO/SHIPADDR') = 1 AND
      extractvalue(e.poDoc, '//PONO') = 300 AND
      extractvalue(e.poDoc, '//@CUSTNAME') like '%John%';
```

Example 4–30 Extracting Fragments from XMLType Using extract()

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

```
SELECT e.poDoc.extract('/PO/SHIPADDR/STATE').isFragment()
FROM po_xml_tab e;
```

Note: You cannot insert fragments into `XMLType` columns. You can use `SYS_XMLGEN()` to convert a fragment into a well-formed document by adding an enclosing tag. See "[SYS_XMLGEN\(\) Function](#)" on page 10-41. You can, however, query further on the fragment using the various `XMLType` functions.

The previous SQL statement returns 0, since the extraction `/PO/SHIPADDR/STATE` returns a singleton well-formed node which is not a fragment.

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

Updating XML Instances and Data in Tables and Columns

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

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

updateXML() SQL Function

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

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

```
UPDATEXML(xmlinstance, xpath1, value_expr1
          [, xpath2, value_expr2]...[,namespace_string]);
```

Example 4–31 Updating XMLType Using the UPDATE Statement

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

```
UPDATE po_xml_tab e
SET e.poDoc = XMLType(
'<?xml version="1.0"?>
<PO pono="2">
  <PNAME>Po_2</PNAME>
  <CUSTNAME>Nance</CUSTNAME>
  <SHIPADDR>
    <STREET>2 Avocet Drive</STREET>
    <CITY>Redwood Shores</CITY>
    <STATE>CA</STATE>
```

```

        </SHIPADDR>
    </PO>')
WHERE e.poDoc.EXTRACT('/PO/PONO/text()').getNumberVal() = 2001;

```

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

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

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

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

```

UPDATE po_xml_tab
SET poDoc = UPDATEXML(poDoc,
    '/PO/CUSTNAME/text()', 'John');

```

1 row updated

```

SELECT e.poDoc.getstringval() AS newpo
FROM po_xml_tab e;

```

NEWPO

```

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

```

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

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

```
UPDATE emp_tab e
SET e.emp_col = UPDATEXML(e.emp_col,
    '/EMPLOYEES/EMP[EMPNAME="Joe"]/SALARY/text()',100000,
    '//EMP[EMPNAME="Jack"]/EMPNAME/text()', 'Jackson',
    '//EMP[EMPNO=217]',XMLTYPE.CREATEXML(
        '<EMP><EMPNO>217</EMPNO><EMPNAME>Jane</EMPNAME></EMP>'))
WHERE EXISTSNODE(e.emp_col, '//EMP') = 1;
```

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

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

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

Note: This example only selects the document and the update occurs on a transient `XMLType` instance. The original document is not affected.

```
SELECT
    UPDATEXML(poDoc,
        '/PO/CUSTNAME/text()', 'John').getStringval() AS updatedPO
FROM po_xml_tab;
```

UPDATEDPO

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

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

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

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

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

```
SELECT UPDATEXML(emp_col, ' /EMPLOYEES/EMP[EMPNAME="Joe"] /SALARY/text()', 100000,
                  '//EMP[EMPNAME="Jack"] /EMPNAME/text()', 'Jackson',
                  '//EMP[EMPNO=217]',
                  XMLTYPE.CREATEXML('<EMP><EMPNO>217</EMPNO><EMPNAME>Jane</EMPNAME>'))
FROM emp_tab e;
```

This generates the following updated XML:

```
<EMPLOYEES>
  <EMP>
    <EMPNO>112</EMPNO>
    <EMPNAME>Joe</EMPNAME>
    <SALARY>100000</SALARY>
  </EMP>
  <EMP>
    <EMPNO>217</EMPNO>
    <EMPNAME>Jane</EMPNAME>
  </EMP>
```

```

<EMP>
  <EMPNO>412</EMPNO>
  <EMPNAME>Jackson</EMPNAME>
  <SALARY>40000</SALARY>
</EMP>
</EMPLOYEES>

```

Creating Views of XML Data with updateXML()

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

Example 4-36 Creating Views Using updateXML()

A view such as:

```

CREATE VIEW new_emp_view
AS SELECT
  UPDATEXML(emp_col, '/EMPLOYEES/EMP/SALARY/text()', 0) emp_view_col
FROM emp_tab e;

```

ensures that users selecting from view, `new_emp_view`, do not see the SALARY field for any employee.

updateXML() and NULL Values

`UpdateXML()` treats NULL values by mapping them to non-existent attribute, element, or text values. For instance if you update node, `'//empno/text()'` with a NULL value, it is treated as if element `empno` is being removed. Setting an attribute to NULL removes the attribute. There are exceptions to this. The section, "[NULL Updates When Object Types Are Generated by XML Schema Registration](#)" on page 4-36 discusses this further.

In this case, when you update an element and pass a NULL value to it, the attributes and children of the element disappear, and the element becomes empty. A NULL value for an element update is equivalent to setting the element to empty.

Note: Setting `'//empno'` to NULL has the same effect as setting `'//empno/text()'` to NULL.

NULL Updates When Object Types Are Generated by XML Schema Registration

NULL updates remove the element except when DOM fidelity is not maintained.

Example 4–37 NULL Updates with updateXML()

Consider the XML document:

```
<PO>
  <pono>21</pono>
  <shipAddr gate="xxx">
    <street>333</street>
    <city>333</city>
  </shipAddr>
</PO>
```

The clause:

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

is equivalent to making it:

```
<PO>
  <pono>21</pono>
  <shipAddr/>
</PO>
```

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

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

results in:

```
<PO>
  <pono>21</pono>
  <shipAddr>
    <street/>
    <city>333</city>
  </shipAddr>
</PO>
```

Updating the Same XML Node More Than Once

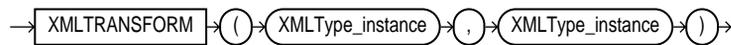
You can update the same XML node more than once in the `updateXML()` statement. For example, you can update both `/EMP[EMPNO=217]` and `/EMP[EMPNAME="Jane"]/EMPNO`, where the first XPath identifies the EMPNO node containing it as well. The order of updates is determined by the order of the XPath

expressions in left-to-right order. Each successive XPath works on the result of the previous XPath update.

XMLTransform() Function

The `XMLTransform()` function takes in an `XMLType` instance and an XSLT stylesheet. It applies the stylesheet to the XML document and returns a transformed XML instance. See [Figure 4-5](#).

Figure 4-5 *XMLTransform() Syntax*



`XMLTransform()` is explained in detail in [Chapter 6, "Transforming and Validating XMLType Data"](#).

Deleting XML Data

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

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

For example, to delete all purchase order rows with a purchase order name of "Po_2", execute a statement such as:

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

Using XMLType In Triggers

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

Example 4-39 *Creating XMLType Triggers*

For example, you can write a trigger to change the purchase order if it does not contain a shipping address:

```
CREATE OR REPLACE TRIGGER po_trigger
```

```

BEFORE INSERT OR UPDATE ON po_xml_tab FOR EACH ROW
declare
  pono Number;
begin

  if inserting then:

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

```

when updating, if the old poDoc has purchase order number different from the new one then make it an invalid PO.

if updating then:

```

    if :OLD.poDoc.extract('//PONO/text()').getNumberVal() !=
      :NEW.poDoc.extract('//PONO/text()').getNumberVal() then

      :NEW.poDoc := xmltype('<PO>INVALID_PO</PO>');
    end if;
  end if;
end;
/

```

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

Indexing XMLType Columns

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

Creating Function-Based Indexes on XMLType Columns

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

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

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

```

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

```

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

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

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

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

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

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

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

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

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

This speeds up the query processing.

Creating Oracle Text Indexes on XMLType Columns

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

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

Example 4–42 Creating an Oracle Text Index

```
CREATE INDEX po_text_index ON
```

```
po_xml_tab(poDoc) indextype is ctxsys.context;
```

You can also perform Oracle Text operations such as `CONTAINS` and `SCORE`, on `XMLType` columns. In Oracle9i Release (9.0.1), the `CONTAINS` operator was enhanced to support XPath using two new operators, `INPATH` and `HASPATH`:

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

Example 4–43 Searching XML Data Using HASPATH

For example:

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

QUERY_REWRITE PRIVILEGE Is No Longer Needed

In Oracle9i Release (9.0.1), to create and use Oracle Text index in queries, in addition to having the privileges for creating indexes and for creating Oracle Text indexes, you also needed privileges and settings for creating function-based indexes:

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

Oracle Text index uses the `PATH_SECTION_GROUP` as the default section group when indexing `XMLType` columns. This default can be overridden during Oracle Text index creation.

With this release, you no longer need the additional `QUERY_REWRITE` privileges when creating Oracle Text indexes.

See Also:

- [Chapter 7, "Searching XML Data with Oracle Text"](#)
- [Chapter 10, "Generating XML Data from the Database"](#)
- *Oracle Text Reference*
- *Oracle Text Application Developer's Guide*

Note: The `QUERY_REWRITE_INTEGRITY` and `QUERY_REWRITE_ENABLED` session settings are no longer needed to create Oracle Text or other function-based indexes on XMLType columns.

Creating XPath Indexes on XMLType Columns: CTXXPATH Index

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

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

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

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

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

For example, a query such as:

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

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

See Also:

- [Chapter 7, "Searching XML Data with Oracle Text"](#)
- [Chapter 10, "Generating XML Data from the Database"](#)

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

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

- Since Oracle Text index ignores spaces, the XPath expression may not yield accurate results when spaces are significant.

- Oracle Text index also supports certain predicate expressions with string equality, but cannot support numerical and range comparisons.
- Oracle Text index may give wrong results if the XML document only has tag names and attribute names without any text. For example, consider the following XML document:

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

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

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

See Also: [Chapter 7, "Searching XML Data with Oracle Text"](#), [Table 7-6, "Using CONTAINS\(\) and existsNode\(\) to Search XMLType Data"](#) on page 7-38

Structured Mapping of XMLType

This chapter introduces XML Schema and explains how XML schema is used in Oracle XML DB applications. It describes how to register your XML schema, create storage structures for storing schema-based XML, and generate Java Beans to access and manipulate data in Java applications.

It explains in detail the mapping from XML to SQL storage types, including techniques for maintaining the DOM fidelity of XML data. This chapter also describes how queries over `XMLType` tables and columns based on this mapping are optimized using query rewrite techniques. It discusses the mechanism for generating XML schemas from existing object types.

This chapter contains the following sections:

- [Introducing XML Schema](#)
- [XML Schema and Oracle XML DB](#)
- [Using Oracle XML DB and XML Schema](#)
- [Introducing DBMS_XMLSCHEMA](#)
- [Registering Your XML Schema Before Using Oracle XML DB](#)
- [Deleting Your XML Schema Using DBMS_XMLSCHEMA](#)
- [Guidelines for Using Registered XML Schemas](#)
- [Java Bean Generation During XML Schema Registration](#)
- [Generating XML Schema Using DBMS_XMLSCHEMA.generateSchema\(\)](#)
- [XML Schema-Related Methods of XMLType](#)
- [Managing and Storing XML Schema](#)
- [DOM Fidelity](#)

-
- Creating XMLType Tables and Columns Based on XML Schema
 - Specifying SQL Object Type Names with SQLName, SQLType Attributes
 - Mapping of Types Using DBMS_XMLSCHEMA
 - XML Schema: Mapping SimpleTypes to SQL
 - XML Schema: Mapping complexTypes to SQL
 - Oracle XML DB complexType Extensions and Restrictions
 - Further Guidelines for Creating XML Schema-Based XML Tables
 - Query Rewrite with XML Schema-Based Structured Storage
 - Creating Default Tables During XML Schema Registration
 - Ordered Collections in Tables (OCTs)
 - Cyclical References Between XML Schemas

Introducing XML Schema

The XML Schema Recommendation was created by the World Wide Web Consortium (W3C) to describe the content and structure of XML documents in XML. It includes the full capabilities of Document Type Definitions (DTDs) so that existing DTDs can be converted to XML schema. XML schemas have additional capabilities compared to DTDs.

See Also: [Appendix B, "XML Schema Primer"](#)

XML Schema and Oracle XML DB

XML Schema is a schema definition language written in XML. It can be used to describe the structure and various other semantics of conforming instance documents. For example, the following XML schema definition, *po.xsd*, describes the structure and other properties of purchase order XML documents.

This manual refers to an XML schema definition as an *XML schema*.

Example 5-1 XML Schema Definition, *po.xsd*

The following is an example of an XML schema definition, *po.xsd*:

```
<schema targetNamespace="http://www.oracle.com/PO.xsd"
xmlns:po="http://www.oracle.com/PO.xsd"
xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="PurchaseOrderType">
    <sequence>
      <element name="PONum" type="decimal"/>
      <element name="Company">
        <simpleType>
          <restriction base="string">
            <maxLength value="100"/>
          </restriction>
        </simpleType>
      </element>
      <element name="Item" maxOccurs="1000">
        <complexType>
          <sequence>
            <element name="Part">
              <simpleType>
                <restriction base="string">
                  <maxLength value="1000"/>
                </restriction>
              </simpleType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</schema>
```

```
        </element>
        <element name="Price" type="float"/>
    </sequence>
</complexType>
</element>
</sequence>
</complexType>
<element name="PurchaseOrder" type="po:PurchaseOrderType"/>
</schema>
```

Example 5–2 XML Document, *po.xml* Conforming to XML Schema, *po.xsd*

The following is an example of an XML document that conforms to XML schema *po.xsd*:

```
<PurchaseOrder xmlns="http://www.oracle.com/PO.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
  <PONum>1001</PONum>
  <Company>Oracle Corp</Company>
  <Item>
    <Part>9i Doc Set</Part>
    <Price>2550</Price>
  </Item>
</PurchaseOrder>
```

Note:

The URL 'http://www.oracle.com/PO.xsd' used here is simply a name that uniquely identifies the registered XML schema within the database and need not be the physical URL at which the XML schema document is located. Also, the target namespace of the XML schema is another URL, different from the XML schema location URL, that specifies an abstract namespace within which elements and types get declared.

An XML schema can optionally specify the target namespace URL. If this attribute is omitted, the XML schema has no target namespace. **Note:** The `targetNamespace` is commonly the same as XML schema's URL.

An XML instance document must specify both the namespace of the root element (same as the XML schema's target namespace) and the location (URL) of the XML schema that defines this root element. The location is specified with attribute `xsi:schemaLocation`. When the XML schema has no target namespace, use attribute `xsi:noNamespaceSchemaLocation` to specify the XML schema URL.

Using Oracle XML DB and XML Schema

Oracle XML DB uses annotated XML schema as metadata, that is, the standard XML Schema definitions along with several Oracle XML DB-defined attributes. These attributes are in a different namespace and control how instance documents get mapped into the database. Since these attributes are in a different namespace from the XML schema namespace, such annotated XML schemas are still legal XML schema documents:

See Also: Namespace of XML Schema constructs:
<http://www.w3.org/2001/XMLSchema>

When using Oracle XML DB, you must first register your XML schema. You can then use the XML schema URLs while creating `XMLType` tables, columns, and views.

Oracle XML DB provides XML Schema support for the following tasks:

- Registering any W3C-compliant XML schemas.

- Validating your XML documents against a registered XML schema definitions.
- Registering local and global XML schemas.
- Generating XML schema from object types.
- Referencing an XML schema owned by another user.
- Explicitly referencing a global XML schema when a local XML schema exists with the same name.
- Generating a structured 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 type mapping when there are multiple legal mappings.
- Creating `XMLType` tables, views and columns based on registered XML schemas.
- Performing manipulation (DML) and queries on XML schema-based `XMLType` tables.
- Automatically inserting data into default tables when schema-based XML instances are inserted into Oracle XML DB Repository using FTP, HTTP/WebDav protocols and other languages.

Why Do We Need XML Schema?

As described in [Chapter 4, "Using XMLType"](#), `XMLType` is a datatype that facilitates storing XML in columns and tables in the database. XML schemas further facilitate storing XML columns and tables in the database, and they offer you more storage and access options for XML data along with space- performance-saving options.

For example, you can use XML schema to declare which elements and attributes can be used and what kinds of element nesting, and datatypes are allowed in the XML documents being stored or processed.

XML Schema Provides Flexible XML-to-SQL Mapping Setup

Using XML schema with Oracle XML DB provides a flexible setup for XML storage mapping. For example:

- If your data is highly structured (mostly XML), each element in the XML documents can be stored as a column in a table.

- If your data is unstructured (all or mostly non-XML data), the data can be stored in a Character Large Object (CLOB).

Which storage method you choose depends on how your data will be used and depends on the queriability and your requirements for querying and updating your data. In other words, using XML schema gives you more flexibility for storing highly structured or unstructured data.

XML Schema Allows XML Instance Validation

Another advantage of using XML schema with Oracle XML DB is that you can perform XML instance validation according to the XML schema and with respect to Oracle XML Repository requirements for optimal performance. For example, an XML schema can check that all incoming XML documents comply with definitions declared in the XML schema, such as allowed structure, type, number of allowed item occurrences, or allowed length of items.

Also, by registering XML schema in Oracle XML DB, when inserting and storing XML instances using Protocols, such as FTP or HTTP, the XML schema information can influence how efficiently XML instances are inserted.

When XML instances must be handled without any prior information about them, XML schema can be useful in predicting optimum storage, fidelity, and access.

Introducing DBMS_XMLSCHEMA

Oracle XML DB's XML schema functionality is available through the PL/SQL supplied package, `DBMS_XMLSCHEMA`, a server-side component that handles the registration of XML schema definitions for use by Oracle XML DB applications.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

Two of the main `DBMS_XMLSCHEMA` functions are:

- `registerSchema()`. This registers an XML schema given:
 - XML schema source, which can be in a variety of formats, including string, LOB, XMLType, and URType
 - Its schema URL or XMLSchema name
- `deleteSchema()`. This deletes a previously registered XML schema, identified by its URL or XMLSchema name.

Registering Your XML Schema Before Using Oracle XML DB

An XML schema must be registered before it can be used or referenced in any context by Oracle XML DB. XML schema are registered by using `DBMS_XMLSCHEMA.registerSchema()` and specifying the following:

- The XML schema source document as a `VARCHAR`, `CLOB`, `XMLType`, or `URIType`.
- The XML schema URL. This is a name for the XML schema that is used within XML instance documents to specify the location of the XML schema to which they conform.

After registration has completed:

- XML documents conforming to this XML schema, and referencing it using the XML schema's URL within the XML document, can be processed by Oracle XML DB.
- Tables and columns can be created for root XML elements defined by this XML schema to store the conforming XML documents.

Registering Your XML Schema Using `DBMS_XMLSCHEMA`

Use `DBMS_XMLSCHEMA` to register your XML schema. This involves specifying the XML schema document and its URL, also known as the *XML schema location*.

Example 5-3 Registering an XML Schema That Declares a complexType Using `DBMS_XMLSCHEMA`

Consider the following XML schema. It declares a `complexType` called `PurchaseOrderType` and an element `PurchaseOrder` of this type. The schema is stored in the PL/SQL variable `doc`. The following registers the XML schema at URL: `http://www.oracle.com/PO.xsd`:

```
declare
    doc varchar2(1000) := '<schema
targetNamespace="http://www.oracle.com/PO.xsd"
xmlns:po="http://www.oracle.com/PO.xsd"
xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="PurchaseOrderType">
    <sequence>
      <element name="PONum" type="decimal"/>
      <element name="Company">
        <simpleType>
          <restriction base="string">
```

```

        <maxLength value="100"/>
    </restriction>
</simpleType>
</element>
<element name="Item" maxOccurs="1000">
    <complexType>
        <sequence>
            <element name="Part">
                <simpleType>
                    <restriction base="string">
                        <maxLength value="1000"/>
                    </restriction>
                </simpleType>
            </element>
            <element name="Price" type="float"/>
        </sequence>
    </complexType>
</element>
</sequence>
</complexType>
<element name="PurchaseOrder" type="po:PurchaseOrderType"/>
</schema>';
begin
    dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;
```

The registered schema can be used to create XML schema-Based tables, or XML schema-based columns. For example, the following statement creates a table with an XML schema-based column.

```

create table po_tab(
    id number,
    po sys.XMLType
)
xmltype column po
    XMLSCHEMA "http://www.oracle.com/PO.xsd"
    element "PurchaseOrder";
```

The following shows an XMLType instance that conforms to the preceding XML schema being inserted into the preceding table. The schemaLocation attribute specifies the schema URL:

```

insert into po_tab values (1,
xmltype('<PurchaseOrder xmlns="http://www.oracle.com/PO.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
  <PONum>1001</PONum>
  <Company>Oracle Corp</Company>
  <Item>
    <Part>9i Doc Set</Part>
    <Price>2550</Price>
  </Item>
  <Item>
    <Part>8i Doc Set</Part>
    <Price>350</Price>
  </Item>
</PurchaseOrder>');
```

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

Local and Global XML Schemas

XML schemas can be registered as local or global:

- *Local XML schema:* An XML schema registered as a local schema is, by default, visible only to the owner.
- *Global XML schema:* An XML schema registered as a global schema is, by default, visible and usable by all database users.

When you register an XML schema, DBMS_XMLSCHEMA adds an Oracle XML DB resource corresponding to the XML schema into the Oracle XML DB Repository. The XML schema URL determines the path name of the resource in Oracle XML DB Repository according to the following rules:

Local XML Schema

In Oracle XML DB, *local XML schema* resources are created under the `/sys/schemas/<username>` directory. The rest of the path name is derived from the schema URL.

Example 5–4 A Local XML Schema

For example, a local XML schema with schema URL:

```
http://www.myco.com/PO.xsd
```

registered by SCOTT, is given the path name:

```
/sys/schemas/SCOTT/www.myco.com/PO.xsd.
```

Database users need appropriate permissions (ACLs) to create a resource with this path name in order to register the XML schema as a local XML schema.

See Also: [Chapter 18, "Oracle XML DB Resource Security"](#)

By default, an XML schema belongs to you after registering the XML schema with Oracle XML DB. A reference to the XML schema document is stored in Oracle XML DB Repository, in directory:

```
/sys/schemas/<username>/...
```

For example, if you, SCOTT, registered the preceding XML schema, it is mapped to the file:

```
/sys/schemas/SCOTT/www.oracle.com/PO.xsd
```

Such XML schemas are referred to as *local*. In general, they are usable only by you to whom they belong.

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 supports fully qualified XML schema URLs which can be specified as:

```
http://xmlns.oracle.com/xdb/schemas/SCOTT/www.oracle.com/PO.xsd
```

This extended URL can be used by privileged users to specify XML schema belonging to other users.

Global XML Schema

In contrast to local schema, privileged users can register an XML schema as a *global XML schema* by specifying an argument in the `DBMS_XMLSCHEMA` registration function.

Global schemas are visible to *all* users and stored under the

```
/sys/schemas/PUBLIC/ directory in Oracle XML DB Repository.
```

Note: Access to this directory is controlled by Access Control Lists (ACLs) and, by default, is writeable only by a DBA. You need WRITE privileges on this directory to register global schemas.

XDBAdmin role also provides WRITE access to this directory, assuming that it is protected by the default "protected" ACL.

See also [Chapter 18, "Oracle XML DB Resource Security"](#) for further information on privileges and for details on XDBAdmin role.

You can register a local schema with the same URL as an existing global schema. A local schema always hides any global schema with the same name (URL).

Example 5-5 A Global XML Schema

For example, a global schema registered by SCOTT with the URL:

```
www.myco.com/PO.xsd
```

is mapped to Oracle XML DB Repository at:

```
/sys/schemas/PUBLIC/www.myco.com/PO.xsd
```

Database users need appropriate permissions (ACLs) to create this resource in order to register the XML schema as *global*.

Registering Your XML Schema: Oracle XML DB Sets Up the Storage and Access Infrastructure

As part of registering an XML schema, Oracle XML DB also performs several other steps to facilitate storing, accessing, and manipulating XML instances that conform to the XML schema. These steps include:

- *Creating types:* When an XML schema is registered, Oracle creates the appropriate SQL object types that enable the structured storage of XML documents that conform to this XML schema. You can use Oracle XML DB-defined attributes in XML schema documents to control how these object types are generated.
- *Creating default tables:* As part of XML schema registration, Oracle XML DB generates default XMLTYPE tables for all root elements. You can also specify any column and table level constraints for use during table creation.

See Also:

- ["Specifying SQL Object Type Names with SQLName, SQLType Attributes"](#) on page 5-25
- [Chapter 3, "Using Oracle XML DB"](#)
- *Creating Java beans*: Java beans can be optionally generated during XML schema registration. These Java classes provide accessor and mutator methods for elements and attributes declared in the schema. Access using Java beans offers better performance for manipulating XML when the XML schema is well known. This helps avoid run-time name translation.

Deleting Your XML Schema Using DBMS_XMLSCHEMA

You can delete your registered XML schema by using the `DBMS_XMLSCHEMA.deleteSchema` procedure. When you attempt to delete an XML schema, `DBMS_XMLSCHEMA` checks:

- That the current user has the appropriate privileges (ACLs) to delete the resource corresponding to the XML schema within Oracle XML DB Repository. You can thus control which users can delete which XML schemas by setting the appropriate ACLs on the XML schema resources.
- For dependents. If there are any dependents, it raises an error and the deletion operation fails. This is referred to as the **RESTRICT** mode of deleting XML schemas.

FORCE Mode

A *FORCE mode* option is provided while deleting XML schemas. If you specify the *FORCE mode* option, the XML schema deletion proceeds even if it fails the dependency check. In this mode, XML schema deletion marks all its dependents as invalid.

CASCADE Mode

The *CASCADE mode* option drops all generated types, default tables, and Java beans as part of a previous call to register schema.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB* the chapter on `DBMS_XMLSCHEMA`.

Example 5–6 Deleting the XML Schema Using DBMS_XMLSCHEMA

The following example deletes XML schema `PO.xsd`. First, the dependent table `po_tab` is dropped. Then, the schema is deleted using the `FORCE` and `CASCADE` modes with `DBMS_XMLSCHEMA.DELETE_SCHEMA`:

```
drop table po_tab;

EXEC dbms_xmlschema.deleteSchema('http://www.oracle.com/PO.xsd',
                                dbms_xmlschema.DELETE_CASCADE_FORCE);
```

Guidelines for Using Registered XML Schemas

The following sections describe guidelines for registering XML schema with Oracle XML DB.

Objects That Depend on Registered XML Schemas

The following objects depend on a registered XML schemas:

- Tables or views that have an `XMLType` column that conforms to some element in the XML schema.
- XML schemas that include or import this schema as part of their definition.
- Cursors that reference the XML schema name, for example, within `DBMS_XMLGEN` operators. Note that these are purely transient objects.

Creating XMLType Tables, Views, or Columns

After an XML schema has been registered, it can be used to create XML schema-based `XMLType` tables, views, and columns by referencing the following:

- The XML schema URL of a registered XML schema
- The name of the root element

Example 5–7 Post-Registration Creation of an XMLType Table

For example you can create an XML schema-based `XMLType` table as follows:

```
CREATE TABLE po_tab OF XMLTYPE
    XMLSCHEMA "http://www.oracle.com/PO.xsd" ELEMENT "PurchaseOrder";
```

The following statement inserts schema-conformant data:

```
insert into po_tab values (
```

```

xmltype(' <PurchaseOrder xmlns="http://www.oracle.com/PO.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
  <PONum>1001</PONum>
  <Company>Oracle Corp</Company>
  <Item>
    <Part>9i Doc Set</Part>
    <Price>2550</Price>
  </Item>
  <Item>
    <Part>8i Doc Set</Part>
    <Price>350</Price>
  </Item>
</PurchaseOrder>');

```

Validating XML Instances Against the XML Schema: `schemaValidate()`

You can validate an `XMLType` instance against a registered XML schema by using one of the validation methods.

See Also: [Chapter 6, "Transforming and Validating XMLType Data"](#)

Example 5–8 Validating XML Using `schemaValidate()`

The following PL/SQL example validates an XML instance against XML schema `PO.xsd`:

```

declare
  xmldoc xmltype;
begin

  -- populate xmldoc (by fetching from table)
  select value(p) into xmldoc from po_tab p;

  -- validate against XML schema
  xmldoc.schemavalidate();

  if xmldoc.isschemavalidated() = 1 then
    dbms_output.put_line('Data is valid');
  else
    dbms_output.put_line('Data is invalid');
  end if;
end;

```

Fully Qualified XML Schema URLs

By default, XML schema URL names are always referenced within the scope of the current user. In other words, when database users specify XML Schema URLs, they 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, then Oracle XML DB raises an error.

XML Schema That Users Cannot Reference

These rules imply that, by default, users cannot reference the following kinds of XML schemas:

- XML schemas owned by a different database user
- Global XML schemas that have the same name as local XML schemas

Fully Qualified XML Schema URLs Permit Explicit Reference to XML Schema URLs

To permit explicit reference to XML schemas in these cases, Oracle XML DB supports a notion of *fully qualified* XML schema URLs. In this form, the name of the database user owning the XML schema is also specified as part of the XML schema URL, except that such XML schema URLs belong to the Oracle XML DB namespace as follows:

```
http://xmlns.oracle.com/xdb/schemas/<database-user-name>/<schemaURL-minus-protocol>
```

Example 5–9 Using Fully Qualified XML Schema URL

For example, consider the global XML schema with the following URL:

```
http://www.example.com/po.xsd
```

Assume that database user SCOTT has a local XML schema with the same URL:

```
http://www.example.com/po.xsd
```

User JOE can reference the local XML schema owned by SCOTT as follows:

```
http://xmlns.oracle.com/xdb/schemas/SCOTT/www.example.com/po.xsd
```

Similarly, the fully qualified URL for the global XML schema is:

```
http://xmlns.oracle.com/xdb/schemas/PUBLIC/www.example.com/po.xsd
```

Transactional Behavior of XML Schema Registration

Registration of an XML schema is non transactional and auto committed as with other SQL DDL operations, as follows:

- If registration succeeds, the operation is auto committed.
- If registration fails, the database is rolled back to the state before the registration began.

Since XML schema registration potentially involves creating object types and tables, error recovery involves dropping any such created types and tables. Thus, the entire XML schema registration is guaranteed to be atomic. That is, either it succeeds or the database is restored to the state before the start of registration.

Java Bean Generation During XML Schema Registration

Java beans can be optionally generated during XML schema registration and provide accessor and mutator methods for elements and attributes declared in the XML schema. Access to XML data stored in the database, using Java beans, offers better performance for manipulating the XML data when the XML schema is well known and mostly fixed by avoiding run-time name translation.

Example 5–10 Generating Java Bean Classes During XML Schema Registration

For example, the Java bean class corresponding to the XML schema `PO.xsd` has the following accessor and mutator methods:

```
public class PurchaseOrder extends XMLTypeBean
{
    public BigDecimal getPONum()
    {
        ....
    }
    public void setPONum(BigDecimal val)
    {
        ....
    }
    public String getCompany()
    {
        ....
    }
    public void setCompany(String val)
    {
        ....
    }
}
```

```

    }
    ....
}

```

Note: Java bean support in Oracle XML DB is only for XML schema-based XML documents. Non-schema-based XML documents can be manipulated using Oracle XML DB DOM API. See [Chapter 9, "Java and Java Bean APIs for XMLType"](#).

Generating XML Schema Using DBMS_XMLSCHEMA.generateSchema()

An XML schema can be generated from an object-relational type automatically using a default mapping. The `generateSchema()` and `generateSchemas()` functions in the `DBMS_XMLSCHEMA` package take in a string that has the object type name and another that has the Oracle XML DB XML schema.

- `generateSchema()` returns an `XMLType` containing an XML schema. It can optionally generate XML schema for all types referenced by the given object type or restricted only to the top-level types.
- `generateSchemas()` is similar, except that it returns an `XMLSequenceType` of XML schemas, each corresponding to a different namespace. It also takes an additional optional argument, specifying the root URL of the preferred XML schema location:

```
http://xmlns.oracle.com/xdb/schemas/<schema>.xsd
```

They can also optionally generate annotated XML schemas that can be used to register the XML schema with Oracle XML DB.

Example 5–11 *Generating XML Schema: Using generateSchema()*

For example, given the object type:

```

connect t1/t1
CREATE TYPE employee_t AS OBJECT
(
    empno NUMBER(10),
    ename VARCHAR2(200),
    salary NUMBER(10,2)
);

```

You can generate the schema for this type as follows :

```
select dbms_xmlschema.generateschema('T1', 'EMPLOYEE_T') from dual;
```

This returns a schema corresponding to the type EMPLOYEE_T. The schema declares an element named EMPLOYEE_T and a complexType called EMPLOYEE_TType. The schema includes other annotation from <http://xmlns.oracle.com/xdb>.

```
DBMS_XMLSCHEMA.GENERATESCHEMA('T1', 'EMPLOYEE_T')
```

```
-----
<xsd:schema targetNamespace="http://ns.oracle.com/xdb/T1" xmlns="http://ns.oracle.com/xdb/T1" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xdb="http://xmlns.oracle.com/xdb" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/xdb http://xmlns.oracle.com/xdb/XDBSchema.xsd">
  <xsd:element name="EMPLOYEE_T" type="EMPLOYEE_TType" xdb:SQLType="EMPLOYEE_T" xdb:SQLSchema="T1"/>
  <xsd:complexType name="EMPLOYEE_TType">
    <xsd:sequence>
      <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="EMPNO" xdb:SQLType="NUMBER"/>
      <xsd:element name="ENAME" type="xsd:string" xdb:SQLName="ENAME" xdb:SQLType="VARCHAR2"/>
      <xsd:element name="SALARY" type="xsd:double" xdb:SQLName="SALARY" xdb:SQLType="NUMBER"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

XML Schema-Related Methods of XMLType

[Table 5-1](#) lists the `XMLType` API's XML schema-related methods.

Table 5-1 XMLType API XML Schema-Related Methods

XMLType API Method	Description
<code>isSchemaBased()</code>	Returns TRUE if the <code>XMLType</code> instance is based on an XML schema, FALSE otherwise.
<code>getSchemaURL()</code> <code>getRootElement()</code> <code>getNamespace()</code>	Returns the XML schema URL, name of root element, and the namespace for an XML schema-based <code>XMLType</code> instance.
<code>schemaValidate()</code> <code>isSchemaValid()</code> <code>isSchemaValidated()</code> <code>setSchemaValidated()</code>	An <code>XMLType</code> instance can be validated against a registered XML schema using the <code>validation</code> methods. See Chapter 6, "Transforming and Validating XMLType Data" .

Managing and Storing XML Schema

XML schema documents are themselves stored in Oracle XML DB as `XMLType` instances. XML schema-related `XMLType` types and tables are created as part of the Oracle XML DB installation script, `catxdbs.sql`.

Root XML Schema, XDBSchema.xsd

The XML schema for XML schemas is called the root XML schema, `XDBSchema.xsd`. `XDBSchema.xsd` describes any valid XML schema document that can be registered by Oracle XML DB. You can access `XDBSchema.xsd` through Oracle XML DB Repository at:

`/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBSchema.xsd`

See Also:

- [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)
- [Appendix A, "Installing and Configuring Oracle XML DB"](#)

How Are XML Schema-Based XMLType Structures Stored?

XML Schema-based XMLType structures are stored in one of the following ways:

- *In underlying object type columns.* This is the default storage mechanism.
 - SQL object types can be created optionally during the XML schema registration process. See ["Creating XMLType Tables and Columns Based on XML Schema"](#) on page 5-23.
 - See ["Specifying SQL Object Type Names with SQLName, SQLType Attributes"](#) on page 5-25.
- *In a single underlying LOB column.* Here the storage choice is specified in the `STORE AS` clause of the `CREATE TABLE` statement:

```
CREATE TABLE po_tab OF xmltype
  STORE AS CLOB
  ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

Design criteria for storing XML data are discussed in [Chapter 2, "Getting Started with Oracle XML DB"](#) and [Chapter 3, "Using Oracle XML DB"](#).

Specifying the Storage Mechanism

Instead of using the `STORE AS` clause, you can specify that the table and column be stored according to a mapping based on a particular XML schema. You can specify the URL for the XML schema used for the mapping.

Non-schema-based XML data can be stored in tables using CLOBs. However you do not gain benefits such as indexing, query-rewrite, and so on.

DOM Fidelity

Document Object Model (DOM) fidelity is the concept of retaining the structure of a retrieved XML document, compared to the original XML document, for DOM traversals. DOM fidelity is needed to ensure the accuracy and integrity of XML documents stored in Oracle XML DB.

See Also: ["Setting the SQLInLine Attribute to FALSE for Out-of-Line Storage"](#) on page 5-38

How Oracle XML DB Ensures DOM Fidelity with XML Schema

All elements and attributes declared in the XML schema are mapped to separate attributes in the corresponding SQL object type. However, some pieces of information in XML instance documents are not represented directly by these element or attributes, such as:

- Comments
- Namespace declarations
- Prefix information

To ensure the integrity and accuracy of this data, for example, when regenerating XML documents stored in the database, Oracle XML DB uses a data integrity mechanism called *DOM fidelity*.

DOM fidelity refers to how identical the *returned* XML documents are compared to the *original* XML documents, particularly for purposes of DOM traversals.

DOM Fidelity and SYS_XDBPD\$

To guarantee that DOM fidelity is maintained and that the *returned* XML documents are identical to the *original* XML document for DOM traversals, Oracle XML DB adds a system binary attribute, `SYS_XDBPD$`, to each created object type.

This attribute stores all pieces of information that cannot be stored in any of the other attributes, thereby ensuring the DOM fidelity of all XML documents stored in Oracle XML DB. Examples of such pieces of information include: ordering information, comments, processing instructions, namespace prefixes, and so on.

This is mapped to a Positional Descriptor (PD) column.

Note: In general, it is not a good idea to set this information because the extra pieces of information, such as, comments, processing instructions, and so on, could be lost if there is no PD column.

How to Suppress SYS_XDBPD\$

If DOM fidelity is not required, you can suppress `SYS_XDBPD$` in the XML schema definition by setting the attribute, `maintainDOM=FALSE`.

Note: The attribute `SYS_XDBPD$` is omitted in many examples here for clarity. However, the attribute is always present as a Positional Descriptor (PD) column in all SQL object types generated by the XML schema registration process.

Creating XMLType Tables and Columns Based on XML Schema

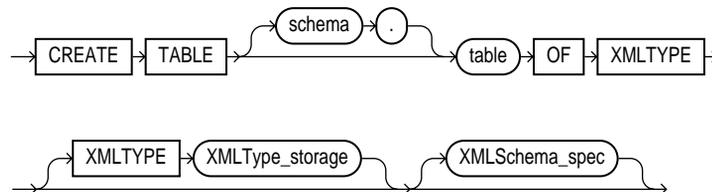
Oracle XML DB creates XML schema-based `XMLType` tables and columns by referencing:

- The XML schema URL of a registered XML schema
- The name of the root element

Figure 5–1 shows the syntax for creating an `XMLType` table:

```
CREATE TABLE [schema.] table OF XMLTYPE
  [XMLTYPE XMLType_storage] [XMLSchema_spec];
```

Figure 5–1 *Creating an XMLType Table*



A subset of the XPointer notation, shown in the following example, can also be used to provide a single URL containing the XML schema location and element name.

Example 5–12 *Creating XML Schema-Based XMLType Table*

This example creates the `XMLType` table `po_tab` using the XML schema at the given URL:

```
CREATE TABLE po_tab OF XMLTYPE
  XMLSCHEMA "http://www.oracle.com/PO.xsd" ELEMENT "PurchaseOrder";
```

An equivalent definition is:

```
CREATE TABLE po_tab OF XMLTYPE
  ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

SQL Object-Relational Types Store XML Schema-Based XMLType Tables

When an XML schema is registered, Oracle XML DB creates the appropriate SQL object types that enable structured storage of XML documents that conform to this XML schema. All SQL object types are created based on the current registered XML schema, by default.

Example 5–13 *Creating SQL Object Types to Store XMLType Tables*

or example, when PO.xsd is registered with Oracle XML DB, the following SQL types are created.

Note: The names of the types are generated names, and will not necessarily match Itemxxx_t, Itemxxx_COLL and PurchaseOrderTypexxx_T, where xxx is a 3-digit integer.

```
CREATE TYPE "Itemxxx_T" as object
(
  part varchar2(1000),
  price number
);

CREATE TYPE "Itemxxx_COLL" AS varray(1000) OF "Item_T";
CREATE TYPE "PurchaseOrderTypexxx_T" AS OBJECT
(
  ponum number,
  company varchar2(100),
  item Item_varray_COLL
);
```

Note: The names of the object types and attributes in the preceding example can be system-generated.

- If the XML schema already contains the `SQLName`, `SQLType`, or `SQLColType` attribute filled in (see ["Specifying SQL Object Type Names with SQLName, SQLType Attributes"](#) for details), this name is used as the object attribute's name.
- If the XML schema does not contain the `SQLName` attribute, the name is derived from the XML name, unless it cannot be used because of length or conflict reasons.

If the `SQLSchema` attribute is used, Oracle XML DB attempts to create the object type using the specified database schema. The current user must have the necessary privileges to perform this.

Specifying SQL Object Type Names with SQLName, SQLType Attributes

To specify specific names of SQL objects generated include the attributes `SQLName` and `SQLType` in the XML schema definition prior to registering the XML schema.

- If you specify the `SQLName` and `SQLType` values, Oracle XML DB creates the SQL object types using these names.
- If you do not specify these attributes, Oracle XML DB uses system-generated names.

Note: You do not have to specify values for any of these attributes. Oracle XML DB fills in appropriate values during the XML schema registration process. However, it is recommended that you specify the names of at least the top-level SQL types so that you can reference them later.

All annotations are in the form of attributes that can be specified within attribute and element declarations. These attributes belong to the Oracle XML DB namespace: `http://xmlns.oracle.com/xdb`

[Table 5-2](#) lists Oracle XML DB attributes that you can specify in element and attribute declarations.

Table 5–2 Attributes You Can Specify in Elements

Attribute	Values	Default	Description
SQLName	Any SQL identifier	Element name	Specifies the name of the attribute within the SQL object that maps to this XML element.
SQLType	Any SQL type name	Name generated from element name	Specifies the name of the SQL type corresponding to this XML element declaration.
SQLCollType	Any SQL collection type name	Name generated from element name	Specifies the name of the SQL collection type corresponding to this XML element that has <code>maxOccurs > 1</code> .
SQLSchema	Any SQL username	User registering XML schema	Name of database user owning the type specified by <code>SQLType</code> .
SQLCollSchema	Any SQL username	User registering XML schema	Name of database user owning the type specified by <code>SQLCollType</code> .
maintainOrder	true false	true	If true, the collection is mapped to a VARRAY. If false, the collection is mapped to a NESTED TABLE.
SQLInline	true false	true	If true this element is stored inline as an embedded attribute (or a collection if <code>maxOccurs > 1</code>). If false, a REF (or collection of REFs if <code>maxOccurs > 1</code>) is stored. This attribute will be forced to false in certain situations (like cyclic references) where SQL will not support inlining.
maintainDOM	true false	true	If true, instances of this element are stored such that they retain DOM fidelity on output. This implies that all comments, processing instructions, namespace declarations, and so on are retained in addition to the ordering of elements. If false, the output need not be guaranteed to have the same DOM behavior as the input.

Table 5–2 Attributes You Can Specify in Elements(Cont.)

Attribute	Values	Default	Description
columnProps	Any valid column storage clause	NULL	Specifies the column storage clause that is inserted into the default CREATE TABLE statement. It is useful mainly for elements that get mapped to tables, namely top-level element declarations and out-of-line element declarations.
tableProps	Any valid table storage clause	NULL	Specifies the TABLE storage clause that is appended to the default CREATE TABLE statement. This is meaningful mainly for global and out-of-line elements.
defaultTable	Any table name	Based on element name.	Specifies the name of the table into which XML instances of this schema should be stored. This is most useful in cases when the XML is being inserted from APIs where table name is not specified, for example, FTP and HTTP.
beanClassname	Any Java class name	Generated from element name.	Can be used within element declarations. If the element is based on a global complexType, this name must be identical to the beanClassname value within the complexType declaration. If a name is specified by the user, the bean generation will generate a bean class with this name instead of generating a name from the element name.
JavaClassname	Any Java class name	None	Used to specify the name of a Java class that is derived from the corresponding bean class to ensure that an object of this class is instantiated during bean access. If a JavaClassname is not specified, Oracle XML DB will instantiate an object of the bean class directly.

Table 5–3 Attributes You Can Specify in Elements Declaring Global complexTypes

Attribute	Values	Default	Description
SQLType	Any SQL type name	Name generated from element name	Specifies the name of the SQL type corresponding to this XML element declaration.
SQLSchema	Any SQL username	User registering XML schema	Name of database user owning the type specified by SQLType.
beanClassname	Any Java class name	Generated from element name.	Can be used within element declarations. If the element is based on a global complexType, this name must be identical to the beanClassname value within the complexType declaration. If a name is specified by the user, the bean generation will generate a bean class with this name, instead of generating a name from the element name.
maintainDOM	true false	true	If true, instances of this element are stored such that they retain DOM fidelity on output. This implies that all comments, processing instructions, namespace declarations, and so on, are retained in addition to the ordering of elements. If false, the output need not be guaranteed to have the same DOM behavior as the input.

Table 5–4 Attributes You Can Specify in XML Schema Declarations

Attribute	Values	Default	Description
mapUnboundedStringToLob	true false	false	If true, unbounded strings are mapped to CLOB by default. Similarly, unbounded binary data gets mapped to BLOB, by default. If false, unbounded strings are mapped to VARCHAR2(4000) and unbounded binary components are mapped to RAW(2000).
storeVarrayAsTable	true false	false	If true, the VARRAY is stored as a table (OCT). If false, the VARRAY is stored in a LOB.

SQL Mapping Is Specified in the XML Schema During Registration

Information regarding the SQL mapping is stored in the XML schema document. The registration process generates the SQL types, as described in "[Mapping of Types Using DBMS_XMLSCHEMA](#)" on page 5-32 and adds annotations to the XML schema document to store the mapping information. Annotations are in the form of new attributes.

Example 5–14 Capturing SQL Mapping Using SQLType and SQLName Attributes

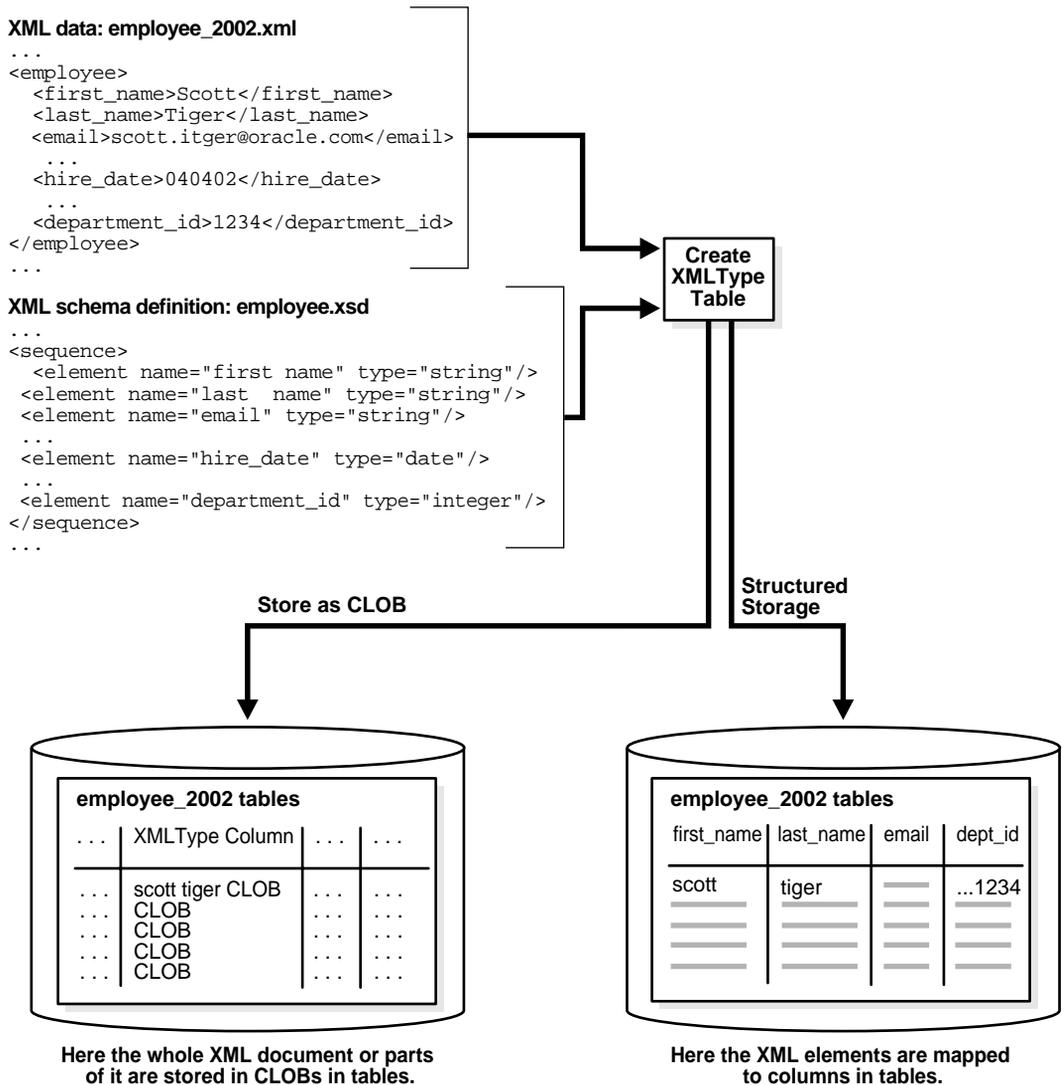
The following XML schema definition shows how SQL mapping information is captured using SQLType and SQLName attributes:

```
declare
  doc varchar2(3000) := '<schema
targetNamespace="http://www.oracle.com/PO.xsd"
xmlns:po="http://www.oracle.com/PO.xsd" xmlns:xdb="http://xmlns.oracle.com/xdb"
xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="PurchaseOrderType">
    <sequence>
      <element name="PONum" type="decimal" xdb:SQLName="PONUM"
xdb:SQLType="NUMBER" />
      <element name="Company" xdb:SQLName="COMPANY" xdb:SQLType="VARCHAR2">
        <simpleType>
          <restriction base="string">
            <maxLength value="100" />
          </restriction>
        </simpleType>
      </element>
      <element name="Item" xdb:SQLName="ITEM" xdb:SQLType="ITEM_T"
maxOccurs="1000">
        <complexType>
          <sequence>
            <element name="Part" xdb:SQLName="PART" xdb:SQLType="VARCHAR2">
              <simpleType>
                <restriction base="string">
                  <maxLength value="1000" />
                </restriction>
              </simpleType>
            </element>
            <element name="Price" type="float" xdb:SQLName="PRICE"
xdb:SQLType="NUMBER" />
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</schema>
```

```
</sequence>
</complexType>
<element name="PurchaseOrder" type="po:PurchaseOrderType"/>
</schema>;
begin
  dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;
```

Figure 5–2 shows how Oracle XML DB creates XML schema-based `XMLType` tables using an XML document and mapping specified in an XML schema. An `XMLType` table is first created and depending on how the storage is specified in the XML schema, the XML document is mapped and stored either as a CLOB in one `XMLType` column, or stored object-relationally and spread out across several columns in the table.

Figure 5–2 How Oracle XML DB Maps XML Schema-Based XMLType Tables



An XMLType table is first created and depending on how the storage is specified in the XML schema, the XML document is mapped and stored either as a CLOB in one XMLType column, or stored object-rationally and spread out across several columns in the table.

Mapping of Types Using DBMS_XMLSCHEMA

Use `DBMS_XMLSCHEMA` to set the mapping of type information for attributes and elements.

Setting Attribute Mapping Type Information

An attribute declaration can have its type specified in terms of one of the following:

- Primitive type
- Global `simpleType`, declared within this XML schema or in an external XML schema
- Reference to global attribute (`ref=" . . "`), declared within this XML schema or in an external XML schema
- Local `simpleType`

In all cases, the SQL type and associated information (length and precision) as well as the memory mapping information, are derived from the `simpleType` on which the attribute is based.

Overriding SQL Types

You can explicitly specify an `SQLType` value in the input XML schema document. In this case, your specified type is validated. This allows for the following specific forms of overrides:

- If the default type is a `STRING`, you can override it with any of the following: `CHAR`, `VARCHAR`, or `CLOB`.
- If the default type is `RAW`, you can override it with `RAW` or `BLOB`.

Setting Element Mapping Type Information

An element declaration can specify its type in terms of one of the following:

- Any of the ways for specifying type for an attribute declaration. See "[Setting Attribute Mapping Type Information](#)" on page 5-32.
- Global `complexType`, specified within this XML schema document or in an external XML schema.
- Reference to a global element (`ref=" . . . "`), which could itself be within this XML schema document or in an external XML schema.

- Local `complexType`.

Overriding SQL Type

An element based on a `complexType` is, by default, mapped to an object type containing attributes corresponding to each of the sub-elements and attributes. However, you can override this mapping by explicitly specifying a value for `SQLType` attribute in the input XML schema. The following values for `SQLType` are permitted in this case:

- `VARCHAR2`
- `RAW`
- `CLOB`
- `BLOB`

These represent storage of the XML in a text or unexploded form in the database. The following special cases are handled:

- If a cycle is detected, as part of processing the `complexTypes` used to declare elements and 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 may need to be created.
 - If `SQLInline = "true"`, a `varray` type is created whose element type is the SQL type previously determined.
 - * Cardinality of the `VARRAY` is determined based on the value of `maxOccurs` attribute.
 - * The name of the `VARRAY` type is either explicitly specified by the user using `SQLCollType` attribute or obtained by mangling the element name.
 - If `SQLInline = "false"`, the SQL type is set to `XDB.XDB$XMLTYPE_REF_LIST_T`, a predefined type representing an array of `REFs` to `XMLType`.
- If the element is a global element, or if `SQLInline = "false"`, a default table needs to be created. It is added to the table creation context. The name of the default table has either been specified by the user, or derived by mangling the element name.

XML Schema: Mapping SimpleTypes to SQL

This section describes how XML schema definitions map XML schema `simpleType` to SQL object types. [Figure 5-3](#) shows an example of this.

[Table 5-5](#) through [Table 5-8](#) list the default mapping of XML schema `simpleType` to SQL, as specified in the XML schema definition. For example:

- An XML primitive type is mapped to the closest SQL datatype. For example, DECIMAL, POSITIVEINTEGER, and FLOAT are all mapped to SQL NUMBER.
- An XML enumeration type is mapped to an object type with a single RAW(n) attribute. The value of n is determined by the number of possible values in the enumeration declaration.
- An XML list or a union datatype is mapped to a string (VARCHAR2/CLOB) datatype in SQL.

Figure 5-3 Mapping simpleType: XML Strings to SQL VARCHAR2 or CLOBs

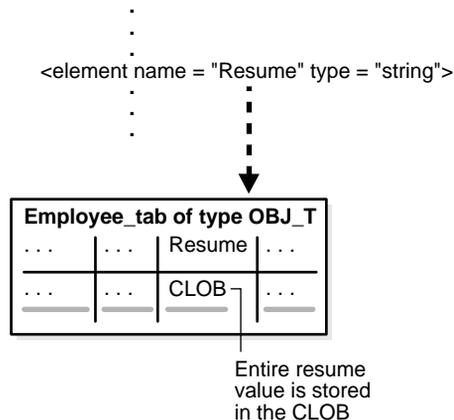


Table 5–5 Mapping XML String Datatypes to SQL

XML Primitive Type	Length or MaxLength Facet	Default Mapping	Compatible Datatype
string	n	VARCHAR2(n) if n < 4000, else VARCHAR2(4000)	CHAR, VARCHAR2, CLOB
string	--	VARCHAR2(4000) if mapUnboundedStringToLob="true", CLOB	CHAR, VARCHAR2, CLOB

Table 5–6 Mapping XML Binary Datatypes (hexBinary/base64Binary) to SQL

XML Primitive Type	Length or MaxLength Facet	Default Mapping	Compatible Datatype
hexBinary, base64Binary	n	RAW(n) if n < 2000, else RAW(2000)	RAW, BLOB
hexBinary, base64Binary	-	RAW(2000) if mapUnboundedStringToLob="true", BLOB	RAW, BLOB

Table 5–7 Default Mapping of Numeric XML Primitive Types to SQL

XML Simple Type	Default Oracle DataType	totalDigits (m), fractionDigits(n) Specified	Compatible Datatypes
float	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
double	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
decimal	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
integer	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
nonNegativeInteger	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
positiveInteger	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
nonPositiveInteger	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE

Table 5–7 Default Mapping of Numeric XML Primitive Types to SQL(Cont.)

XML Simple Type	Default Oracle Data Type	totalDigits (m), fractionDigits(n) Specified	Compatible Datatypes
negativeInteger	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
long	NUMBER(20)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
unsignedLong	NUMBER(20)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
int	NUMBER(10)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
unsignedInt	NUMBER(10)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
short	NUMBER(5)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
unsignedShort	NUMBER(5)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
byte	NUMBER(3)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
unsignedByte	NUMBER(3)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE

Table 5–8 Mapping XML Date Datatypes to SQL

XML Primitive Type	Default Mapping	Compatible Datatypes
datetime	TIMESTAMP	DATE
time	TIMESTAMP	DATE
date	DATE	DATE
gDay	DATE	DATE
gMonth	DATE	DATE
gYear	DATE	DATE
gYearMonth	DATE	DATE
gMonthDay	DATE	DATE
duration	VARCHAR2(4000)	none

Table 5–9 Default Mapping of Other XML Primitive Datatypes to SQL

XML Simple Type	Default Oracle DataType	Compatible Datatypes
boolean	RAW(1)	VARCHAR2
Language(string)	VARCHAR2(4000)	CLOB, CHAR
NMTOKEN(string)	VARCHAR2(4000)	CLOB, CHAR
NMTOKENS(string)	VARCHAR2(4000)	CLOB, CHAR
Name(string)	VARCHAR2(4000)	CLOB, CHAR
NCName(string)	VARCHAR2(4000)	CLOB, CHAR
ID	VARCHAR2(4000)	CLOB, CHAR
IDREF	VARCHAR2(4000)	CLOB, CHAR
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	--

simpleType: Mapping XML Strings to SQL VARCHAR2 Versus CLOBs

If the XML schema specifies the datatype to be string with a `maxLength` value of less than 4000, it is mapped to a VARCHAR2 attribute of the specified length. However, if `maxLength` is not specified in the XML schema, it can only be mapped to a LOB. This is sub-optimal when most of the string values are small and only a small fraction of them are large enough to need a LOB.

See Also: [Table 5–5, "Mapping XML String Datatypes to SQL"](#)

XML Schema: Mapping complexTypes to SQL

Using XML schema, a `complexType` is mapped to an SQL object type as follows:

- *XML attributes* declared within the `complexType` are mapped to *object attributes*. The `simpleType` defining the XML attribute determines the SQL datatype of the corresponding attribute.
- *XML elements* declared within the `complexType` are also mapped to *object attributes*. The datatype of the object attribute is determined by the `simpleType` or `complexType` defining the XML element.

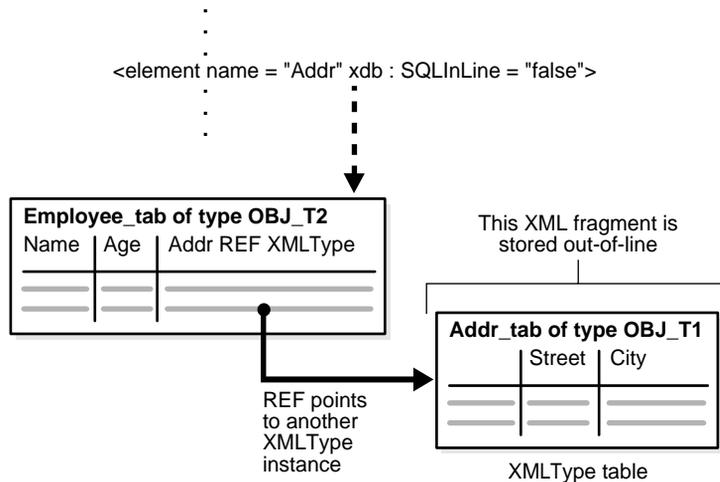
If the XML element is declared with attribute `maxOccurs > 1`, it is mapped to a collection attribute in SQL. The collection could be a VARRAY (default) or nested table if the `maintainOrder` attribute is set to false. Further, the default storage of the VARRAY is in Ordered Collections in Tables (OCTs) instead of LOBs. You can choose LOB storage by setting the `storeAsLob` attribute to true.

See Also: ["Ordered Collections in Tables \(OCTs\)"](#) on page 5-72

Setting the SQLInline Attribute to FALSE for Out-of-Line Storage

By default, a sub-element is mapped to an embedded object attribute. However, there may be scenarios where out-of-line storage offers better performance. In such cases the `SQLInline` attribute can be set to false, and Oracle XML DB generates an object type with an embedded `REF` attribute. `REF` points to another instance of `XMLType` that corresponds to the XML fragment that gets stored out-of-line. Default `XMLType` tables are also created to store the out-of-line fragments.

[Figure 5-4](#) illustrates the mapping of a `complexType` to SQL for out-of-line storage.

Figure 5–4 Mapping complexType to SQL for Out-of-Line Storage

Example 5–15 Oracle XML DB XML Schema: complexType Mapping - Setting SQLInline Attribute to False for Out-of-Line Storage

In this example element `Addr`'s attribute, `xdb:SQLInline`, is set to `false`. The resulting object type `OBJ_T2` has a column of type `XMLType` with an embedded `REF` attribute. The `REF` attribute points to another `XMLType` instance created of object type `OBJ_T1` in table `Addr_tab`. `Addr_tab` has columns `Street` and `City`. The latter `XMLType` instance is stored out-of-line.

```

declare
  doc varchar2(3000) := '<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/emp.xsd"
xmlns:emp="http://www.oracle.com/emp.xsd"
xmlns:xdb="http://xmlns.oracle.com/xdb">
  <complexType name = "Employee" xdb:SQLType="OBJ_T2">
    <sequence>
      <element name = "Name" type = "string"/>
      <element name = "Age" type = "decimal"/>
      <element name = "Addr" xdb:SQLInline = "false">
        <complexType xdb:SQLType="OBJ_T1">
          <sequence>
            <element name = "Street" type = "string"/>
            <element name = "City" type = "string"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>

```

```
        </element>
    </sequence>
</complexType>
</schema>';
begin
    dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;
```

On registering this XML schema, Oracle XML DB generates the following types and XMLType tables:

```
CREATE TYPE OBJ_T1 AS OBJECT
(
    SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
    Street VARCHAR2(4000),
    City VARCHAR2(4000)
);

CREATE TYPE OBJ_T2 AS OBJECT
(
    SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
    Name VARCHAR2(4000),
    Age NUMBER,
    Addr REF XMLType
);
```

Mapping XML Fragments to Large Objects (LOBs)

You can specify the `SQLType` for a complex element as a Character Large Object (CLOB) or Binary Large Object (BLOB) as shown in [Figure 5-5](#). Here the entire XML fragment is stored in a LOB attribute. This is useful when parts of the XML document are seldom queried but are mostly retrieved and stored as single pieces. By storing XML fragments as LOBs, you can save on parsing/decomposition/recomposition overheads.

Example 5-16 Oracle XML DB XML Schema: complexType Mapping XML Fragments to LOBs

In the following example, the XML schema specifies that the XML fragment's element `Addr` is using the attribute `SQLType="CLOB"`:

```
declare
    doc varchar2(3000) := '<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/emp.xsd"
xmlns:emp="http://www.oracle.com/emp.xsd"
```

```

xmlns:xdb="http://xmlns.oracle.com/xdb">
  <complexType name = "Employee" xdb:SQLType="OBJ_T2">
    <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('http://www.oracle.com/PO.xsd', doc);
end;

```

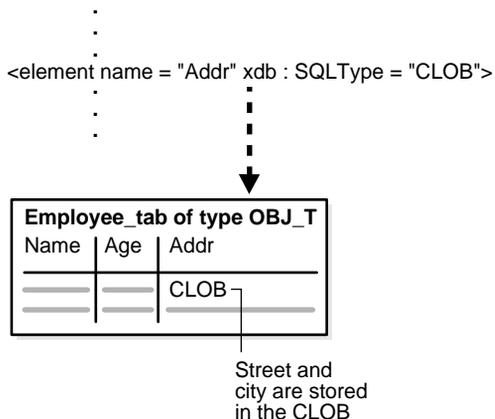
On registering this XML schema, Oracle XML DB generates the following types and XMLType tables:

```

CREATE TYPE OBJ_T AS OBJECT
(
  SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
  Name VARCHAR2(4000),
  Age NUMBER,
  Addr CLOB
);

```

Figure 5–5 Mapping complexType XML Fragments to Character Large Objects (CLOBs)



Oracle XML DB complexType Extensions and Restrictions

In XML schema, complexTypes are declared based on complexContent and simpleContent.

- simpleContent is declared as an extension of simpleType.
- complexContent is declared as one of the following:
 - Base type
 - complexType extension
 - complexType restriction.

complexType Declarations in XML Schema: Handling Inheritance

For complexType, Oracle XML DB handles inheritance in the XML schema as follows:

- *For complexTypes declared to extend other complexTypes*, 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 complexTypes declared to restrict other complexTypes*, the SQL type for the sub-complex type is set to be the same as the SQL type for its base type. This is

because SQL does not support restriction of object types through the inheritance mechanism. Any constraints are imposed by the restriction in XML schema.

Example 5–17 Inheritance in XML Schema: complexContent as an Extension of complexTypes

Consider an XML schema that defines a base complexType "Address" and two extensions "USAddress" and "IntlAddress".

```

declare
  doc varchar2(3000) := '<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:xdb="http://xmlns.oracle.com/xdb">
<xs:complexType name="Address" xdb:SQLType="ADDR_T">
  <xs:sequence>
    <xs:element name="street" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="USAddress" xdb:SQLType="USADDR_T">
  <xs:complexContent>
    <xs:extension base="Address">
      <xs:sequence>
        <xs:element name="zip" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="IntlAddress" final="#all" xdb:SQLType="INTLADDR_T">
  <xs:complexContent>
    <xs:extension base="Address">
      <xs:sequence>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
</xs:schema>';
begin
  dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;

```

Note: Type INTLADDR_T is created as a *final* type because the corresponding complexType specifies the "final" attribute. By default, all complexTypes can be extended and restricted by other types, and hence, all SQL object types are created as *not final* types.

```
create type ADDR_T as object (  
    SYS_XDBPD$ XDB.XDB$RAW_LIST_T,  
    "street" varchar2(4000),  
    "city" varchar2(4000)  
) not final;  
  
create type USADDR_T under ADDR_T (  
    "zip" varchar2(4000)  
) not final;  
  
create type INTLADDR_T under ADDR_T (  
    "country" varchar2(4000)  
) final;
```

Example 5–18 Inheritance in XML Schema: Restrictions in complexTypes

Consider an XML schema that defines a base complexType Address and a restricted type LocalAddress that prohibits the specification of country attribute.

```
declare  
    doc varchar2(3000) := '<xs:schema  
xmlns:xs="http://www.w3.org/2001/XMLSchema"  
    xmlns:xdb="http://xmlns.oracle.com/xdb">  
  <xs:complexType name="Address" xdb:SQLType="ADDR_T">  
    <xs:sequence>  
      <xs:element name="street" type="xs:string"/>  
      <xs:element name="city" type="xs:string"/>  
      <xs:element name="zip" type="xs:string"/>  
      <xs:element name="country" type="xs:string" minOccurs="0" maxOccurs="1"/>  
    </xs:sequence>  
  </xs:complexType>  
  
  <xs:complexType name="LocalAddress" xdb:SQLType="USADDR_T">  
    <xs:complexContent>  
      <xs:restriction base="Address">  
        <xs:sequence>  
          <xs:element name="street" type="xs:string"/>  

```

```

        <xs:element name="city" type="xs:string"/>
        <xs:element name="zip" type="xs:string"/>
        <xs:element name="country" type="xs:string"
            minOccurs="0" maxOccurs="0"/>
    </xs:sequence>
</xs:restriction>
</xs:complexContent>
</xs:complexType>
</xs:schema>';
begin
    dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;

```

Since inheritance support in SQL does not support a notion of restriction, the SQL type corresponding to the restricted complexType is a empty subtype of the parent object type. For the preceding XML schema, the following SQL types are generated:

```

create type ADDR_T as object (
    SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
    "street" varchar2(4000),
    "city" varchar2(4000),
    "zip" varchar2(4000),
    "country" varchar2(4000)
) not final;

create type USADDR_T under ADDR_T;

```

Mapping complexType: simpleContent to Object Types

A complexType based on a simpleContent declaration is mapped to an object type with attributes corresponding to the XML attributes and an extra SYS_XDBBODY attribute corresponding to the body value. The datatype of the body attribute is based on simpleType which defines the body's type.

Example 5–19 XML Schema complexType: Mapping complexType to simpleContent

```

declare
    doc varchar2(3000) := '<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/emp.xsd"
xmlns:emp="http://www.oracle.com/emp.xsd"
xmlns:xdb="http://xmlns.oracle.com/xdb">
<complexType name="name" xdb:SQLType="OBJ_T">
    <simpleContent>
        <restriction base = "string">
            </restriction>

```

```

    </simpleContent>
</complexType>
</schema>' ;
begin
    dbms_xmlschema.registerSchema('http://www.oracle.com/emp.xsd', doc);
end;
```

On registering this XML schema, Oracle XML DB generates the following types and XMLType tables:

```

create type OBJ_T as object
(
    SYS_XDBPD$ xdb.xdb$raw_list_t,
    SYS_XDBBODY$ VARCHAR2(4000)
);
```

Mapping complexType: Any and AnyAttributes

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.

Example 5–20 Oracle XML DB XML Schema: Mapping complexType to Any/AnyAttributes

This XML schema example declares an `any` element and maps it to the column `SYS_XDBANY$`, in object type `OBJ_T`. This element also declares that the attribute, `processContents`, skips validating contents that match the `any` declaration.

```

declare
doc varchar2(3000) := '<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/any.xsd"
xmlns:emp="http://www.oracle.com/any.xsd"
xmlns:xdb="http://xmlns.oracle.com/xdb">
<complexType name = "Employee" xdb:SQLType="OBJ_T">
  <sequence>
    <element name = "Name" type = "string" />
    <element name = "Age" type = "decimal"/>
    <any namespace = "http://www.w3.org/2001/xhtml" processContents = "skip"/>
```

```

    </sequence>
  </complexType>
</schema>';
begin
    dbms_xmlschema.registerSchema('http://www.oracle.com/emp.xsd', doc);
end;

```

It results in the following statement:

```

CREATE TYPE OBJ_T AS OBJECT
(
    SYS_XDBPD$ xdb.xdb$raw_list_t,
    Name VARCHAR2(4000),
    Age NUMBER,
    SYS_XDBANY$ VARCHAR2(4000)
);

```

Handling Cycling Between complexTypes in XML Schema

Cycles in the XML schema are broken while generating the object types, because object types do not allow cycles, by introducing a REF attribute at the point at which the cycle gets completed. Thus part of the data is stored out-of-line yet still belongs to the parent XML document when it is retrieved.

Example 5-21 XML Schema: Cycling Between complexTypes

XML schemas permit cycling between definitions of complexTypes. [Figure 5-6](#) shows this example, where the definition of complexType CT1 can reference another complexType CT2, whereas the definition of CT2 references the first type CT1.

XML schemas permit cycling between definitions of complexTypes. This is an example of cycle of length 2:

```

declare
doc varchar2(3000) := '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xdb="http://xmlns.oracle.com/xdb">
  <xs:complexType name="CT1" xdb:SQLType="CT1">
    <xs:sequence>
      <xs:element name="e1" type="xs:string"/>
      <xs:element name="e2" type="CT2"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="CT2" xdb:SQLType="CT2">

```

```
<xs:sequence>
  <xs:element name="e1" type="xs:string"/>
  <xs:element name="e2" type="CT1"/>
</xs:sequence>
</xs:complexType>
</xs:schema>;
begin
  dbms_xmlschema.registerSchema('http://www.oracle.com/emp.xsd', doc);
end;
```

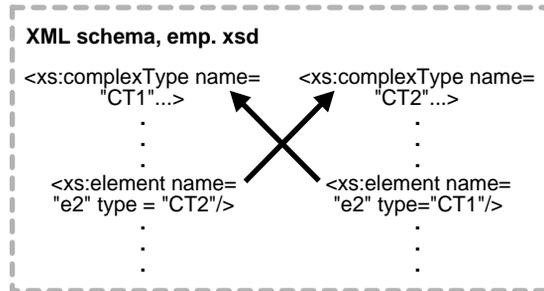
SQL types do not allow cycles in type definitions. However, they support weak cycles, that is, cycles involving REF (references) attributes. Therefore, cyclic XML schema definitions are mapped to SQL object types such that any cycles are avoided by forcing `SQLInline="false"` at the appropriate point. This creates a weak cycle.

For the preceding XML schema, the following SQL types are generated:

```
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 5–6 Cross Referencing Between Different complexTypes in the Same XML Schema



Example 5–22 XML Schema: Cycling Between complexTypes, Self-Referencing

Another example of a cyclic complexType involves the declaration of the complexType having a reference to itself. The following is an example of type `<SectionT>` that references itself:

```
declare
    doc varchar2(3000) := '<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xdb="http://xmlns.oracle.com/xdb">
  <xs:complexType name="SectionT" xdb:SQLType="SECTION_T">
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="body" type="xs:string" xdb:SQLCollType="BODY_COLL"/>
        <xs:element name="section" type="SectionT"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:schema>';
begin
    dbms_xmlschema.registerSchema('http://www.oracle.com/section.xsd', doc);
end;
```

The following SQL types are generated.

Note: The `section` attribute is declared as a varray of REFs to `XMLType` instances. Since there can be more than one occurrence of embedded sections, the attribute is a `VARRAY`. And it's a `VARRAY` of REFs to `XMLTypes` in order to avoid forming a cycle of SQL objects.

```
create type BODY_COLL as varray(32767) of VARCHAR2(4000);

create type SECTION_T as object
(
  SYS_XDBPD$ xdb.xdb$raw_list_t,
  "title" varchar2(4000),
  "body" BODY_COLL,
  "section" XDB.XDB$REF_LIST_T
) not final;
```

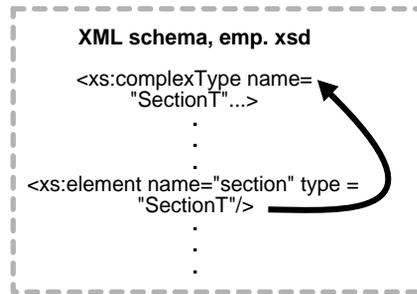
Further Guidelines for Creating XML Schema-Based XML Tables

Assume that your XML schema, identified by `"http://www.oracle.com/PO.xsd"`, has been registered. An `XMLType` table, `myPOs`, can then be created to store instances conforming to element, `PurchaseOrder`, of this XML schema, in an object-relational format as follows:

```
CREATE TABLE MyPOs OF XMLTYPE
  ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

[Figure 5-7](#) illustrates schematically how a `complexType`s can reference or cycle itself.

See Also: ["Cyclical References Between XML Schemas"](#) on page 5-72

Figure 5–7 *complexType Self Referencing Within an XML Schema*

Hidden columns are created. These correspond to the object type to which the `PurchaseOrder` element has been mapped. In addition, an `XMLExtra` object column is created to store the top-level instance data such as namespace declarations.

Note: `XMLDATA` is a pseudo-attribute of `XMLType` that enables direct access to the underlying object column. See [Chapter 4, "Using XMLType"](#), under “Changing the Storage Options on an `XMLType` Column Using `XMLData`”.

Specifying Storage Clauses in `XMLType` `CREATE TABLE` Statements

To specify storage, the underlying columns can be referenced in the `XMLType` storage clauses using either `Object` or `XML` notation:

- **Object notation:** `XMLDATA.<attr1>.<attr2>....`

For example:

```

CREATE TABLE MyPOs OF XMLTYPE
  ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder"
    lob (xmldata.lobattr) STORE AS (tablespace ...);

```

- **XML notation:** `extractValue(xmltypecol, '/attr1/attr2')`

For example:

```

CREATE TABLE MyPOs OF XMLTYPE
  ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder"
    lob (ExtractValue(MyPOs, '/lobattr')) STORE AS (tablespace ...);

```

Referencing XMLType Columns Using CREATE INDEX

As shown in the preceding examples, columns underlying an XMLType column can be referenced using either an *object* or *XML* notation in the CREATE TABLE statements. The same is true in CREATE INDEX statements:

```
CREATE INDEX ponum_idx ON MyPOs (xmldata.ponum);
CREATE INDEX ponum_idx ON MyPOs p (ExtractValue(p, '/ponum');
```

Specifying Constraints on XMLType Columns

Constraints can also be specified for underlying XMLType columns, using either the *object* or *XML* notation:

- *Object notation*

```
CREATE TABLE MyPOs OF XMLTYPE
ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder"
(unique(xmldata.ponum));
```

- *XML notation*

```
CREATE TABLE MyPOs P OF XMLTYPE
ELEMENT
"http://www.oracle.com/PO.xsd#PurchaseOrder" (unique(ExtractValue(p, '/ponum'
));
```

Inserting New Instances into XMLType Columns

New instances can be inserted into an XMLType columns as follows:

```
INSERT INTO MyPOs VALUES
(xmltype.createxml('<PurchaseOrder>.....</PurchaseOrder>'));
```

Query Rewrite with XML Schema-Based Structured Storage

What Is Query Rewrite?

When the XMLType is stored in structured storage (object-rationally) using an XML schema and queries using XPath are used, they are rewritten to go directly to the underlying object-relational columns. This enables the use of B*Tree or other indexes, if present on the column, to be used in query evaluation by the Optimizer. This query rewrite mechanism is used for XPaths in SQL functions such as existsNode(), extract(), extractValue(), and updateXML(). This

enables the XPath to be evaluated against the XML document without having to ever construct the XML document in memory.

Example 5–23 Query Rewrite

For example a query such as:

```
SELECT VALUE(p) FROM MyPOs p
       WHERE extractValue(value(p), '/PurchaseOrder/Company') = 'Oracle';
```

is trying to get the value of the `Company` element and compare it with the literal `'Oracle'`. Since the `MyPOs` table has been created with XML schema-based structured storage, the `extractValue` operator gets rewritten to the underlying relational column that stores the company information for the purchaseorder.

Thus the preceding query is rewritten to the following:

```
SELECT VALUE(p) FROM MyPOs p
       WHERE p.xmldata.company = 'Oracle';
```

See Also: [Chapter 4, "Using XMLType"](#)

If there was a regular index created on the `Company` column, such as:

```
CREATE INDEX company_index ON MyPos e
       (extractvalue(value(e), '/PurchaseOrder/Company'));
```

then the preceding query would use the index for its evaluation.

When Does Query Rewrite Occur?

Query rewrite happens for the following SQL functions:

- `extract()`
- `existsNode()`
- `extractValue`
- `updateXML`

The rewrite happens when these SQL functions are present in any expression in a query, DML, or DDL statement. For example, you can use `extractValue()` to create indexes on the underlying relational columns.

Example 5–24 SELECT Statement and Query Rewrites

This example gets the existing purchase orders:

```
SELECT EXTRACTVALUE(value(x), '/PurchaseOrder/Company')
  FROM MYPOS x
  WHERE EXISTSNODE(value(x), '/PurchaseOrder/Item[1]/Part') = 1;
```

Here are some examples of statements that get rewritten to use underlying columns:

Example 5–25 DML Statement and Query Rewrites

This example deletes all purchaseorders where the Company is not Oracle:

```
DELETE FROM MYPOS x
  WHERE EXTRACTVALUE(value(x), '/PurchaseOrder/Company') = 'Oracle Corp';
```

Example 5–26 CREATE INDEX Statement and Query Rewrites

This example creates an index on the Company column, since this is stored object relationally and the query rewrite happens, a regular index on the underlying relational column will be created:

```
CREATE INDEX company_index ON MyPos e
  (extractvalue(value(e), '/PurchaseOrder/Company'));
```

In this case, if the rewrite of the SQL functions results in a simple relational column, then the index is turned into a B*Tree or a domain index on the column, rather than a function-based index.

What XPath Expressions Are Rewritten?

XPath involving simple expressions with no wild cards or descendant axes get rewritten. The XPath may select an element or an attribute node. Predicates are supported and get rewritten into SQL predicates.

[Table 5–10](#) lists the kinds of XPath expressions that can be translated into underlying SQL queries in this release.

Table 5–10 Supported XPath Expressions for Translation to Underlying SQL Queries

XPath Expression for Translation	Description
Simple XPath expressions: /PurchaseOrder/@PurchaseDate /PurchaseOrder/Company	Involves traversals over object type attributes only, where the attributes are simple scalar or object types themselves. The only axes supported are the child and the attribute axes.
Collection traversal expressions: /PurchaseOrder/Item/Part	Involves traversal of collection expressions. The only axes supported are child and attribute axes. Collection traversal is not supported if the SQL operator is used during <code>CREATE INDEX</code> or <code>updateXML()</code> .
Predicates: [Company="Oracle"]	Predicates in the XPath are rewritten into SQL predicates. Predicates are not rewritten for <code>updateXML()</code> .
List indexes: lineitem[1]	Indexes are rewritten to access the n'th item in a collection. These are not rewritten for <code>updateXML()</code> .

Unsupported XPath Constructs The following XPath constructs do not get rewritten:

- XPath Functions
- XPath Variable references
- All axis other than child and attribute axis
- Wild card and descendant expressions
- UNION operations

Unsupported XML Schema Constructs The following XML schema constructs are not supported. This means that if the XPath expression includes nodes with the following XML schema construct then the entire expression will not get rewritten:

- XPath expressions accessing children of elements containing open content, namely `any` content. When nodes contain `any` content, then the expression cannot be rewritten, except when the `any` targets a namespace other than the namespace specified in the XPath. `any` attributes are handled in a similar way.
- CLOB storage. If the XML schema maps part of the element definitions to an SQL CLOB, then XPath expressions traversing such elements are not supported.
- Enumeration types.
- Substitutable elements.

- Non-default mapping of scalar types. For example, number types mapped to native storage, such as native integers, and so on.
- Child access for inherited `complexType`s where the child is not a member of the declared `complexType`.

For example, consider the case where we have a `address` `complexType` which has a `street` element. We can have a derived type called `shipAddr` which contains `shipmentNumber` element. If the `PurchaseOrder` had an `address` element of type `address`, then an XPath like `"/PurchaseOrder/address/street"` would get rewritten whereas `"/PurchaseOrder/address/shipmentNumber"` would not.

- Non-coercible datatype operations, such as a boolean added with a number.

How are the XPaths Rewritten?

The following sections use the same `purchaseorder` XML schema explained earlier in the chapter to explain how the functions get rewritten.

Example 5–27 Rewriting XPaths During Object Type Generation

Consider the following `purchaseorder` XML schema:

```
declare
    doc varchar2(1000) := '<schema
targetNamespace="http://www.oracle.com/PO.xsd"
xmlns:po="http://www.oracle.com/PO.xsd" xmlns="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <complexType name="PurchaseOrderType">
    <sequence>
      <element name="PONum" type="decimal"/>
      <element name="Company">
        <simpleType>
          <restriction base="string">
            <maxLength value="100"/>
          </restriction>
        </simpleType>
      </element>
      <element name="Item" maxOccurs="1000">
        <complexType>
          <sequence>
            <element name="Part">
              <simpleType>
                <restriction base="string">
```

```

        <maxLength value="1000"/>
    </restriction>
</simpleType>
</element>
<element name="Price" type="float"/>
</sequence>
</complexType>
</element>
</sequence>
</complexType>
<element name="PurchaseOrder" type="po:PurchaseOrderType"/>
</schema>';
begin
    dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;

-- A table is created conforming to this schema
CREATE TABLE MyPOs OF XMLTYPE

    ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";

-- The inserted XML document is partially validated against the schema before
-- it is inserted.
insert into MyPos values (xmltype('<PurchaseOrder
xmlns="http://www.oracle.com/PO.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
    <PONum>1001</PONum>
    <Company>Oracle Corp</Company>
    <Item>
        <Part>9i Doc Set</Part>
        <Price>2550</Price>
    </Item>
    <Item>
        <Part>8i Doc Set</Part>
        <Price>350</Price>
    </Item>
</PurchaseOrder>'));

```

Since the XML schema did not specify anything about maintaining the ordering, the default is to maintain the ordering and DOM fidelity. Hence the types have `SYS_XDBPD$` attribute to store the extra information needed to maintain the ordering of

nodes and to capture extra items such as comments, processing instructions and so on.

The `SYS_XDBPD$` attribute also maintains the existential information for the elements (that is, whether the element was present or not in the input document). This is needed for elements with scalar content, since they map to simple relational columns. In this case, both empty and missing scalar elements map to NULL values in the column and only the `SYS_XDBPD$` attribute can help distinguish the two cases. The query rewrite mechanism takes into account the presence or absence of the `SYS_XDBPD$` attribute and rewrites queries appropriately.

Assuming that this XML schema is registered with the schema URL:

```
http://www.oracle.com/PO.xsd
```

you can create the `po_tab` table with this schema as follows:

```
CREATE TABLE po_tab OF XMLTYPE
  XMLSCHEMA "http://www.oracle.com/PO.xsd" ELEMENT "PurchaseOrder";
```

Now this table has a hidden `XMLData` column of type `"PurchaseOrder_T"` that stores the actual data.

Rewriting XPath Expressions: Mapping Types and Issues

XPath expression mapping of types and topics are described in the following sections:

- ["Mapping for a Simple XPath"](#)
- ["Mapping for Scalar Nodes"](#)
- ["Mapping of Predicates"](#)
- ["Mapping of Collection Predicates"](#)
- ["Document Ordering with Collection Traversals"](#)
- ["Collection Index"](#)
- ["Non-Satisfiable XPath Expressions"](#)
- ["Namespace Handling"](#)
- ["Date Format Conversions"](#)

Mapping for a Simple XPath A rewrite for a simple XPath involves accessing the attribute corresponding to the XPath expression. [Table 5-11](#) lists the XPath map:

Table 5–11 Simple XPath Mapping for purchaseOrder XML Schema

XPath Expression	Maps to
/PurchaseOrder	column XMLData
/PurchaseOrder/@PurchaseDate	column XMLData."PurchaseDate"
/PurchaseOrder/PONum	column XMLData."PONum"
/PurchaseOrder/Item	elements of the collection XMLData."Item"
/PurchaseOrder/Item/Part	attribute "Part" in the collection XMLData."Item"

Mapping for Scalar Nodes An XPath expression can contain a `text()` operator which maps to the scalar content in the XML document. When rewriting, this maps directly to the underlying relational columns.

For example, the XPath expression `"/PurchaseOrder/PONum/text()"` maps to the SQL column `XMLData."PONum"` directly.

A NULL value in the `PONum` column implies that the text value is not available, either because the `text` node was not present in the input document or the element itself was missing. This is more efficient than accessing the scalar element, since we do not need to check for the existence of the element in the `SYS_XBBDP$` attribute.

For example, the XPath `"/PurchaseOrder/PONum"` also maps to the SQL attribute `XMLData."PONum"`,

However, in this case, query rewrite also has to check for the existence of the element itself, using the `SYS_XBBDP$` in the `XMLData` column.

Mapping of Predicates Predicates are mapped to SQL predicate expressions.

Example 5–28 Mapping Predicates

For example the predicate in the XPath expression:

```
/PurchaseOrder[PONum=1001 and Company = "Oracle Corp"]
```

maps to the SQL predicate:

```
( XMLData."PONum" = 20 and XMLData."Company" = "Oracle Corp" )
```

For example, the following query is rewritten to the structured (object-relational) equivalent, and will not require Functional evaluation of the XPath.

```
select extract(value(p), '/PurchaseOrder/Item').getClobval()
```

```
from mypos p
where existsNode(value(p), '/PurchaseOrder[PONum=1001 and Company = "Oracle
Corp"]') =1;
```

Mapping of Collection Predicates XPath expressions may involve relational operators with collection expressions. In Xpath 1.0, conditions involving collections are existential checks. In other words, even if one member of the collection satisfies the condition, the expression is true.

Example 5–29 Mapping Collection Predicates

For example the collection predicate in the XPath:

```
/PurchaseOrder[Items/Price > 200]
-- maps to a SQL collection expression:
EXISTS ( SELECT null
        FROM TABLE (XMLDATA."Item") x
        WHERE x."Price" > 200 )
```

For example, the following query is rewritten to the structured equivalent.

```
select extract(value(p), '/PurchaseOrder/Item').getClobval()
from mypos p
where existsNode(value(p), '/PurchaseOrder[Item/Price > 400]') = 1;
```

More complicated rewrites occur when you have a collection <condition> collection. In this case, if at least one combination of nodes from these two collection arguments satisfy the condition, then the predicate is deemed to be satisfied.

Example 5–30 Mapping Collection Predicates, Using existsNode()

For example, consider a fictitious XPath which checks to see if a Purchaseorder has Items such that the price of an item is the same as some part number:

```
/PurchaseOrder[Items/Price = Items/Part]
-- maps to a SQL collection expression:
EXISTS ( SELECT null
        FROM TABLE (XMLDATA."Item") x
        WHERE EXISTS ( SELECT null
                      FROM TABLE(XMLDATA."Item") y
                      WHERE y."Part" = x."Price" ))
```

For example, the following query is rewritten to the structured equivalent:

```
select extract(value(p), '/PurchaseOrder/Item').getClobval()
from mypos p
```

```
where existsNode(value(p), '/PurchaseOrder[Item/Price = Item/Part]') = 1;
```

Document Ordering with Collection Traversals Most of the rewrite preserves the original document ordering. However, since the SQL system does not guarantee ordering on the results of subqueries, when selecting elements from a collection using the `extract()` function, the resultant nodes may not be in document order.

Example 5–31 Document Ordering with Collection Traversals

For example:

```
SELECT extract(value(p), '/PurchaseOrder/Item[Price>2100]/Part')
FROM mypos p;
```

is rewritten to use subqueries as shown in the following:

```
SELECT (SELECT XMLAgg( XMLForest(x."Part" AS "Part"))
        FROM TABLE (XMLData."Item") x
        WHERE x."Price" > 2100 )
FROM po_tab p;
```

Though in most cases, the result of the aggregation would be in the same order as the collection elements, this is not guaranteed and hence the results may not be in document order. This is a limitation that may be fixed in future releases.

Collection Index An XPath expression can also access a particular index of a collection. For example, `"/PurchaseOrder/Item[1]/Part"` is rewritten to extract out the first Item of the collection and then access the Part attribute within that.

If the collection has been stored as a VARRAY, then this operation retrieves the nodes in the same order as present in the original document. If the mapping of the collection is to a nested table, then the order is undetermined. If the VARRAY is stored as an Ordered Collection Table (OCT), (the default for the tables created by the schema compiler, if `storeVarrayAsTable="true"` is set), then this collection index access is optimized to use the IOT index present on the VARRAY.

Non-Satisfiable XPath Expressions An XPath expression can contain references to nodes that cannot be present in the input document. Such parts of the expression map to SQL NULLs during rewrite. For example the XPath expression: `"/PurchaseOrder/ShipAddress"` cannot be satisfied by any instance document conforming to the `PO.xsd` XML schema, since the XML schema does not allow for `ShipAddress` elements under `PurchaseOrder`. Hence this expression would map to a SQL NULL literal.

Namespace Handling Namespaces are handled in the same way as the function-based evaluation. For schema based documents, if the function (like `existsNode()` or `extract()`) does not specify any namespace parameter, then the target namespace of the schema is used as the default namespace for the XPath expression.

Example 5–32 Handling Namespaces

For example, the XPath expression `/PurchaseOrder/PONum` is treated as `/a:PurchaseOrder/a:PONum` with `xmlns:a="http://www.oracle.com/PO.xsd"` if the SQL function does not explicitly specify the namespace prefix and mapping. In other words:

```
SELECT * FROM po_tab p
       WHERE EXISTSNODE(value(p), '/PurchaseOrder/PONum') = 1;
```

is equivalent to the query:

```
SELECT * FROM po_tab p
       WHERE EXISTSNODE(value(p), '/PurchaseOrder/PONum',
                        'xmlns="http://www.oracle.com/PO.xsd") = 1;
```

When performing query rewrite, the namespace for a particular element is matched with that of the XML schema definition. If the XML schema contains `elementFormDefault="qualified"` then each node in the XPath expression must target a namespace (this can be done using a default namespace specification or by prefixing each node with a namespace prefix).

If the `elementFormDefault` is unqualified (which is the default), then only the node that defines the namespace should contain a prefix. For instance if the `PO.xsd` had the element form to be unqualified, then the `existsNode()` function should be rewritten as:

```
EXISTSNODE(value(p), '/a:PurchaseOrder/PONum',
            'xmlns:a="http://www.oracle.com/PO.xsd") = 1;
```

Note: For the case where `elementFormDefault` is unqualified, omitting the namespace parameter in the SQL function `existsNode()` in the preceding example, would cause each node to default to the target namespace. This would not match the XML schema definition and consequently would not return any result. This is true whether the function is rewritten or not.

Date Format Conversions The default date formats are different for XML schema and SQL. Consequently, when rewriting XPath expressions involving comparisons with dates, you need to use XML formats.

Example 5–33 Date Format Conversions

For example, the expression:

```
[@PurchaseDate="2002-02-01"]
```

cannot be simply rewritten as:

```
XMLData."PurchaseDate" = "2002-02-01"
```

since the default date format for SQL is not YYYY-MM-DD. Hence during query rewrite, the XML format string is added to convert text values into date datatypes correctly. Thus the preceding predicate would be rewritten as:

```
XMLData."PurchaseDate" = TO_DATE("2002-02-01", "YYYY-MM-DD");
```

Similarly when converting these columns to text values (needed for `extract()`, and so on), XML format strings are added to convert them to the same date format as XML.

XPath Expression Rewrites for `existsNode()`

`existsNode()` returns a numerical value 0 or 1 indicating if the XPath returns any nodes (`text()` or `element` nodes). Based on the mapping discussed in the earlier section, an `existsNode()` simply checks if a scalar element is non-NULL in the case where the XPath targets a `text()` node or a non-scalar node and checks for the existence of the element using the `SYS_XDBPD$` otherwise. If the `SYS_XDBPD$` attribute is absent, then the existence of a scalar node is determined by the NULL information for the scalar column.

existsNode Mapping with Document Order Maintained Table 5–12 shows the mapping of various XPaths in the case of `existsNode()` when document ordering is preserved, that is, when `SYS_XDBPD$` exists and `maintainDOM="true"` in the schema document.

Table 5–12 XPath Mapping for existsNode() with Document Ordering Preserved

XPath Expression	Maps to
/PurchaseOrder	CASE WHEN XMLData IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/@PurchaseDate	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD\$, 'PurchaseDate') = 1 THEN 1 ELSE 0 END
/PurchaseOrder/PONum	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD\$, 'PONum') = 1 THEN 1 ELSE 0 END
/PurchaseOrder[PONum = 2100]	CASE WHEN XMLData."PONum" = 2100 THEN 1 ELSE 0
/PurchaseOrder[PONum = 2100]/@PurchaseDate	CASE WHEN XMLData."PONum" = 2100 AND Check_Node_Exists(XMLData.SYS_XDBPD\$, 'PurchaseDate') = 1 THEN 1 ELSE 0 END
/PurchaseOrder/PONum/text()	CASE WHEN XMLData."PONum" IS NOT NULL THEN 1 ELSE 0
/PurchaseOrder/Item	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE value(x) IS NOT NULL) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE Check_Node_Exists(x.SYS_XDBPD\$, 'Part') = 1) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part/text()	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE x."Part" IS NOT NULL) THEN 1 ELSE 0 END

Example 5–34 existsNode Mapping with Document Order Maintained

Using the preceding mapping, a query which checks whether the purchaseorder with number 2100 contains a part with price greater than 2000:

```
SELECT count(*)
FROM mypos p
WHERE EXISTSNODE(value(p), '/PurchaseOrder[PONum=1001 and Item/Price > 2000]')=
1;
```

would become:

```
SELECT count(*)
```

```

FROM mypos p
WHERE CASE WHEN
      p.XMLData."PONum" = 1001 AND
      EXISTS ( SELECT NULL
                FROM TABLE ( XMLData."Item") p
                WHERE p."Price" > 2000 ) THEN 1 ELSE 0 END = 1;

```

The CASE expression gets further optimized due to the constant relational equality expressions and this query becomes:

```

SELECT count(*)
FROM mypos p
WHERE p.XMLData."PONum" = 1001 AND
      EXISTS ( SELECT NULL
                FROM TABLE ( p.XMLData."Item") x
                WHERE x."Price" > 2000 );

```

which would use relational indexes for its evaluation, if present on the Part and PONum columns.

existsNode Mapping Without Maintaining Document Order If the SYS_XDBPD\$ does not exist (that is, if the XML schema specifies maintainDOM="false") then NULL scalar columns map to non-existent scalar elements. Hence you do not need to check for the node existence using the SYS_XDBPD\$ attribute. [Table 5-13](#) shows the mapping of existsNode() in the absence of the SYS_XDBPD\$ attribute.

Table 5-13 XPath Mapping for existsNode Without Document Ordering

XPath Expression	Maps to
/PurchaseOrder	CASE WHEN XMLData IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/@PurchaseDate	CASE WHEN XMLData.'PurchaseDate' IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/PONum	CASE WHEN XMLData."PONum" IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder[PONum = 2100]	CASE WHEN XMLData."PONum" = 2100 THEN 1 ELSE 0 END
/PurchaseOrder[PONum = 2100]/@PurchaseOrderDate	CASE WHEN XMLData."PONum" = 2100 AND XMLData."PurchaseDate" NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/PONum/text()	CASE WHEN XMLData."PONum" IS NOT NULL THEN 1 ELSE 0 END

Table 5–13 XPath Mapping for existsNode Without Document Ordering (Cont.)

XPath Expression	Maps to
/PurchaseOrder/Item	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE value(x) IS NOT NULL) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE x."Part" IS NOT NULL) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part/text()	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE x."Part" IS NOT NULL) THEN 1 ELSE 0 END

Rewrite for extractValue()

`extractValue()` is a shortcut for extracting text nodes and attributes using `extract()` and then using a `getStringVal()` or `getNumberVal()` to get the scalar content. `extractValue` returns the text nodes for scalar elements or the values of attribute nodes. `extractValue()` cannot handle returning multiple values or non-scalar elements.

Table 5–14 shows the mapping of various XPath expressions in the case of `extractValue()`. If an XPath expression targets an element, `extractValue` retrieves the text node child of the element. Thus the two XPath expressions, `/PurchaseOrder/PONum` and `/PurchaseOrder/PONum/text()` are handled identically by `extractValue` and both of them retrieve the scalar content of `PONum`.

Table 5–14 XPath Mapping for extractValue()

XPath Expression	Maps to
/PurchaseOrder	Not supported - ExtractValue can only retrieve values for scalar elements and attributes
/PurchaseOrder/@PurchaseDate	XMLData."PurchaseDate"
/PurchaseOrder/PONum	XMLData."PONum"
/PurchaseOrder[PONum = 2100]	(SELECT TO_XML(x.XMLData) FROM Dual WHERE x."PONum" = 2100)

Table 5–14 XPath Mapping for extractValue() (Cont.)

XPath Expression	Maps to
/PurchaseOrder[PONum = 2100]/@PurchaseDate	(SELECT x.XMLData."PurchaseDate") FROM Dual WHERE x."PONum" = 2100)
/PurchaseOrder/PONum/text()	XMLData."PONum"
/PurchaseOrder/Item	Not supported - ExtractValue can only retrieve values for scalar elements and attributes
/PurchaseOrder/Item/Part	Not supported - ExtractValue cannot retrieve multiple scalar values
/PurchaseOrder/Item/Part/text()	Not supported - ExtractValue cannot retrieve multiple scalar values

Example 5–35 Rewriting extractValue()

For example, an SQL query such as:

```
SELECT ExtractValue(value(p), '/PurchaseOrder/PONum')
FROM   mypos p
WHERE  ExtractValue(value(p), '/PurchaseOrder/PONum') = 1001;
```

would become:

```
SELECT p.XMLData."PONum"
FROM   mypos p
WHERE  p.XMLData."PONum" = 1001;
```

Since it gets rewritten to simple scalar columns, indexes if any, on the PONum attribute may be used to satisfy the query.

Creating Indexes ExtractValue can be used in index expressions. If the expression gets rewritten into scalar columns, then the index is turned into a B*Tree index instead of a function-based index.

Example 5–36 Creating Indexes with extract

For example:

```
create index my_po_index on mypos x
  (Extract(value(x), '/PurchaseOrder/PONum/text()').getnumberval());
```

would get rewritten into:

```
create index my_po_index on mypos x ( x.XMLData."PONum");
```

and thus becomes a regular B*Tree index. This is useful, since unlike a function-based index, the same index can now satisfy queries which target the column such as:

```
EXISTSNODE(value(x), '/PurchaseOrder[PONum=1001]') = 1;
```

Rewrite for extract()

`extract()` retrieves the results of XPath as XML. The rewrite for `extract()` is similar to that of `extractValue()` for those XPath expressions involving text nodes.

Extract Mapping with Document Order Maintained Table 5–15 shows the mapping of various XPath in the case of `extract()` when document order is preserved (that is, when `SYS_XDBPD$` exists and `maintainDOM="true"` in the schema document).

Note: The examples show `XMLElement()` and `XMLForest()` with an empty alias string "" to indicate that you create a XML instance with only text values. This is shown for illustration only.

Table 5–15 XPath Mapping for extract() with Document Ordering Preserved

XPath	Maps to
/PurchaseOrder	XMLForest(XMLData as "PurchaseOrder")
/PurchaseOrder/@PurchaseDate	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD\$, 'PurchaseDate') = 1 THEN XMLElement("", XMLData."PurchaseDate") ELSE NULL END
/PurchaseOrder/PONum	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD\$, 'PONum') = 1 THEN XMLElement("PONum", XMLData."PONum") ELSE NULL END
/PurchaseOrder[PONum = 2100]	(SELECT XMLForest(XMLData as "PurchaseOrder") from Dual where x."PONum" = 2100)
/PurchaseOrder[PONum = 2100]/@PurchaseDate	(SELECT CASE WHEN Check_Node_Exists(x.XMLData.SYS_XDBPD\$, 'PurchaseDate') = 1 THEN XMLElement("", XMLData."PurchaseDate") ELSE NULL END from Dual where x."PONum" = 2100)

Table 5–15 XPath Mapping for extract() with Document Ordering Preserved (Cont.)

XPath	Maps to
/PurchaseOrder/PONum/text()	XMLElement("", XMLData.PONum)
/PurchaseOrder/Item	(SELECT XMLAgg(XMLForest(value(p) as "Item")) from TABLE (x.XMLData."Item") p where value(p) IS NOT NULL)
/PurchaseOrder/Item/Part	(SELECT XMLAgg(CASE WHEN Check_Node_Exists(p.SYS_XDBPD\$, 'Part') = 1 THEN XMLForest(p."Part" as "Part") ELSE NULL END) from TABLE (x.XMLData."Item") p)
/PurchaseOrder/Item/Part/text()	(SELECT XMLAgg(XMLElement(" ", p."Part") from TABLE (x.XMLData."Item") x)

Example 5–37 XPath Mapping for extract() with Document Ordering Preserved

Using the mapping in [Table 5–15](#), a query that extracts the PONum element where the purchaseorder contains a part with price greater than 2000:

```
SELECT Extract(value(p), '/PurchaseOrder[Item/Part > 2000]/PONum')
FROM po_tab p;
```

would become:

```
SELECT (SELECT CASE WHEN Check_Node_Exists(p.XMLData.SYS_XDBPD$, 'PONum') = 1
THEN XMLElement("PONum", p.XMLData."PONum")
ELSE NULL END)
FROM DUAL
WHERE EXISTS( SELECT NULL
FROM TABLE ( XMLData."Item" ) p
WHERE p."Part" > 2000)
)
FROM po_tab p;
```

Check_Node_Exists is an internal function that is for illustration purposes only.

Extract Mapping Without Maintaining Document Order If the SYS_XDBPD\$ does not exist, that is, if the XML schema specifies maintainDOM="false", then NULL scalar columns map to non-existent scalar elements. Hence you do not need to check for the node existence using the SYS_XDBPD\$ attribute. [Table 5–16](#) shows the mapping of existsNode() in the absence of the SYS_XDBPD\$ attribute.

Table 5–16 XPath Mapping for extract() Without Document Ordering Preserved

XPath	Equivalent to
/PurchaseOrder	XMLForest(XMLData AS "PurchaseOrder")
/PurchaseOrder/@PurchaseDate	XMLForest(XMLData."PurchaseDate" AS "")
/PurchaseOrder/PONum	XMLForest(XMLData."PONum" AS "PONum")
/PurchaseOrder[PONum = 2100]	(SELECT XMLForest(XMLData AS "PurchaseOrder") from Dual where x."PONum" = 2100)
/PurchaseOrder[PONum = 2100]/@PurchaseDate	(SELECT XMLForest(XMLData."PurchaseDate" AS "") from Dual where x."PONum" = 2100)
/PurchaseOrder/PONum/text()	XMLForest(XMLData.PONum AS "")
/PurchaseOrder/Item	(SELECT XMLAgg(XMLForest(value(p) as "Item") from TABLE (x.XMLData."Item") p where value(p) IS NOT NULL)
/PurchaseOrder/Item/Part	(SELECT XMLAgg(XMLForest(p."Part" AS "Part") from TABLE (x.XMLData."Item") p)
/PurchaseOrder/Item/Part/text()	(SELECT XMLAgg(XMLForest(p."Part" AS "Part") from TABLE (x.XMLData."Item") p)

Optimizing Updates Using updateXML()

A regular update using `updateXML()` involves updating a value of the XML document and then replacing the whole document with the newly updated document.

When `XMLType` is stored object relationally, using XML schema mapping, updates are optimized to directly update pieces of the document. For example, updating the `PONum` element value can be rewritten to directly update the `XMLData.PONum` column instead of materializing the whole document in memory and then performing the update.

`updateXML()` must satisfy the following conditions for it to use the optimization:

- The `XMLType` column supplied to `updateXML()` must be the same column being updated in the SET clause. For example:

```
UPDATE po_tab p SET value(p) = updatexml(value(p),...);
```

- The `XMLType` column must have been stored object relationally using Oracle XML DB's XML schema mapping.
- The XPath expressions must not involve any predicates or collection traversals.
- There must be no duplicate scalar expressions.
- All XPath arguments in the `updateXML()` function must target only scalar content, that is, text nodes or attributes. For example:

```
UPDATE po_tab p SET value(p) =
    updatexml(value(p), '/PurchaseOrder/@PurchaseDate', '2002-01-02',
              '/PurchaseOrder/PONum/text()', 2200);
```

If all the preceding conditions are satisfied, then the `updateXML` is rewritten into a simple relational update. For example:

```
UPDATE po_tab p SET value(p) =
    updatexml(value(p), '/PurchaseOrder/@PurchaseDate', '2002-01-02',
              '/PurchaseOrder/PONum/text()', 2200);
```

becomes:

```
UPDATE po_tab p
    SET p.XMLData."PurchaseDate" = TO_DATE('2002-01-02', 'YYYY-MM-DD'),
        p.XMLData."PONum" = 2100;
```

DATE Conversions Date datatypes such as `DATE`, `gMONTH`, `gDATE`, and so on, have different format in XML schema and SQL. In such cases, if the `updateXML()` has a string value for these columns, the rewrite automatically puts the XML format string to convert the string value correctly. Thus string value specified for `DATE` columns, must match the XML date format and not the SQL `DATE` format.

Creating Default Tables During XML Schema Registration

As part of XML schema registration, you can also create default tables. Default tables are most useful when XML instance documents conforming to this XML schema are inserted through APIs that do not have any table specification, such as with FTP or HTTP. In such cases, the XML instance is inserted into the default table.

If you have given a value for attribute `defaultTable`, the `XMLType` table is created with that name. Otherwise it gets created with an internally generated name.

Further, any text specified using the `tableProps` and `columnProps` attribute are appended to the generated `CREATE TABLE` statement.

Ordered Collections in Tables (OCTs)

Arrays in XML schemas (elements with `maxOccurs > 1`) are usually stored in VARRAYs, which can be stored either in a Large Object (LOB) or in a separate store table, similar to a nested table.

Note: When elements of a VARRAY are stored in a separate table, the VARRAY is referred to as an Ordered Collection in Tables (OCT). In the following paragraphs, references to OCT also assume that you are using Index Organized Table (IOT) storage for the “store” table.

This allows the elements of a VARRAY to reside in a separate table based on an IOT. The primary key of the table is (`NESTED_TABLE_ID`, `ARRAY_INDEX`). `NESTED_TABLE_ID` is used to link the element with their containing parents while the `ARRAY_INDEX` column keeps track of the position of the element within the collection.

Using OCT for VARRAY Storage

There are two ways to specify an OCT storage:

- By means of the schema attribute “`storeVarrayAsTable`”. By default this is “false” and VARRAYs are stored in a LOB. If this is set to “true”, all VARRAYs, all elements that have `maxOccurs > 1`, will be stored as OCTs.
- By explicitly specifying the storage using the “`tableProps`” attribute. The exact SQL needed to create an OCT can be used as part of the `tableProps` attribute:

```
"VARRAY xmldata.<array> STORE AS TABLE <myTable> ((PRIMARY KEY (NESTED_
TABLE_ID, ARRAY_INDEX)) ORGANIZATION INDEX)"
```

The advantages of using OCTs for VARRAY storage include faster access to elements and better queryability. Indexes can be created on attributes of the element and these can aid in better execution for query rewrite.

Cyclical References Between XML Schemas

XML schema documents can have cyclic dependencies that can prevent them from being registered one after the other in the usual manner. Examples of such XML schemas follow:

Example 5-38 Cyclic Dependencies

An XML schema that includes another xml schema cannot be created if the included xml schema does not exist.

```
begin dbms_xmlschema.registerSchema('xm40.xsd',
'

```

It can however be created with the FORCE option:

```
begin dbms_xmlschema.registerSchema('xm40.xsd',
'

```

Attempts to use this schema and recompile will fail:

```
create table foo of sys.xmltype xmlschema "xm40.xsd" element "Emp";
```

Now create the second XML schema with `FORCE` option. This should also make the first XML schema valid:

```
begin dbms_xmlschema.registerSchema('xm40a.xsd',
'<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:my="xm40"
targetNamespace="xm40">
  <include schemaLocation="xm40.xsd"/>
  <!-- Define a global complextype here -->
  <complexType name="Employee">
    <sequence>
      <element name="Name" type="string"/>
      <element name="Age" type="positiveInteger"/>
      <element name="Phone" type="string"/>
    </sequence>
  </complexType>
  <!-- Define a global element depending on included schema -->
  <element name="Comp" type="my:Company"/>
</schema>',
true, true, false, true, true); end;
/
```

Both XML schemas can be used to create tables, and so on:

```
create table foo of sys.xmltype xmlschema "xm40.xsd" element "Emp";
create table foo2 of sys.xmltype xmlschema "xm40a.xsd" element "Comp";
```

To register both these XML schemas which have a cyclic dependency on each other, you must use the `FORCE` parameter in `DBMS_XMLSCHEMA.registerSchema` as follows:

1. Step 1 : Register "s1.xsd" in `FORCE` mode:

```
dbms_xmlschema.registerSchema("s1.xsd", "<schema ...", ..., force => true)
```

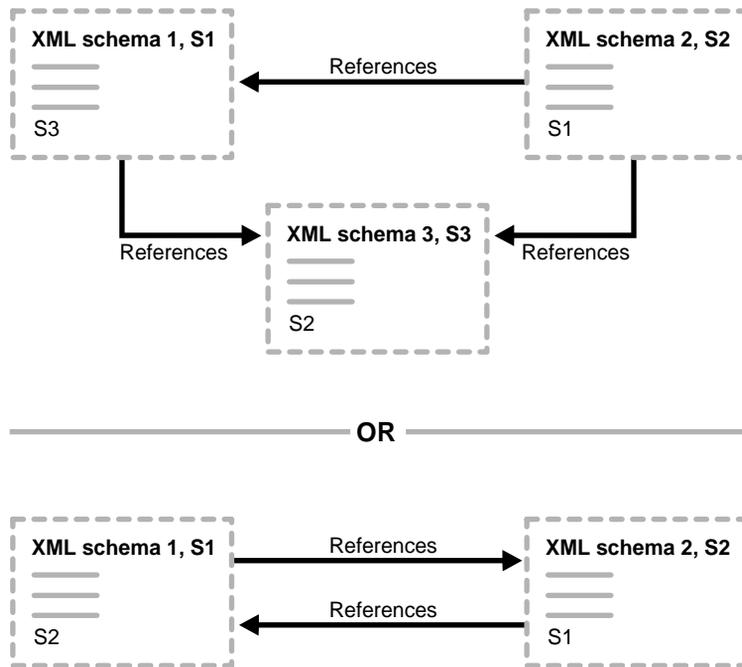
At this point, `s1.xsd` is invalid and cannot be used.

2. Step 2 : Register "s2.xsd" in `FORCE` mode:

```
dbms_xmlschema.registerSchema("s2.xsd", "<schema ..", ..., force => true)
```

The second operation automatically compiles `s1.xsd` and makes both XML schemas valid.

See [Figure 5-8](#). The preceding example is illustrated in the lower half of the figure.

Figure 5–8 *Cyclical References Between XML Schemas*

Transforming and Validating XMLType Data

This chapter describes the SQL functions and `XMLType` APIs for transforming `XMLType` data using XSLT stylesheets. It also explains the various functions and APIs available for validating the `XMLType` instance against an XML schema. It contains the following sections:

- [Transforming XMLType Instances](#)
- [XMLTransform\(\) Examples](#)
- [Validating XMLType Instances](#)
- [Validating XML Data Stored as XMLType: Examples](#)

Transforming XMLType Instances

XML documents have structure but no format. To add format to the XML documents you can use Extensible Stylesheet Language (XSL). XSL provides a way of displaying XML semantics. It can map XML elements into other formatting or mark-up languages such as HTML.

In Oracle XML DB, XMLType instances or XML data stored in XMLType tables, columns, or views in Oracle9i database, can be (formatted) transformed into HTML, XML, and other mark-up languages, using XSL stylesheets and XMLType's function, transform(). This process conforms to W3C's XSLT 1.1 recommendation.

XMLType instance can be transformed in the following ways:

- Using the XMLTransform() SQL function (or the transform() member function of XMLType) in the database
- Using XDK transformation options in the middle tier, such as XSLT Processor for Java.

See Also:

- [Appendix D, "XSLT Primer"](#)
- *Oracle9i XML Developer's Kits Guide - XDK*, the chapter on XSQL Pages Publishing Framework.

XMLTransform() and XMLType.transform()

[Figure 6-1](#) shows the XMLTransform() syntax. The XMLTransform() function takes as arguments an XMLType instance and an XSL stylesheet (which is itself an XMLType instance). It applies the stylesheet to the instance and returns an XMLType instance.

Note: You can also use the syntax, XMLTYPE.transform(). This is the same as XMLtransform().

[Figure 6-2](#) shows how XMLTransform() transforms the XML document by using the XSL stylesheet passed in. It returns the processed output as XML, HTML, and so on, as specified by the XSL stylesheet. You typically need to use XMLTransform() when retrieving or generating XML documents stored as XMLType in Oracle9i database.

See Also: Figure 1-1, "Oracle XML DB Architecture: XMLType Storage and Repository" in Chapter 1, "Introducing Oracle XML DB"

Figure 6-1 XMLTransform() Syntax

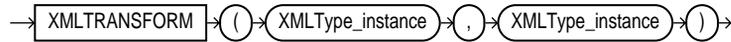
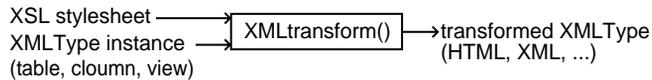


Figure 6-2 Using XMLTransform()

XMLType function



XMLTransform() Examples

Use the following code to set up the XML schema and tables needed to run the examples in this chapter:

```

--register schema
begin
  dbms_xmlschema.deleteSchema('http://www.example.com/schemas/ipo.xsd',4);
end;
/
begin
  dbms_xmlschema.registerSchema('http://www.example.com/schemas/ipo.xsd',
    '<schema targetNamespace="http://www.example.com/IPO"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:ipo="http://www.example.com/IPO">
    <!-- annotation>
    <documentation xml:lang="en">
      International Purchase order schema for Example.com
      Copyright 2000 Example.com. All rights reserved.
    </documentation>
    </annotation -->
    <element name="purchaseOrder" type="ipo:PurchaseOrderType"/>
    <element name="comment" type="string"/>
    <complexType name="PurchaseOrderType">

```

```
<sequence>
  <element name="shipTo"      type="ipo:Address"/>
  <element name="billTo"      type="ipo:Address"/>
  <element ref="ipo:comment"  minOccurs="0"/>
  <element name="items"      type="ipo:Items"/>
</sequence>
<attribute name="orderDate" type="date"/>
</complexType>
<complexType name="Items">
  <sequence>
    <element name="item" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="productName" type="string"/>
          <element name="quantity">
            <simpleType>
              <restriction base="positiveInteger">
                <maxExclusive value="100"/>
              </restriction>
            </simpleType>
          </element>
          <element name="USPrice" type="decimal"/>
          <element ref="ipo:comment" minOccurs="0"/>
          <element name="shipDate" type="date" minOccurs="0"/>
        </sequence>
        <attribute name="partNum" type="ipo:SKU" use="required"/>
      </complexType>
    </element>
  </sequence>
</complexType>
<complexType name="Address">
  <sequence>
    <element name="name" type="string"/>
    <element name="street" type="string"/>
    <element name="city" type="string"/>
    <element name="state" type="string"/>
    <element name="country" type="string"/>
    <element name="zip" type="string"/>
  </sequence>
</complexType>
<simpleType name="SKU">
  <restriction base="string">
    <pattern value="\{3\}-[A-Z]{2}"/>
  </restriction>
</simpleType>
```

```

</schema>',
    TRUE, TRUE, FALSE);
end;
/

-- create table to hold XML instance documents
DROP TABLE po_tab;
CREATE TABLE po_tab (id number, xmlcol xmltype)
XMLTYPE COLUMN xmlcol
XMLSCHEMA "http://www.example.com/schemas/ipo.xsd"
ELEMENT "purchaseOrder";

INSERT INTO po_tab VALUES(1, xmltype(
'<?xml version="1.0"?>
<ipo:purchaseOrder
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ipo="http://www.example.com/IPO"
  xsi:schemaLocation="http://www.example.com/IPO
                    http://www.example.com/schemas/ipo.xsd"
  orderDate="1999-12-01">
  <shipTo xsi:type="ipo:Address">
    <name>Helen Zoe</name>
    <street>121 Broadway</street>
    <city>Cardiff</city>
    <state>Wales</state>
    <country>UK</country>
    <zip>CF2 1QJ</zip>
  </shipTo>
  <billTo xsi:type="ipo:Address">
    <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>');
```

The following examples illustrate how to use `XMLTransform()` to transform XML data stored as `XMLType` to HTML, XML, or other languages.

Example 6–1 Transforming an XMLType Instance Using XMLTransform() and DBUriType to Get the XSL Stylesheet

```
DROP TABLE stylesheet_tab;
CREATE TABLE stylesheet_tab(id NUMBER, stylesheet xmltype);
INSERT INTO stylesheet_tab VALUES (1, xmltype(
'<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="*">
  <td>
    <xsl:choose>
      <xsl:when test="count(child:*) > 1">
        <xsl:call-template name="nested"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="name(.)"/>:<xsl:value-of select="text()"/>
      </xsl:otherwise>
    </xsl:choose>
  </td>
</xsl:template>
<xsl:template match="*" name="nested" priority="-1" mode="nested2">
  <b>
    <!-- xsl:value-of select="count(child:*)"/ -->
    <xsl:choose>
      <xsl:when test="count(child:*) > 1">
        <xsl:value-of select="name(.)"/>:<xsl:apply-templates mode="nested2"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="name(.)"/>:<xsl:value-of select="text()"/>
      </xsl:otherwise>
    </xsl:choose>
  </b>
</xsl:template>
</xsl:stylesheet>'
));

SELECT XMLTransform(x.xmlcol,
  dburiType('/SCOTT/STYLESHEET_TAB/ROW[ID =
1]/STYLESHEET/text()').getXML()).getStringVal()
AS result
```

```

FROM po_tab x;

-- The preceding statement produces the following output:
-- RESULT
-----
-- <td>
--   <b>ipo:purchaseOrder:
--     <b>shipTo:
--       <b>name:Helen Zoe</b>
--       <b>street:100 Broadway</b>
--       <b>city:Cardiff</b>
--       <b>state:Wales</b>
--       <b>country:UK</b>
--       <b>zip:CF2 1QJ</b>
--     </b>
--     <b>billTo:
--       <b>name:Robert Smith</b>
--       <b>street:8 Oak Avenue</b>
--       <b>city:Old Town</b>
--       <b>state:CA</b>
--       <b>country:US</b>
--       <b>zip:95819</b>
--     </b>
--     <b>items:</b>
--   </b>
-- </td>

```

Example 6–2 Transforming an XMLType Instance Using XMLTransform() and a Subquery SELECT to Retrieve the XSL Stylesheet

This example illustrates the use of a stored stylesheet to transform XMLType instances. Unlike the previous example, this example uses a scalar subquery to retrieve the stored stylesheet:

```

SELECT XMLTransform(x.xmlcol,
  (select stylesheet from stylesheet_tab where id = 1)).getStringVal()
AS result
FROM po_tab x;

```

Example 6–3 Transforming XMLType Instances Using Transient Stylesheets and XMLTransform()

This example describes how you can transform XMLType instances using a transient stylesheet:

```

SELECT x.xmlcol.transform(xmltype(
'<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="*">
  <td>
    <xsl:choose>
      <xsl:when test="count(child:*) > 1">
        <xsl:call-template name="nested"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="name(.)"/><xsl:value-of select="text()"/>
      </xsl:otherwise>
    </xsl:choose>
  </td>
</xsl:template>
<xsl:template match="*" name="nested" priority="-1" mode="nested2">
  <b>
    <!-- xsl:value-of select="count(child:*)"/ -->
    <xsl:choose>
      <xsl:when test="count(child:*) > 1">
        <xsl:value-of select="name(.)"/><xsl:apply-templates mode="nested2"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="name(.)"/><xsl:value-of select="text()"/>
      </xsl:otherwise>
    </xsl:choose>
  </b>
</xsl:template>
</xsl:stylesheet>'
)).getStringVal()
FROM po_tab x;

```

Validating XMLType Instances

Often, besides knowing that a particular XML document is well-formed, it is necessary to know if a particular document conforms to a specific XML schema, that is, is VALID with respect to a specific XML schema.

By default, Oracle9i does check to make sure that XMLType instances are well-formed. In addition, for schema-based XMLType instances, Oracle9i performs few basic validation checks. Since full XML schema validation (as specified by the W3C) is an expensive operation, when XMLType instances are constructed, stored, or retrieved, they are not also fully validated.

To validate and manipulate the “validated” status of XML documents, the following functions and SQL operator are provided:

XMLIsValid()

`XMLIsValid()` is an SQL Operator. It checks if the input instance conforms to a specified XML schema. It does not change the validation status of the XML instance. If an XML schema URL is not specified and the XML document is schema-based, the conformance is checked against the `XMLType` instance’s own schema. If any of the arguments are specified to be NULL, then the result is NULL. If validation fails, 0 is returned and no errors are reported explaining why the validation has failed.

Syntax

```
XMLIsValid ( XMLType_inst [, schemaur1 [, elem]])
```

Parameters:

- `XMLType_inst` - The `XMLType` instance to be validated against the specified XML Schema.
- `schurl` - The URL of the XML Schema against which to check conformance.
- `elem` - Element of a specified schema, against which to validate. This is useful when we have a XML Schema which defines more than one top level element, and we want to check conformance against a specific one of these elements.

schemaValidate

`schemaValidate` is a member procedure. It validates the XML instance against its XML schema if it has not already been done. For non-schema-based documents an error is raised. If validation fails an error is raised otherwise, the document’s status is changed to `VALIDATED`.

Syntax

```
MEMBER PROCEDURE schemaValidate
```

isSchemaValidated()

`isSchemaValidated()` is a member function. It returns the validation status of the `XMLType` instance and tells if a schema-based instance has been actually validated against its schema. It returns 1 if the instance has been validated against the schema, 0 otherwise.

Syntax

```
MEMBER FUNCTION isSchemaValidated return NUMBER deterministic
```

setSchemaValidated()

`setSchemaValidated()` is a member function. It sets the `VALIDATION` state of the input XML instance.

Syntax

```
MEMBER PROCEDURE setSchemaValidated(flag IN BINARY_INTEGER := 1)
```

Parameters:

`flag`, 0 - NOT VALIDATED; 1 - VALIDATED; The default value for this parameter is 1.

isSchemaValid()

`isSchemaValid()` is a member function. It checks if the input instance conforms to a specified XML schema. It does not change the validation status of the XML instance. If an XML Schema URL is not specified and the XML document is schema-based, the conformance is checked against the `XMLType` instance's own schema. If the validation fails, exceptions are thrown with the reason why the validation has failed.

Syntax

```
member function isSchemaValid(schurl IN VARCHAR2 := NULL, elem IN VARCHAR2 :=  
    NULL) return NUMBER deterministic
```

Parameters:

`schurl` - The URL of the XML Schema against which to check conformance.

`elem` - Element of a specified schema, against which to validate. This is useful when we have a XML Schema which defines more than one top level element, and we want to check conformance against a specific one of these elements.

Validating XML Data Stored as XMLType: Examples

The following examples illustrate how to use `isSchemaValid()`, `setSchemaValidated()`, and `isSchemaValidated()` to validate XML data being stored as `XMLType` in Oracle XML DB.

Example 6-4 Using isSchemaValid()

```
SELECT x.xmlcol.isSchemaValid('http://www.example.com/schemas/ipo.xsd',  
    'purchaseOrder')  
FROM po_tab x;
```

Example 6–5 Validating XML Using isSchemaValid()

The following PL/SQL example validates an XML instance against XML schema PO.xsd:

```
declare
  xmldoc xmltype;
begin
  -- populate xmldoc (for example, by fetching from table)
  -- validate against XML schema
  xmldoc.isSchemaValid('http://www.oracle.com/PO.xsd');
  if xmldoc.isSchemaValid = 1 then --
  else --
  end if;
end;
```

Example 6–6 Using schemaValidate() Within Triggers

The schemaValidate() method of XMLType can be used within INSERT and UPDATE TRIGGERS to ensure that all instances stored in the table are validated against the XML schema:

```
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.sys_nc_rowinfo$;
  xmltype.schemavalidate(newxml);
END;
/
```

Example 6–7 Using XMLIsValid() Within CHECK Constraints

This example uses XMLIsValid() to:

- Verify that the XMLType instance conforms to the specified XML schema
- Ensure that the incoming XML documents are valid by using CHECK constraints

```
DROP TABLE po_tab;
CREATE TABLE po_tab OF XMLTYPE
  (CHECK (XMLIsValid(sys_nc_rowinfo$) = 1))
```

```
XMLSchema "http://www.example.com/schemas/ipo.xsd" element "purchaseOrder" ;
```

Note: The validation functions and operators described in the preceding section, facilitate validation checking. Of these, `isSchemaValid()` is the only one that throws errors that include why the validation has failed.

Searching XML Data with Oracle Text

This chapter explains the use of Oracle Text functionality in indexing and querying XML data. It contains the following sections:

- [Searching XML Data with Oracle Text](#)
- [Introducing Oracle Text](#)
- [Assumptions Made in This Chapter's Examples](#)
- [Oracle Text Users and Roles](#)
- [Querying with the CONTAINS Operator](#)
- [Using the WITHIN Operator to Narrow Query Down to Document Sections](#)
- [Introducing SECTION_GROUPS](#)
- [INPATH or HASPATH Operators Search Using XPath-Like Expressions](#)
- [Building a Query Application with Oracle Text](#)
- [Step 1. Create a Section Group Preference](#)
- [Step 2. Set the Preference's Attributes](#)
- [Step 3. Create an Index Using the Section Preference Created in Step 2](#)
- [Step 4. Create Your Query Syntax](#)
- [Presenting the Results of Your Query](#)
- [XMLType Indexing](#)
- [Using Oracle Text with Oracle XML DB](#)
- [Full-Text Search Functions in XPath Using ora:contains](#)
- [Oracle XML DB: Creating a Policy for ora:contains\(\)](#)

-
- [Oracle XML DB: Using CTXXPATH Indexes for existsNode\(\)](#)
 - [Using Oracle Text: Advanced Techniques](#)
 - [Case Study: Searching XML-Based Conference Proceedings](#)
 - [Frequently Asked Questions About Oracle Text](#)

Note: In Oracle9i, you can use the `WITHIN` or `INPATH` operators. `INPATH` was introduced in Oracle9i Release 1 (9.0.1) to handle XPath searching in XML documents. Everything you can do with the `WITHIN` operator, you can also do using `INPATH`. `INPATH` is the recommended syntax in Oracle9i Release 1 (9.0.1) and higher when searching XML data.

Searching XML Data with Oracle Text

This chapter describes the following aspects of Oracle Text:

- How to create a section group and index your XML document(s)
- How to build an XML query application with Oracle Text, to search and retrieve data from your XML document(s)
- Using Oracle Text to search `XMLType` data

Introducing Oracle Text

Note: Oracle Text is a strictly server-based implementation.

See Also: <http://otn.oracle.com/products/text>

Oracle Text (aka *interMedia Text*) can be used to search XML documents. It extends Oracle9i by indexing any text or document stored in Oracle. It can also search documents in the file system and URLs.

Oracle Text enables the following:

- *Content-based queries*, such as, finding text and documents which contain particular words, using familiar, standard SQL.
- File-based text applications to use Oracle9i to *manage text and documents* in an integrated fashion with traditional relational information.
- *Concept searching* of English language documents.
- *Theme analysis* of English language documents using the theme/gist package.
- *Highlighting hit words*. With Oracle Text, you can render a document in different ways. For example, you can present documents with query terms highlighted, either the “words” of a word query or the “themes” of an ABOUT query in English. Use the `CTX_DOC.MARKUP` or `HIGHLIGHT` procedures for this.
- With Oracle Text PL/SQL packages for *document presentation and thesaurus maintenance*.

You can query XML data stored in the database directly, without using Oracle Text. However, Oracle Text is useful for boosting query performance.

See Also :

- *Oracle Text Reference*
- *Oracle Text Application Developer's Guide*
- <http://otn.oracle.com/products/text>

Accessing Oracle Text

Oracle Text is a standard feature that comes with every Oracle9i Standard, Enterprise, and Personal edition license. It needs to be selected during installation. No special installation instructions are required.

Oracle Text is essentially a set of schema objects owned by CTXSYS. These objects are linked to the Oracle kernel. The schema objects are present when you perform an Oracle9i installation.

Oracle Text Now Supports XMLType

You can now perform Oracle Text searches on tables containing XMLType columns.

Further Oracle Text Examples

You can find more examples for Oracle Text and for creating section group indexes at the following site: <http://otn.oracle.com/products/text>

Assumptions Made in This Chapter's Examples

XML text is a VARCHAR2 or CLOB type in an Oracle9i database table with character semantics. Oracle Text can also deal with documents in a file system or in URLs, but we are not considering these document types in this chapter.

To simplify the examples included in this chapter they use a subset of the Oracle Text options and make the following assumptions:

- All XML data here is represented using US-ASCII, a 7 bit character set.
- Issues about whether a character such as "*" is treated as white space or as part of a word are not included.
- Storage characteristics of the Oracle schema object that implement the Oracle Text index are not considered.
- They focus on the SECTION GROUP parameter in the CREATE INDEX or ALTER INDEX statement. Other parameter types available for CREATE INDEX

and ALTER INDEX, are DATASTORE, FILTER, LEXER, STOPLIST, and WORDLIST.

Here is an example of using SECTION GROUP in CREATE INDEX:

```
CREATE INDEX my_index
ON my_table ( my_column )
INDEXTYPE IS ctxsys.context
PARAMETERS ( 'SECTION GROUP my_section_group' ) ;
```

- Specifically, the examples focus on using AUTO_SECTION_GROUP and XML_SECTION_GROUP, and PATH_SECTION_GROUP.
- Tagged or marked up data. In this chapter, the examples focus on how to handle XML data. Oracle Text handles many other kinds of data besides XML data.

See Also:

- *Oracle Text Application Developer's Guide*
- *Oracle Text Reference*, for more information on these parameter types.

Oracle Text Users and Roles

With Oracle Text you can use the following users/roles:

- user CTXSYS to administer users
- role CTXAPP to create and delete Oracle Text *preferences* and use Oracle Text PL/SQL packages

User CTXSYS

User CTXSYS is created at install time. Administer Oracle Text users as this user. User CTXSYS has the following privileges:

- Modify system-defined preferences
- Drop and modify other user preferences
- Call procedures in the CTX_ADM PL/SQL package to start servers and set system-parameters
- Start a ctxsrv server
- Query all system-defined views

- Perform all the tasks of a user with the CTXAPP role

Role CTXAPP

Any user can create an Oracle Text index and issue a Text query. For additional tasks, use the CTXAPP role. This is a system-defined role that enables you to perform the following tasks:

- Create and delete Oracle Text *preferences*
- Use Oracle Text PL/SQL packages, such as the CTX_DDL package

Querying with the CONTAINS Operator

Oracle Text’s main purpose is to provide an implementation for the CONTAINS operator. The CONTAINS operator can be used in the WHERE clause of a SELECT statement to specify the query expression for a Text query.

CONTAINS Syntax

Here is the CONTAINS syntax:

```
...WHERE CONTAINS([schema.]column,text_query VARCHAR2,[label NUMBER])
```

where:

Table 7–1 CONTAINS Operator: Syntax Description

Syntax	Description
[schema.] column	Specifies the text column to be searched on. This column must have a Text index associated with it.
text_query	Specifies the query expression that defines your search in column.
label	Optionally specifies the label that identifies the score generated by the CONTAINS operator.

For each row selected, CONTAINS returns a number between 0 and 100 that indicates how relevant the document row is to the query. The number 0 means that Oracle found no matches in the row. You can obtain this score with the SCORE operator.

Note: You must use the SCORE operator with a label to obtain this number.

Example 7-1 Using a Simple SELECT Statement with CONTAINS

The following example illustrates how the CONTAINS operator is used in a SELECT statement:

```
SELECT id FROM my_table
WHERE
CONTAINS (my_column, 'receipts') > 0
```

The 'receipts' parameter of the CONTAINS operator is called the "Text Query Expression".

Note: The SQL statement with the CONTAINS operator requires an Oracle Text index in order to run.

Example 7-2 Using the Score Operator with a Label to Obtain the Relevance

The following example searches for all documents in the text column that contain the word Oracle. The score for each row is selected with the SCORE operator using a label of 1:

```
SELECT SCORE(1), title from newsindex
WHERE CONTAINS(text, 'oracle', 1) > 0 ORDER BY SCORE(1) DESC;
```

The CONTAINS operator must always be followed by the > 0 syntax. This specifies that the score value calculated by the CONTAINS operator must be greater than zero for the row selected.

When the SCORE operator is called, such as in a SELECT clause, the operator must reference the label value as shown in the example.

Using the WITHIN Operator to Narrow Query Down to Document Sections

When documents have internal structure such as in HTML and XML, you can define document sections using embedded tags before you index. This enables you to query within the sections using the WITHIN operator.

Note: This is only true for XML_SECTION_GROUP, but not true for AUTO_ or PATH_SECTION_GROUP.

Introducing SECTION_GROUPS

You can query within attribute sections when you index with either XML_SECTION_GROUP, AUTO_SECTION_GROUP, or PATH_SECTION_GROUP your section group type. Consider the following XML document:

```
<book title="Tale of Two Cities">It was the best of times.</book>
```

XML_SECTION_GROUP

If you use XML_SECTION_GROUP, you can specify any of the following sections:

- Zone sections
- Field sections
- Attribute section
- Special sections

This chapter only focuses on Zone, Field, and Attribute sections. For more information on Special sections see *Oracle Text Reference* and *Oracle Text Application Developer's Guide*.

Zone Sections: CTX_DDL.ADD_ZONE_SECTION Procedure

The syntax for this is:

```
CTX_DDL.ADD_ZONE_SECTION(  
  group_name      in   varchar2,  
  section_name    in   varchar2,  
  tag             in   varchar2);
```

To define a chapter as a Zone section, create an XML_SECTION_GROUP and define the Zone section as follows:

```
EXEC ctx_ddl_create_section_group('myxmlgroup', 'XML_SECTION_GROUP');  
EXEC ctx_ddl.add_zone_section('myxmlgroup', 'chapter', 'chapter');
```

When you define Zone section as such and index the document set, you can query the XML chapter Zone section as follows:

```
'Cities within chapter'
```

Field Sections: CTX_DLL.ADD_FIELD_SECTION Procedure

The syntax for this is:

```
CTX_DLL.ADD_FIELD_SECTION(
  group_name    in    varchar2,
  section_name  in    varchar2,
  tag           in    varchar2);
```

To define a `abstract` as a Field section, create an `XML_SECTION_GROUP` and define the Field section as follows:

```
EXEC ctx_ddl_create_section_group('myxmlgroup', 'XML_SECTION_GROUP');
EXEC ctx_ddl.add_field_section('myxmlgroup', 'abstract', 'abstract');
```

When you define Field section as such and index the document set, you can query the XML `abstract` Field section as follows:

```
'Cities within abstract'
```

Attribute Section: CTX_DLL.ADD_ATTR_SECTION Procedure

The syntax for this is:

```
CTX_DLL.ADD_ATTR_SECTION(
  group_name    in    varchar2,
  section_name  in    varchar2,
  tag           in    varchar2);
```

To define the `booktitle` attribute as an Attribute section, create an `XML_SECTION_GROUP` and define the Attribute section as follows:

```
EXEC ctx_ddl_create_section_group('myxmlgroup', 'XML_SECTION_GROUP');
EXEC ctx_ddl.add_attr_section('myxmlgroup', 'booktitle', 'book@title');
```

When you define the Attribute section as such and index the document set, you can query the XML `booktitle` attribute text as follows:

```
'Cities within booktitle'
```

Constraints for Querying Attribute or Field Sections

The following constraints apply to querying within Attribute or Field sections:

- Regular queries on attribute text will not work unless qualified in a `WITHIN` clause. Using the following XML document:

```
<book title="Tale of Two Cities">It was the best of times.</book>
```

querying on Tale will not work unless qualified with 'WITHIN title@book'.

- You cannot use Attribute or Field sections in a nested WITHIN query.
- Phrases ignore attribute text. For example, if the original document looked like:

```
...Now is the time for all good <word type="noun"> men </word> to come to
the aid.....
```

The search would result in a regular query's, "good men", and ignore the intervening attribute text.

AUTO_SECTION_GROUP/ PATH_SECTION_GROUP for INPATH and HASPATH

When you use the AUTO_SECTION_GROUP or PATH_SECTION_GROUP to index XML documents, Oracle9i automatically creates sections.

To search on Tale within the Attribute section booktitle, include the following WITHIN clause in your SELECT statement:

- If you are using XML_SECTION_GROUP:


```
... WHERE CONTAINS ('Tale INPATH booktitle')>0;
```
- If you are using PATH_SECTION_GROUP


```
... WHERE CONTAINS ('Tale INPATH title@book')>0;
```

See Also: ["Distinguishing Tags Across DocTypes"](#) on page 7-49.

Dynamically Adding Sections or Stop Section Using ALTER INDEX

The syntax for ALTER INDEX is:

```
ALTER INDEX [schema.]index REBUILD [ONLINE] [PARAMETERS (paramstring)];
```

where

```
paramstring = 'replace [datastore datastore_pref]
               [filter filter_pref]
               [lexer lexer_pref]
               [wordlist wordlist_pref]
               [storage storage_pref]
               [stoplist stoplist]
               [section group section_group]
               [memory memsize]
```

```

| ...
| add zone section section_name tag tag
| add field section section_name tag tag [(VISIBLE | INVISIBLE)]
| add attr section section_name tag tag@attr
| add stop section tag'

```

The added section applies only to documents indexed after this operation. Thus for the change to take effect, you must manually re-index any existing documents that contain the tag. The index is not rebuilt by this statement.

WITHIN Syntax for Section Querying

Here is the WITHIN syntax for querying sections:

```
...WHERE CONTAINS(text,'XML WITHIN title') >0;...
```

This searches for *expression* text within a section. If you are using XML_SECTION_GROUP the following restrictions apply to the pre-defined zone, field, or attribute section:

- If section is a **zone**, expression can contain one or more WITHIN operators (nested WITHIN) whose section is a zone or special section.
- If section is a **field** or **attribute** section, expression cannot contain another WITHIN operator.

You can combine and nest WITHIN clauses. For finer grained searches of XML sections, you can use WITHIN clauses inside CONTAINS select statements.

WITHIN Operator Limitations

The WITHIN operator has the following limitations:

- You cannot embed the WITHIN clause in a phrase. For example, you cannot write: term1 WITHIN section term2
- You cannot combine WITHIN with expansion operators, such as \$! and *.
- Since WITHIN is a reserved word, you must escape the word with braces to search on it.

See Also: *Oracle Text Reference*

INPATH or HASPATH Operators Search Using XPath-Like Expressions

Path Indexing and Path Querying with Oracle Text

In Oracle9i Oracle Text introduced a new section type and new query operators which support an XPath-like query language. Indexes of type context with XML path searching are able to perform very complex section searches on XML documents. Here are the basic concepts of path indexing and path querying.

Path Indexing

Section searching is enabled by defining section groups. To use XML path searching, the Oracle Text index must be created with the new section group, `PATH_SECTION_GROUP` as follows:

```
begin
  ctx_ddl.create_section_group('mypathgroup', 'PATH_SECTION_GROUP');
end;
```

To create the Oracle Text index use this command:

```
create index order_idx on library_catalog(text)
  indextype is ctxsys.context
  parameters ('SECTION GROUP mypathgroup');
```

Path Querying

The Oracle Text path query language is based on W3C XPath. For Oracle9i Release 1 (9.0.1) and higher, you can use the `INPATH` and `HASPATH` operators to express path queries.

Using INPATH Operator for Path Searching in XML Documents

You can use `INPATH` operator to perform path searching in XML documents. [Table 7-2](#) summarizes the ways you can use the `INPATH` operator for path searching.

Table 7–2 Path Searching XML Documents Using the INPATH Operator

Path Search Feature	Syntax	Description
Simple Tag Searching	virginia INPATH (STATE) virginia INPATH (//STATE)	Finds all documents where the word “virginia” appears between <STATE> and </STATE>. The STATE element can appear at any level of the document structure.
Case-sensitivity	virginia INPATH (STATE) virginia INPATH (State)	Tags and attribute names in path searching are case-sensitive. <code>virginia INPATH STATE --</code> finds <code><STATE>virginia</STATE></code> but NOT <code><State>virginia</State></code> . To find the latter you must do <code>virginia INPATH State</code> .
Top-Level Tag Searching	virginia INPATH (Legal) virginia INPATH (/Legal) For example, the following query finds Quijote where it occurs between <order> and </order>: <code>select id from library_catalog where contains(text,'Quijote INPATH(order')) > 0;</code> Here <order> must be the top level tag.	Finds all documents where “virginia” appears in a Legal element which is the top-level tag. 'Legal' MUST be the top-level tag of the document. 'virginia' may appear anywhere in this tag regardless of other intervening tags. For example: <pre><?xml version="1.0" standalone="yes"?> <!-- <?xmlstylesheet type="text/xsl" href="/xsl/vacourtfiling(html).xsl"? --> <Legal> <CourtFiling> <Filing ID="f001" FilingType="Civil"> <LeadDocument> <CaseCaption> <CourtInformation> <Location> <Address> <AddressState>VIRGINIA</AddressState> </Address> ... </Legal></pre>

Table 7–2 Path Searching XML Documents Using the INPATH Operator (Cont.)

Path Search Feature	Syntax	Description
Any Level Tag Searching	<p>virginia INPATH (//Address)</p> <p>For example, a double slash indicates "any number of levels" down. The following query finds Quijote inside a <title> tag that occurs at the top level or any lower level:</p> <pre>select id from library_catalog where contains(text, 'Quijote INPATH(//title') > 0;</pre>	<p>'Virginia' can appear anywhere within an 'Address' tag, which may appear within any other tags. for example:</p> <pre><?xml version="1.0" standalone="yes"?> <!-- <?xml-stylesheet type="text/xsl" href="./xsl/vacourtfiling(html).xsl"? --> <Legal> <CourtFiling> <Filing ID="f001" FilingType="Civil"> <LeadDocument> <CaseCaption> <CourtInformation> <Location> <Address> <AddressState> VIRGINIA </AddressState>... </Legal></pre>
Direct Parentage Path Searching	<p>virginia INPATH (//CourtInformation/Location)</p> <p>for example:</p> <pre>select id from library_catalog where contains(text, 'virginia INPATH(order/item)) > 0;</pre>	<p>Finds all documents where "virginia" appears in a Location element which is a direct child of a CourtInformation element. For example:</p> <pre><?xml version="1.0" standalone="yes"?> <!-- <?xml-stylesheet type="text/xsl" href="./xsl/vacourtfiling(html).xsl"? --> <Legal> <CourtFiling> <Filing ID="f001" FilingType="Civil"> <LeadDocument> <CaseCaption> <CourtInformation> <Location> <Address> <AddressState> VIRGINIA </AddressState> </Address>... </CourtInformation></pre>

Table 7–2 Path Searching XML Documents Using the INPATH Operator (Cont.)

Path Search Feature	Syntax	Description
Single-Level Wildcard Searching	virginia INPATH(A*/B) 'virginia INPATH (//CaseCaption*/Location)'	<p>Finds all documents where “virginia” appears in a B element which is a grandchild of an A element. For instance, <A><D>virginia</D>. The intermediate element does not need to be an indexed XML tag. For example:</p> <pre> <?xml version="1.0" standalone="yes"?> <!-- <?xml-stylesheet type="text/xsl" href="/xsl/vacourtfiling(html).xsl"? -> <Legal> <CourtFiling> <Filing ID="f001" FilingType="Civil"> <LeadDocument> <CaseCaption> <CourtInformation> <Location> <Address> <AddressState>VIRGINIA</AddressState>... </Legal> </pre>

Table 7–2 Path Searching XML Documents Using the INPATH Operator (Cont.)

Path Search Feature	Syntax	Description
Multi-level Wildcard Searching	'virginia INPATH (Legal/*/Filing/*/CourtInformation)'	<p>'Legal' must be a top-level tag, and there must be exactly one tag-level between 'Legal' and 'Filing', and two between 'Filing' and 'CourtInformation'. 'Virginia' may then appear anywhere within 'CourtInformation'. For example:</p> <pre><?xml version="1.0" standalone="yes"?> <!-- <?xml-stylesheet type="text/xsl" href="./xsl/vacourtfiling(html).xsl"? --> <Legal> <CourtFiling> <Filing ID="f001" FilingType="Civil"> <LeadDocument> <CaseCaption> <CourtInformation> <Location> <Address> <AddressState>VIRGINIA</AddressState> </Address> </Location> <CourtName> IN THE CIRCUIT COURT OF LOUDOUN COUNTY </CourtName> </CourtInformation>....</pre>
Descendant Searching	virginia INPATH(A/B)	Finds all documents where “virginia” appears in a B element which is some descendant (any level) of an A element.
Attribute Searching	virginia INPATH(A/@B)	<p>Finds all documents where “virginia” appears in the B attribute of an A element. You can search within an attribute value using the syntax <code><tag>/@<attribute></code>:</p> <pre>select id from library_catalog where contains(text,'dvd INPATH(//item/@type')) > 0; AND and OR</pre> <p>You can use boolean AND and OR to combine existence or equality predicates in a test.</p> <pre>select id from library_catalog where contains(text,'Levy or Cervantes INPATH(//title)') >0;</pre>

Table 7–2 Path Searching XML Documents Using the INPATH Operator (Cont.)

Path Search Feature	Syntax	Description
Descendant/Attribute Existence Testing	<p>virginia INPATH (A[B])</p> <p>You can search for documents using the any-level tag searching:</p> <pre>select id from library_catalog where contains (text,'Quijote INPATH(/order/title)') > 0;</pre> <p>You can also use the "*" as a single level wildcard. The * matches exactly one level.:</p> <pre>select id from library_catalog where contains (text,'Cervantes INPATH(/order/*/author)') > 0;</pre>	<p>Finds all documents where “virginia” appears in an A element which has a B element as a direct child.</p> <ul style="list-style-type: none"> ▪ <code>virginia INPATH A[./B]</code> -- Finds all documents where “virginia” appears in an A element which has a B element as a descendant (any level). ▪ <code>virginia INPATH A[@B]</code> -- Finds all documents where “virginia” appears in an A element which has a B attribute

Table 7–2 Path Searching XML Documents Using the INPATH Operator (Cont.)

Path Search Feature	Syntax	Description
Attribute Value Testing	virginia INPATH A[@B = "foo"]	<p>Finds all documents where “virginia” appears in an A element which has a B attribute whose value is “foo”.</p> <ul style="list-style-type: none"> Only equality is supported as a test. Range operators and functions are not supported. The left-hand-side of the equality MUST be an attribute or tag. Literals here are not allowed. The right-hand-side must be a literal. Tags and attributes here are not allowed.
Within Equality	<p>That means that:</p> <p>virginia INPATH (A[@B = "pot of gold"]), would, with the default lexer and stoplist, match any of the following:</p> <pre>virginia</pre> <p>By default, lexing is case-independent, so “pot” matches “POT”, virginia</p> <p>By default, “of” is a stopword, and, in a query, would match any word in that position, virginia</p>	<p>Within equality (See "Using INPATH Operator for Path Searching in XML Documents" on page 7-12) is used to evaluate the test.</p> <p>Whitespace is mainly ignored in text indexing. Again, lexing is case-independent:</p> <pre>virginia</pre> <p>Underscore is a non-alphabetic character, and is not a join character by default. As a result, it is treated more or less as whitespace and breaks up that string into three words.</p> <p>Example:</p> <pre>select id from library_catalog where contains(text,(Bob the Builder) INPATH(//item[@type="dvd"])) > 0;</pre> <p>The following will not return rows:</p> <pre>select id from library_catalog where contains(text,(Bob the Builder) INPATH(//item[@type="book"])) > 0;</pre>
Numeric Equality	virginia INPATH (A[@B = 5])	<p>Numeric literals are allowed. But they are treated as text. The within equality is used to evaluate. This means that the query does NOT match. That is, virginia does not match A[@B=5] where "5.0", a decimal is not considered the same as 5, an integer.</p>
Conjunctive Testing	<p>virginia INPATH (A[B AND C])</p> <p>virginia INPATH (A[B AND @C = "foo"])</p>	<p>Predicates can be conjunctively combined.</p>
Combining Path and Node Tests	<p>virginia INPATH (A[@B = "foo"]/C/D)</p> <p>virginia INPATH(A/B[@C]/D[E])</p>	<p>Node tests can be applied to any node in the path.</p>

Using HASPATH Operator for Path Searching in XML Documents

Use the `HASPATH` operator to find all XML documents that contain a specified section path. `HASPATH` is used when you want to test for path existence. It is also very useful for section equality testing. To find all XML documents where an order has an item within it:

```
select id from library_catalog
       where contains(text, 'HASPATH(order/item)') > 0;
```

will return all documents where the top-level tag is a `order` element which has a `item` element as a direct child.

In Oracle9i, Oracle Text introduces a new section type and new query operators which support an XPath-like query language. Indexes of type context with XML path searching are able to perform very complex section searches on XML documents. Here are more examples of path querying using `INPATH` and `HASPATH`. Assuming the following XML document:

```
<?xml version="1.0"?>
<order>
  <item type="book">
    <title>Crypto</title>
    <author>Levi</author>
  </item>
  <item type="dvd">
    <title> Bob the Builder</title>
    <author>Auerbach</author>
  </item>
  <item type="book">
    <title>Don Quijote</title>
    <author>Cervantes</author>
  </item>
</order>
```

In general, use `INPATH` and `HASPATH` operators only when your index has been created with `PATH_SECTION_GROUP`. Use of `PATH_SECTION_GROUP` enables path searching. Path searching extends the syntax of the `WITHIN` operator so that the section name operand (right-hand-side) is a *path* instead of a *section name*.

Using HASPATH Operator for Path Existence Searching

Note: The HASPATH operator functions in a similar fashion to the existsNode() in XMLType.

Only use the HASPATH operator when your index has been created with the PATH_SECTION_GROUP. The syntax for the HASPATH operator is:

- *WHERE CONTAINS(column, 'HASPATH(path)'...):* Here HASPATH searches an XML document set and returns a score of 100 for all documents where path exists. Parent and child paths are separated with the / character, for example, A/B/C. For example, the query:

```
...WHERE CONTAINS (col, 'HASPATH(A/B/C)')>0;
```

finds and returns a score of 100 for the document:

```
<A><B><C>Virginia</C></B></A>
```

without having to reference Virginia at all.

- *WHERE CONTAINS(column, 'HASPATH(A="value")'...):* Here the HASPATH clause searches an XML document set and returns a score of 100 for all documents that have element A with content value and only that value. HASPATH is used to test equality. This is the "Section Equality Testing" feature of the HASPATH operator. The query:

```
...WHERE CONTAINS virginia INPATH A
```

finds <A>virginia, but it also finds <A>virginia state. To limit the query to the term virginia and nothing else, you can use a section equality test with the HASPATH operator. For example:

```
... WHERE CONTAINS (col, 'HASPATH(A="virginia")'
```

finds and returns a score of 100 only for the first document, and not the second.

Tag Value Equality Testing

You can do tag value equality test with HASPATH:

```
select id from library_catalog
       where CONTAINS(text, 'HASPATH (//author="Auerbach")') >0;
```

Building a Query Application with Oracle Text

To build a Oracle Text query application carry out the following steps:

1. **Create a section preference group.** Before you create a section group and Oracle text index you must first determine the role you will need and grant the appropriate privilege. See ["Oracle Text Users and Roles"](#) on page 7-5, and grant the appropriate privilege.

After creating and preparing your data, you are ready to perform the next step. See ["Step 1. Create a Section Group Preference"](#) on page 7-21.

2. **Add sections or stop_sections**
3. **Create an Oracle Text index** based on the section group you created. Using the section preference created, you then create an Oracle Text index. See [Building a Query Application with Oracle Text](#).
4. **Build your query application** using the CONTAINS operator. Now you can finish building your query application. See ["Building a Query Application with Oracle Text"](#).

What Role Do You Need?

First determine the role you need. See *Oracle Text Reference* and ["Oracle Text Users and Roles"](#) on page 7-5, and grant the appropriate privilege as follows:

```
CONNECT system/manager
GRANT ctxapp to scott;
CONNECT scott/tiger
```

Step 1. Create a Section Group Preference

The first thing you must do is create a preference. This section describes how to create section preferences using `PATH_SECTION_GROUP`, `XML_SECTION_GROUP`, and `AUTO_SECTION_GROUP`. [Table 7-3](#) describes the groups and summarizes their features.

Table 7–3 Comparing Oracle Text Section Groups

Section Group	Description
XML_SECTION_GROUP	Use this group type for indexing XML documents and for defining sections in XML documents.
AUTO_SECTION_GROUP	<p>Use this group type to automatically create a zone section for each start-tag/end-tag pair in an XML document. The section names derived from XML tags are case-sensitive as in XML. Attribute sections are created automatically for XML tags that have attributes. Attribute sections are named in the form attribute@tag. Stop sections, empty tags, processing instructions, and comments are not indexed. The following limitations apply to automatic section groups:</p> <ul style="list-style-type: none">■ You cannot add zone, field or special sections to an automatic section group.■ Automatic sectioning does not index XML document types (root elements.) However, you can define stop-sections with document type.■ The length of the indexed tags including prefix and namespace cannot exceed 64 characters. Tags longer than this are not indexed.
PATH_SECTION_GROUP	<p>Use this group type to index XML documents. Behaves like the AUTO_SECTION_GROUP. With this section group you can do path searching with the INPATH and HASPATH operators. Queries are case-sensitive for tag and attribute names.</p> <p>How is PATH_SECTION_GROUP Similar to AUTO_SECTION_GROUP?</p> <p>Documents are assumed to be XML, Every tag and every attribute is indexed by default, Stop sections can be added to prevent certain tags from being indexed, Only stop sections can be added -- ZONE, FIELD, and SPECIAL sections cannot be added, When indexing XML document collections, you do not need to explicitly define sections as Oracle automatically does this for you.</p> <p>How Does PATH_SECTION_GROUP Differ From AUTO_SECTION_GROUP?</p> <p>Path Searching is allowed at query time (see "Case Study: Searching XML-Based Conference Proceedings" and "You can use INPATH operator to perform path searching in XML documents. Table 7-2 summarizes the ways you can use the INPATH operator for path searching." on page 7-12) with the new INPATH and HASPATH operators, Tag and attribute names are case-sensitive in queries.</p>

Note: If you are using the AUTO_SECTION_GROUP or PATH_SECTION_GROUP to index an XML document collection, you need not explicitly define sections since the system does this for you during indexing.

Deciding Which Section Group to Use

How do you determine which section groups is best for your application? This depends on your application. Table 7-4 lists some general guidelines to help you decide which of the XML_, AUTO_, or PATH_ section groups to use when indexing your XML documents, and why.

Table 7-4 Guidelines for Choosing XML_, AUTO_, or PATH_ Section Groups

Application Criteria	XML_section_...	AUTO_section_...	PATH_section_...
You are using XPATH search features	--	--	Yes
You know the layout and structure of your XML documents, and you can predefine the sections on which users are most likely to search.	Yes	--	--
You do not know which tags users are most likely to search.	--	Yes	--
Query performance, in general	Fastest	Little slower than XML_section_...	Little slower than AUTO_section_...
Indexing performance, in general	Fastest	Little slower than XML_section_...	Little slower than AUTO_section_...
Index size	Smallest	Little larger than XML_section_...	Little larger than AUTO_section_...
Other features	Mappings can be defined so that tags in one or different DTDs can be mapped to one section. Good for DTD evolution and data aggregation.	Simplest. No need to define mapping, add_stop_section can be used to ignore some sections.	Designed for more sophisticated XPATH-like queries

Creating a Section Preference with XML_SECTION_GROUP

The following command creates a section group called, xmlgroup, with the XML_SECTION_GROUP group type:

```
EXEC ctx_ddl.create_section_group('myxmlgroup', 'XML_SECTION_GROUP');
```

Creating a Section Preference with AUTO_SECTION_GROUP

You can set up your indexing operation to automatically create sections from XML documents using the section group AUTO_SECTION_GROUP. Here, Oracle creates

zone sections for XML tags. Attribute sections are created for those tags that have attributes, and these attribute sections are named in the form “tag@attribute.”

The following command creates a section group called autogroup with the AUTO_SECTION_GROUP group type. This section group *automatically* creates sections from tags in XML documents.

```
EXEC ctx_ddl.create_section_group('autogroup', 'AUTO_SECTION_GROUP');
```

Note: You can add attribute sections only to XML section groups. When you use AUTO_SECTION_GROUP, attribute sections are created automatically. Attribute sections created automatically are named in the form tag@attribute.

Creating a Section Preference with PATH_SECTION_GROUP

To enable path section searching, index your XML document with PATH_SECTION_GROUP. For example:

```
EXEC ctx_ddl.create_section_group('xmlpathgroup', 'PATH_SECTION_GROUP');
```

Step 2. Set the Preference's Attributes

To set the preference's attributes for XML_SECTION_GROUP, use the following procedures:

- Add_Zone_Section
- Add_Attr_Section
- Add_Field_Section
- Add_Special_Section

To set the preference's attributes for AUTO_SECTION_GROUP and PATH_SECTION_GROUP, use the following procedures:

- Add_Stop_Section

There are corresponding CTX_DDL.DROP sections and CTX_DDL.REMOVE section commands.

2.1 XML_SECTION_GROUP: Using CTX_DDL.add_zone_section

The syntax for CTX_DDL.add_zone_section follows:

```
CTX_DDL.Add_Zone_Section (
  group_name      => 'my_section_group' /* whatever you called it in the
preceding section */
  section_name    => 'author' /* what you want to call this section */
  tag             => 'my_tag' /* what represents it in XML */ );
```

where 'my_tag' implies opening with <my_tag> and closing with </my_tag>.

add_zone_section Guidelines

add_zone_section guidelines are listed here:

- Call CTX_DDL.Add_Zone_Section for each tag in your XML document that you need to search on.

2.2 XML_SECTION_GROUP: Using CTX_DDL.Add_Attr_Section

The syntax for CTX_DDL.ADD_ATTR_SECTION follows:

```
CTX_DDL.Add_Attr_Section ( /* call this as many times as you need to describe
the attribute sections */
  group_name      => 'my_section_group' /* whatever you called it in the
preceding section */
  section_name    => 'author' /* what you want to call this section */
  tag             => 'my_tag' /* what represents it in XML */ );
```

where 'my_tag' implies opening with <my_tag> and closing with </my_tag>.

Add_Attr_Section Guidelines

Add_Attr_Section guidelines are listed here:

- Consider meta_data attribute author:


```
<meta_data author = "John Smith" title="How to get to Mars">
```

ADD_ATTR_SECTION adds an attribute section to an XML section group. This procedure is useful for defining attributes in XML documents as sections. This enables searching XML *attribute* text with the WITHIN operator.

The section_name:

- Is the name used for WITHIN queries on the attribute text.

- Cannot contain the colon (:) or dot (.) characters.
- Must be unique within group_name.
- Is case-insensitive.
- Can be no more than 64 bytes.

The tag specifies the name of the attribute in tag@attr format. This is case-sensitive.

Note: In the ADD_ATTR_SECTION procedure, you can have many tags all represented by the same section name at query time. Explained in another way, the names used as the arguments of the keyword WITHIN can be different from the actual XML tag names. That is many tags can be mapped to the same name at query time. This feature enhances query usability.

2.3 XML_SECTION_GROUP: Using CTX_DDL.Add_Field_Section

The syntax for CTX_DDL.Add_Field_Section follows:

```
CTX_DDL.Add_Field_Section (
  group_name      => 'my_section_group' /* whatever you called it in the preceding
section */
  section_name    => 'qq' /* what you want to call this section */
  tag             => 'my_tag' /* what represents it in XML */ );
  visible        => TRUE or FALSE );
```

Add_Field_Section Guidelines

Add_Field_Section guidelines are listed here:

- Searches using Field_Sections are faster than those using Zone_Section.
- Visible attribute: This is available in Add_Field_Section but not available in the Add_Zone_section. If VISIBLE is set to TRUE then the text within the the Field section will be indexed as part of the enclosing document. For example:

```
<state> Virginia </state>
  CTX_DDL.Add_Field_Section (
  group_name      => 'my_section_group'
  section_name    => 'state'
  tag             => 'state'
  visible        => TRUE or FALSE );
```

If `visible` is set to `TRUE`, then searching on Virginia without specifying the `state` Field section produces a hit.

If `visible` is set to `FALSE`, then searching on Virginia without specifying the `state` Field section does not produce a hit.

How Attr_Section Differs from Field_Section

Attribute section differs from Field section in the following ways:

- *Attribute text is considered invisible*, hence the following clause:

```
WHERE CONTAINS (... , '... jeeves',... )...
```

does NOT find the document. This is similar to when Field sections have `visible` set to `FALSE`. Unlike Field sections, however, Attribute section within searches can distinguish between occurrences. Consider the document:

```
<comment author="jeeves">
  I really like Oracle Text
</comment>
<comment author="bertram">
  Me too
</comment>
```

the query:

```
WHERE CONTAINS (... , '(cryil and bertram) WITHIN author', ... )...
```

will NOT find the document, because "jeeves" and "bertram" do not occur within the SAME attribute text.

- *Attribute section names cannot overlap with zone or field section names* although you can map more than one `tag@attr` to a single section name. Attribute sections do not support default values. Given the document:

```
<!DOCTYPE foo [
  <!ELEMENT foo (bar)>
  <!ELEMENT bar (#PCDATA)>
  <!ATTLIST bar
    rev CDATA "8i">
]>
<foo>
  <bar>whatever</bar>
</foo>
```

and attribute section:

```
ctx_ddl.add_attr_section('mysg','barrev','bar@rev');
```

the query:

8i within barrev does not hit the document, although in XML semantics, the “bar” element has a default value for its “rev” attribute.

2.5 AUTO_SECTION_GROUP: Using CtX_DDL.Add_Stop_Section

```
CtX_DDL.Add_Stop_Section (  
group_name      => 'my_section_group' /* whatever you called it in the preceding  
section */  
section_name    => 'qq' /* what you want to call this section */ );
```

Step 3. Create an Index Using the Section Preference Created in Step 2

Create an index depending on which section group you used to create a preference:

Creating an Index Using XML_SECTION_GROUP

To index your XML document when you have used XML_SECTION_GROUP, you can use the following statement:

```
CREATE INDEX myindex ON docs(htmlfile) INDEXTYPE IS ctxsys.context  
parameters('section group xmlgroup');
```

See Also: ["Creating an Index Using XML_SECTION_GROUP"](#) on page 7-29.

Creating an Index Using AUTO_SECTION_GROUP

The following statement creates the index, myindex, on a column containing XML files using the AUTO_SECTION_GROUP:

```
CREATE INDEX myindex ON xmldocs(xmlfile) INDEXTYPE IS ctxsys.context PARAMETERS  
( 'section group autogroup' );
```

Creating an Index Using PATH_SECTION_GROUP

To index your XML document when you have used PATH_SECTION_GROUP, you can use the following statement:

```
CREATE INDEX myindex ON xmldocs(xmlfile) INDEXTYPE IS ctxsys.context PARAMETERS  
( 'section group xmlpathgroup' );
```

See Also: *Oracle Text Reference* for detailed notes on CTX_DDL.

Example 7-3 Creating an Index Using XML_SECTION_GROUP

```
EXEC ctx_ddl_create_section_group('myxmlgroup', 'XML_SECTION_GROUP');

/* ADDING A FIELD SECTION */
EXEC ctx_ddl.Add_Field_Section /* THIS IS KEY */
( group_name =>'my_section_group',
  section_name =>'author',/* do this for EVERY tag used after "WITHIN" */
  tag =>'author'
);

EXEC ctx_ddl.Add_Field_Section /* THIS IS KEY */
( group_name =>'my_section_group',
  section_name =>'document',/*do this for EVERY tag after "WITHIN" */
  tag =>'document'
);

...
/
/* ADDING AN ATTRIBUTE SECTION */
EXEC ctx_ddl.add_attr_section('myxmlgroup', 'booktitle', 'book@title');

/* The more sections you add to your index, the longer your search will take.*/
/* Useful for defining attributes in XML documents as sections. This allows*/
/* you to search XML attribute text using the WITHIN operator.*/
/* The section name:
/* ** Is used for WITHIN queries on the attribute text.
/* ** Cannot contain the colon (:) or dot (.) characters.
/* ** Must be unique within group_name.
/* ** Is case-insensitive.
/* ** Can be no more than 64 bytes.
/* ** The tag specifies the name of the attribute in tag@attr format. This is
/* case-sensitive. */
/* Names used as arguments of the keyword WITHIN can be different from the
/* actual XML tag names. Many tags can be mapped to the same name at query
/* time.*/
/* Call CTX_DDL.Add_Zone_Section for each tag in your XML document that you need
/* to search on. */

EXEC ctx_ddl.add_zone_section('myxmlgroup', 'mybooksec', 'mydocname(book)');

CREATE INDEX my_index ON my_table ( my_column )
  INDEXTYPE IS ctxsys.context
```

```
PARAMETERS ( 'SECTION GROUP my_section_group' );  
  
SELECT my_column FROM my_table  
WHERE CONTAINS(my_column, 'smith WITHIN author') > 0;
```

Step 4. Create Your Query Syntax

See the section, ["Querying with the CONTAINS Operator"](#) for information about how to use the CONTAINS operator in query statements.

Querying Within Attribute Sections

You can query within attribute sections when you index with either XML_SECTION_GROUP or AUTO_SECTION_GROUP as your section group type.

Assume you have an XML document as follows:

```
<book title="Tale of Two Cities">It was the best of times.</book>
```

You can define the section title@book as the attribute section title. You can do so with the CTX_DDL.Add_Attr_Section procedure or dynamically after indexing with ALTER INDEX.

Note: When you use the AUTO_SECTION_GROUP to index XML documents, the system automatically creates attribute sections and names them in the form attribute@tag.

If you use the XML_SECTION_GROUP, you can name attribute sections anything with CTX_DDL.ADD_ATTR_SECTION.

To search on Tale within the attribute section title, issue the following query:

```
WHERE CONTAINS (...,'Tale WITHIN title', ...)
```

When you define the TITLE attribute section as such and index the document set, you can query the XML attribute text as follows:

```
... WHERE CONTAINS (...,'Cities WITHIN booktitle', ....)...
```

When you define the AUTHOR attribute section as such and index the document set, you can query the XML attribute text as follows:

```
... WHERE 'England WITHIN authors'
```

Example 7-4 Querying an XML Document

This example does the following:

1. Creates and populates table `res_xml`
2. Creates an index, `section_group`, and preferences
3. Paramaterizes the preferences
4. Runs a test query against `res_xml`

```
drop table res_xml;

CREATE TABLE res_xml (
  pk          NUMBER PRIMARY KEY ,
  text       CLOB
) ;

insert into res_xml values(111,
  'ENTITY chap8 "Chapter 8, <q>Keeping it Tidy: the XML Rule Book </q>" this is
the document section');
commit;

---
--- script to create index on res_xml
---

--- cleanup, in case we have run this before
DROP INDEX res_index ;
EXEC CTX_DDL.DROP_SECTION_GROUP ( 'res_sections' ) ;

--- create a section group
BEGIN
  CTX_DDL.CREATE_SECTION_GROUP ( 'res_sections', 'XML_SECTION_GROUP' ) ;
  CTX_DDL.ADD_FIELD_SECTION ( 'res_sections', 'chap8', '<q>' ) ;
END ;
/

begin
  ctx_ddl.create_preference
  (
    preference_name => 'my_basic_lexer',
    object_name     => 'basic_lexer'
  );
  ctx_ddl.set_attribute
  (
```

```

        preference_name => 'my_basic_lexer',
        attribute_name   => 'index_text',
        attribute_value  => 'true'
    );
    ctx_ddl.set_attribute
    (
        preference_name => 'my_basic_lexer',
        attribute_name   => 'index_themes',
        attribute_value  => 'false');
end;
/

CREATE INDEX res_index
ON res_xml(text)
INDEXTYPE IS ctxsys.context
PARAMETERS ( 'lexer my_basic_lexer SECTION GROUP res_sections' ) ;

```

Test the preceding index with a test query, such as:

```
SELECT pk FROM res_xml WHERE CONTAINS( text, 'keeping WITHIN chap8' )>0 ;
```

Example 7-5 Creating an Index and Performing a Text Query

```

drop table explain_ex;

create table explain_ex
(
    id          number primary key,
    text        varchar(2000)
);

insert into explain_ex ( id, text )
values ( 1, 'thinks thinking thought go going goes gone went' || chr(10) ||
        'oracle orackle oricle dog cat bird' || chr(10) ||
        'President Clinton' );
insert into explain_ex ( id, text )
values ( 2, 'Last summer I went to New England' || chr(10) ||
        'I hiked a lot.' || chr(10) ||
        'I camped a bit.' );

commit;

```

Example 7-6 Text Query Using "ABOUT" in the Text Query Expression

```

Set Define Off
select text
from explain_ex

```

```

WHERE CONTAINS ( text,
  '( $( think & go ) , ?oracle ) & ( dog , ( cat & bird ) ) & about(mammal
                                     during Bill Clinton)' ) > 0;

select text
  from explain_ex
  WHERE CONTAINS ( text, 'about ( camping and hiking in new england )' ) > 0;

```

Example 7-7 Creating an Index Using AUTO_SECTION_GROUP

```

ctx_ddl_create_section_group('auto', 'AUTO_SECTION_GROUP');

CREATE INDEX myindex ON docs(xmlfile_column)
  INDEXTYPE IS ctxsys.context
  PARAMETERS ('filter ctxsys.null_filter SECTION GROUP auto');

SELECT xmlfile_column FROM docs
  WHERE CONTAINS (xmlfile_column, 'virginia WITHIN title')>0;

```

Example 7-8 Creating an Index Using PATH_SECTION_GROUP

```

EXEC ctx_ddl.create_section_group('xmlpathgroup', 'PATH_SECTION_GROUP');

CREATE INDEX myindex ON xmldocs(xmlfile_column)
  INDEXTYPE IS ctxsys.context
  PARAMETERS ('section group xmlpathgroup');

SELECT xmlfile_column FROM xmldocs
... WHERE CONTAINS (column, 'Tale WITHIN title@book')>0;

```

Example 7-9 Using XML_SECTION_GROUP and add_attr_section to Aid Querying

Consider an XML file that defines the BOOK tag with a TITLE attribute as follows:

```

<BOOK TITLE="Tale of Two Cities">
It was the best of times. </BOOK>
<Author="Charles Dickens">
Born in England in the town, Stratford_Upon_Avon </Author>

```

Recall the CTX_DDL.ADD_ATTR_SECTION syntax is:

```

CTX_DDL.Add_Attr_Section ( group_name, section_name, tag );

```

To define the title attribute as an attribute section, create an XML_SECTION_GROUP and define the attribute section as follows:

```

ctx_ddl_create_section_group('myxmlgroup', 'XML_SECTION_GROUP');

```

```
ctx_ddl.add_attr_section('myxmlgroup', 'booktitle', 'book@title');
ctx_ddl.add_attr_section('myxmlgroup', 'authors', 'author');
end;
```

Note:

- Oracle knows what the end tags look like from the `group_type` parameter you specify when you create the section group. The start tag you specify must be unique within a section group.
 - Section names need not be unique across tags. You can assign the same section name to more than one tag, making details transparent to searches.
-
-

Presenting the Results of Your Query

An Oracle Text query application enables viewing documents returned by a query. You typically select a document from the hit list and then your application presents the document in some form.

With Oracle Text, you can render a document in different ways. For example, with the query terms highlighted. Highlighted query terms can be either the words of a word query or the themes of an ABOUT query in English. This rendering uses the `CTX_DOC.HIGHLIGHT` or `CTX_DOC.MARKUP` procedures.

You can also obtain theme information from documents with the `CTX_DOC.THEMES` PL/SQL package. Besides these there are several other `CTX_DOC` procedures for presenting your query results.

`INPATH` does not support working with highlighting or themes.

See Also: *Oracle Text Reference* for more information on the `CTX_DOC` package.

XMLType Indexing

The Oracle9i datatype for storing XML, `XMLType`, is a core database feature.

You Need Query Rewrite Privileges

Note: These privileges are only required for Oracle9i Release 1 (9.0.1).

You can create an Oracle Text index on this type, but you need a few database privileges first:

1. The user creating the index must have Query Rewrite privileges:

```
GRANT QUERY REWRITE TO <user>
```

Without this privilege, the create index will fail with:

```
ORA-01031: insufficient privileges
```

<user> should be the user creating the index. The database schema that owns the index, if different, does not need the grant.

2. `query_rewrite_enabled` should be true, and `query_rewrite_integrity` should be trusted. You can add them to the `init.ora` file:

```
query_rewrite_enabled=true  
query_rewrite_integrity=trusted
```

or turn it on for the session as follows:

```
ALTER SESSION SET query_rewrite_enabled=true;  
ALTER SESSION SET query_rewrite_integrity=trusted;
```

Without these, queries will fail with:

```
DRG-10599: column is not indexed
```

These privileges are needed because `XMLType` is really an object, and you access it through a function, hence an Oracle Text index on an `XMLType` column is actually a function-based index on the `getclobval()` method of the type. These are the standard grants you need to use function-based indexes, however, unlike function-based B-Tree indexes, you do not need to calculate statistics.

Note: *Oracle9i SQL Reference* under *CREATE INDEX*, states:

To create a function-based index in your own schema on your own table, in addition to the prerequisites for creating a conventional index, you must have the `QUERY REWRITE` system privilege.

To create the index in another schema or on another schema's table, you must have the `GLOBAL QUERY REWRITE` privilege. In both cases, the table owner must also have the `EXECUTE` object privilege on the function(s) used in the function-based index.

In addition, in order for Oracle to use function-based indexes in queries, the `QUERY_REWRITE_ENABLED` parameter must be set to `TRUE`, and the `QUERY_REWRITE_INTEGRITY` parameter must be set to `TRUSTED`.

System Parameter is Set to the Default, `CTXSYS.PATH_SECTION_GROUP`

When an `XMLType` column is detected, and no section group is specified in the parameters string, the default system examines the new system parameter `DEFAULT_XML_SECTION`, and uses the section group specified there. At install time this system parameter is set to `CTXSYS.PATH_SECTION_GROUP`, which is the default path sectioner.

The default filter system parameter for `XMLType` is `DEFAULT_FILTER_TEXT`, which means that the `INSO` filter is not engaged by default.

XMLType Indexes Work Like Other Oracle Text Indexes

Other than the database privileges and the special default section group system parameter, indexes on `XMLType` columns work like any other Oracle Text index.

Example 7–10 *Creating a Text Index on XMLType Columns*

Here is a simple example:

```
connect ctxsys/ctxsys
GRANT QUERY REWRITE TO xtest;
connect xtest/xtest

CREATE TABLE xtest(doc sys.xmltype);
INSERT INTO xtest VALUES (sys.xmltype.createxml('<A>simple</A>'));

CREATE INDEX xtestx ON xtest(doc)
```

```

INDEXTYPE IS ctxsys.context;
ALTER SESSION SET query_rewrite_enabled = true;
ALTER SESSION SET query_rewrite_integrity = trusted;

SELECT a.doc.getclobval() FROM xtest a
       WHERE CONTAINS (doc, 'simple INPATH(A)')>0;

```

Using Oracle Text with Oracle XML DB

Creating an Oracle Text Index on a UriType Column

UriType columns can be indexed natively in Oracle9i database using Oracle Text. No special datastore is needed.

Example 7-11 *Creating an Oracle Text Index on a riType Column*

For example:

```

CREATE TABLE table uri_tab ( url sys.httpuritype);

INSERT INTO uri_tab VALUES
    (sys.httpuritype.createUri('http://www.oracle.com'));

CREATE INDEX urlx ON uri_tab(url) INDEXTYPE IS ctxsys.context;

SELECT url FROM uri_tab WHERE CONTAINS(url, 'Oracle')>0;

```

Table 7-5 lists system parameters used for default preference names for Oracle Text indexing, when the column type is URITYPE:

Table 7-5 *riType Column Default Preference Names for Oracle Text Indexing*

URITYPE Column	Default Preference Names
DATASTORE	DEFAULT_DATASTORE
FILTER	DEFAULT_FILTER_TEXT
SECTION GROUP	DEFAULT_SECTION_HTML
LEXER	DEFAULT_LEXER
STOPLIST	DEFAULT_STOPLIST
WORDLIST	DEFAULT_WORDLIST
STORAGE	DEFAULT_STORAGE

Querying XML Data: Use CONTAINS or existsNode()?

Oracle9i Release 1(9.0.1) introduced the Oracle Text `PATH_SECTION_GROUP`, `INPATH()`, and `HASPATH()` query operators. These allow you to do XPath-like text query searches on XML documents using the `CONTAINS` operator. `CONTAINS`, however, supports only a subset of XPath functionality. Also, there are important semantic differences between the `CONTAINS` operator and the `existsNode()` function.

The `existsNode`, `extract()` and `extractValue()` SQL functions (and the corresponding member functions of `XMLType`) provide full XPath support. This release of Oracle9i also introduces new extension functions to XPath to support full text searches.

Note: This release does not support theme querying for Oracle Text `CONTAINS()` and `existsNode()` searching.

[Table 7-6](#) lists and compares `CONTAINS()` and `existsNode()` features for searching `XMLType` data.

Table 7-6 Using CONTAINS() and existsNode() to Search XMLType Data

Feature	CONTAINS()	existsNode()
XPath Conformance	--	--
Predicate Support	--	--
■ String equality	Y	Y
■ Numerical equality	N	Y
■ Range Predicates	N	Y
■ XPath functions	N	Y
■ Spaces	N	Y
■ Namespaces	N	Y
■ Value case sensitivity	N	Y
■ Entity handling	N	Y
■ Parent-ancestor and sibling axes	N	Y

Table 7-6 Using CONTAINS() and existsNode() to Search XMLType Data (Cont.)

Feature	CONTAINS()	existsNode()
<ul style="list-style-type: none"> ■ Attribute searching under wildcards. For example, */@A or ../ 	Y	Y
<ul style="list-style-type: none"> ■ Uses XML schema or DTD information 	N	Y
<ul style="list-style-type: none"> ■ Empty elements may lead to false matches 	Y	N
Synchronous	--	--
<ul style="list-style-type: none"> ■ DML 	N	CTXXPath = N Other indexes = Y
<ul style="list-style-type: none"> ■ Query 	N	Y
Linguistic search capability	In INPATH() -> Y	Using ora:contains() -> Y
Index type	ctxsys.context	ctxsys.ctxpath
Query rewrites	N	Y, if XML schema-based and stored object-rationally
Functional indexes	N	Y. Can create Functional Index on existsNode() and extractValue() expressions.
Features supported if context index is already built	--	--
<ul style="list-style-type: none"> ■ About 	Y	N
<ul style="list-style-type: none"> ■ Highlighting 	Y	N
Text searching in general	Supports full text searching.	Supports limited text searching with ora:contains.
XPath searching in general	Limited XPath searching. Non-synchronous.	Full XPath searching. Synchronous.

Full-Text Search Functions in XPath Using ora:contains

Though XPath specifies a set of builtin text functions such as `substring()` and `CONTAINS()`, these are considerably less powerful than Oracle's fulltext search capabilities. New XPath extension functions are defined by Oracle to enable a richer set of text search capabilities. These extension functions are defined within the Oracle XML DB namespace: `http://xmlns.oracle.com/xdb`.

They can be used within XPath queries appearing in the context of `existsNode()`, `extract` and `extractValue` functions operating on `XMLType` instances.

Note: Like other procedures in `CTX_DDL` package, you must have `CTXAPP` privilege in order to execute the `CTX_DDL.CREATE_POLICY()` procedure.

ora:contains Features

The following lists the `ora:contains` features:

- The text search extension functions support most of text query operators such as stemming, fuzzy matching, and proximity search.
- These functions do not require a `ConText` index for their evaluation.
- The score values computed by these functions may differ from the regular index based query processing (through `Contains SQL` operator). Due to absence of document statistics, the weight for each term is fixed to 10. This means that a score for a word search is the number of occurrence multiplied by 10. If it exceeds 100, it is truncated to 100. This is also true for fuzzy matched terms.

ora:contains Syntax

The following is the syntax for the `ora:contains` function:

```
number contains(string, string, string?, string?)
```

where:

- `string`, the first argument is input text value
- `string`, the second argument is the text query string
- `string?`, the optional third argument is the policy name
- `string?`, the optional fourth argument is the policy owner

The `contains` extension function in the Oracle XML DB namespace, takes the input text value as the first argument and the text query string as the second argument. It returns the score value - a number between 0 and 100.

The optional third and fourth arguments can be used to specify the name and owner of the CTX policy which is to be used for processing the text query. If the third argument is not specified, it defaults to the CTX policy named `DEFAULT_POLICY_ORACONTAINS` owned by `CTXSYS`. If the fourth argument is not specified, the policy owner is assumed to be the current user.

ora:contains Examples

Assume the table `xmltab` contains XML documents corresponding to books with embedded chapters, each chapter containing a `title` and a `body`.

```
<book>
  <chapter>
    <title>...</title>
    <body>...</body>
  </chapter>
  <chapter>
    <title>...</title>
    <body>...</body>
  </chapter>
  ...
</book>
```

Example 7-12 Using ora:contains to Find a Text Query String

Find books containing a chapter whose body contains the specified text query string:

```
select * from xmltab x where
  existsNode(value(x), '/book/chapter[ora:contains(body,"dog OR cat")>0]',
    'xmlns:ora="http://xmlns.oracle.com/xdb"') = 1;
```

Example 7-13 Using ora:contains and extract() to Find a Text Query String

Extract chapters whose body contains the specified text query string.

```
select extract(value(x),
  '/book/chapter[ora:contains(body,"dog OR cat")>0]',
  'xmlns:ora="http://xmlns.oracle.com/xdb"')
from xmltab x;
```

See Also: [Oracle XML DB: Creating a Policy for ora:contains\(\)](#) on page 7-42.

Oracle XML DB: Creating a Policy for ora:contains()

This section includes syntax and examples for creating, updating, and dropping a policy for `ora:contains()`:

The following `CTX_DDL` procedures creates/updates/drops a policy used by `ora:contains()`:

A policy is a set of preferences used for processing `ora:contains()`.

See Also:

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

for a description of the Oracle Text preferences.

[Table 7-7](#) describes the `CTX_DDL` functions for creating, updating, and dropping policies for use in your XPath searches.

Table 7-7 CTX_DDL Syntax for Creating, Updating, and Dropping Policies

CTX_DDL Function	Description
CREATE_POLICY CTX_DDL.create_policy(policy_name in varchar2, filter in varchar2 default NULL, section_group in varchar2 default NULL, lexer in varchar2 default NULL, stoplist in varchar2 default NULL, wordlist in varchar2 default NULL);	Defines a policy. Arguments: policy_name - the name for the new policy filter - the filter preference to use (reserved for future use) section_group - the section group to use (currently only NULL_SECTION_GROUP is supported) lexer - the lexer preference to use. This should not have theme indexing turned on. stoplist - the stoplist preference to use wordlist - the wordlist preference to use
UPDATE_POLICY CTX_DDL.update_policy(policy_name in varchar2, filter in varchar2 default NULL, section_group in varchar2 default NULL, lexer in varchar2 default NULL, stoplist in varchar2 default NULL, wordlist in varchar2 default NULL);	Updates a policy by replacing specified preferences. Arguments: policy_name - the name for the policy filter - the new filter preference to use (reserved for future use) section_group - the new section group to use (currently only NULL_SECTION_GROUP is supported) lexer - the new lexer preference. This should not have theme indexing turned on. stoplist - the new stoplist preference to use wordlist - the new wordlist preference to use
DROP_POLICY CTX_DDL.drop_policy(policy_name in varchar2);	Deletes a policy. Arguments: policy_name - the name of the policy

Example 7-14 Creating a Policy for ora:contains

Create lexer preference named mylex:

```
begin
  ctx_ddl.create_preference('mylex', 'BASIC_LEXER');
  ctx_ddl.set_attribute('mylex', 'printjoins', '_-');
  ctx_ddl.set_attribute('mylex', 'index_themes', 'NO');
  ctx_ddl.set_attribute('mylex', 'index_text', 'YES');
end;
```

Create a stoplist preference named mystop.

```
begin
  ctx_ddl.create_stoplist('mystop', 'BASIC_STOPLIST');
  ctx_ddl.add_stopword('mystop', 'because');
  ctx_ddl.add_stopword('mystop', 'nonetheless');
  ctx_ddl.add_stopword('mystop', 'therefore');
end;
```

Create a wordlist preference named 'mywordlist'.

```
begin
  ctx_ddl.create_preference('mywordlist', 'BASIC_WORDLIST');
  ctx_ddl.set_attribute('mywordlist', 'FUZZY_MATCH', 'ENGLISH');
  ctx_ddl.set_attribute('mywordlist', 'FUZZY_SCORE', '0');
  ctx_ddl.set_attribute('mywordlist', 'FUZZY_NUMRESULTS', '5000');
  ctx_ddl.set_attribute('mywordlist', 'SUBSTRING_INDEX', 'TRUE');
  ctx_ddl.set
  _attribute('mywordlist', 'STEMMER', 'ENGLISH');
end;

exec ctx_ddl.create_policy('my_policy', NULL, NULL, 'mylex', 'mystop',
  'mywordlist');
```

or

```
exec ctx_ddl.create_policy(policy_name => 'my_policy',
  lexer => 'mylex',
  stoplist => 'mystop',
  wordlist => 'mywordlist');
```

Then you can issue the following `existsNode()` query with your own defined policy:

```
select * from xmltab x where
  existsNode(value(x),
    '/book/chapter[ora:contains(body,"dog OR cat", "my_policy")>0]',
    'xmlns:ora="http://xmlns.oracle.com/xdb"') = 1;
```

You can also update your policy by using the following:

```
exec ctx_ddl.update_policy(policy_name => 'my_policy',
  lexer => 'my_new_lex');
```

You can drop your policy by using:

```
exec ctx_ddl.drop_policy(policy_name => 'my_policy');
```

Querying Using Other User's Policy

You can also issue the `existsNode()` query using another user's policy, in this case, using Scott's policy:

Example 7–15 Querying Another User's Policy

```
select * from xmltab x where
  existsNode(value(x),
    '/book/chapter[ora:contains(body,"dog OR cat", "Scotts_policy","Scott")>0]',
    'xmlns:ora="http://xmlns.oracle.com/xdb"') = 1;
```

Oracle XML DB: Using CTXXPATH Indexes for existsNode()

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

The `CTXXPATH` index is a new indextype provided by Oracle Text. It is designed to serve as a primary filter for `existsNode()` processing, that is, it produces a superset of the results that would be produced by the `existNode()` function.

Why do We Need CTXXPATH When ConText Indexes Can Perform XPath Searches?

The existing `ConText` index type already has some XPath searching capabilities, but the `ConText` index type has some limitations:

- For the `ConText` index to be usable as a primary filter for `existsNode()`,
 - You must create the index using `PATH_SECTION_GROUP`.
 - You cannot create the index with `USER_LEXER` or `MULTI_LEXER` preference.
 - You must create the index with `DIRECT DATASTORE`.
 - You must create the index with `NULL FILTER`.

This limits the linguistic searching capabilities that `ConText` index type provides.

- The `ConText` index is asynchronous and does not follow the same transactional semantics as `existsNode()`.

- The `ConText` index does not handle namespaces nor user-defined entities.

With all these limitations in mind, `CTXXPATH` index type was designed specifically to serve the purpose of `existsNode()` primary filter processing. You can still create `ConText` indexes with whichever preferences you need on `XMLType` columns, and this will be used to speed up `CONTAINS` operators. At the same time, you can create a `CTXXPATH` index to speedup the processing of `existsNode()`.

CTXXPATH Index Type

`CTXXPATH` index type has the following characteristics:

- This index can only be used to speed up `existsNode()` processing. It acts as a primary filter for the `existsNode()` function. In other words, it provides a superset of the results that `existsNode()` would provide
- The index can only handle a limited set of XPath expressions. See the [Section , "Choosing the Right Plan: Using CTXXPATH Index in existsNode\(\) Processing"](#) for the list of XPath expressions not supported by the index.
- The only supported parameter is the `TABLESPACE` parameter. See "[Creating CTXXPATH Storage Preferences with CTX_DDL. Statements](#)" on page 7-47.
- DMLs are asynchronous. Users are required to issue a special DDL command to synchronize the DMLs, similar to that of `ConText` index.
- Despite the asynchronous nature of DML, it still follows transactional semantics of `existsNode()` by also returning unindexed rows as part of its result set in order to guarantee its requirement of returning a superset of the valid results.

Creating CTXXPATH Indexes

You create `CTXXPATH` indexes the same way you create `ConText` indexes. The syntax is the same as that of `ConText` index:

```
CREATE INDEX [schema.]index ON [schema.]table(XMLType column)
  INDEXTYPE IS ctxsys.CTXXPATH [PARAMETERS(paramstring)];
```

where

```
paramstring = '[storage storage_pref] [memory memsize] [populate | nopopulate]'
```

Example 7–16 Creating CTXXPATH Indexes

For example:

```
CREATE INDEX xml_idx ON xml_tab(col_xml) indextype is ctxsys.CTXXPATH;
```

or

```
CREATE INDEX xml_idx ON xml_tab(col_xml) indextype is ctxsys.CTXXPATH
    PARAMETERS('storage my_storage memory 40M');
```

The index can only be used to speed up queries using `existsNode()`:

```
SELECT xml_id FROM xml_tab x WHERE
    x.col_xml.existsnode('/book/chapter[@title="XML"]')>0;
```

See Also: [Chapter 4, "Using XMLType"](#) for more information on using `existsNode()`.

Creating CTXXPATH Storage Preferences with CTX_DDL Statements

The only preference allowed for CTXXPATH index type is the STORAGE preference. You create the storage preference the same way you would for a ConText index type.

Example 7–17 Creating Storage Preferences for CTXXPATH Indexes

For example:

```
begin
ctx_ddl.create_preference('mystore', 'BASIC_STORAGE');
ctx_ddl.set_attribute('mystore', 'I_TABLE_CLAUSE',
    'tablespace foo storage (initial 1K)');
ctx_ddl.set_attribute('mystore', 'K_TABLE_CLAUSE',
    'tablespace foo storage (initial 1K)');
ctx_ddl.set_attribute('mystore', 'R_TABLE_CLAUSE',
    'tablespace foo storage (initial 1K)');
ctx_ddl.set_attribute('mystore', 'N_TABLE_CLAUSE',
    'tablespace foo storage (initial 1K)');
ctx_ddl.set_attribute('mystore', 'I_INDEX_CLAUSE',
    'tablespace foo storage (initial 1K)');
end;
```

Performance Tuning CTXXPATH Index: Synchronizing and Optimizing the Index

To synchronize DMLs, you can use the `SYNC_INDEX` procedure provided in the `CTX_DDL` package.

Example 7–18 Optimizing the CTXXPATH Index

For example:

```
exec ctx_ddl.sync_index('xml_idx');
```

To optimize the CTXXPATH index, you can use the `OPTIMIZE_INDEX()` procedure provided in the `CTX_DDL` package. For example:

```
exec ctx_ddl.optimize_index('xml_idx', 'FAST');
```

or

```
exec ctx_ddl.optimize_index('xml_idx', 'FULL');
```

See Also:

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

Choosing the Right Plan: Using CTXXPATH Index in existsNode() Processing

It is not guaranteed that a CTXXPATH index will always be used to speed up `existsNode()` processing. The following is a list of reasons why Oracle Text index may not be used under `existsNode()`:

- The Cost Based Optimizer decides it is too expensive to use CTXXPATH index as primary filter.
- The XPath expression cannot be handled by CTXXPATH index. Here are a list of XPath constructs CTXXPATH index cannot handle:
 - XPATH functions.
 - Numerical Range operators.
 - Numerical equality.
 - Arithmetic operators.
 - Union operator “|”
 - Existence of attribute
 - Positional/Index predicate, that is, `/A/B[5]`.
 - Parent and sibling axes
 - attribute following a `*`, `//`, `...`, in other words, `'/A/*/@attr'`, `'/A//@attr'`, `'/A//..@attr'`

- '.' or '*' at the end of the path expression.
- predicate following '.' or '*'.
- String literal equality is supported with the following restrictions:
 - * The left hand side must be a path ('.' self by itself is not allowed, .= "dog").
 - * The right hand side must be a literal.

For the Optimizer to better estimate the costs and selectivities for the `existsNode()` function, you must gather statistics on CTXXPATH index by using ANALYZE command or DBMS_STATS package. You can analyze the index and compute the statistics using the ANALYZE command as follows:

```
ANALYZE INDEX myPathIndex COMPUTE STATISTICS;
```

or you can simply analyze the whole table:

```
ANALYZE TABLE XMLTAB COMPUTE STATISTICS;
```

Using Oracle Text: Advanced Techniques

The following sections describe several Oracle Text advanced techniques for fine-tuning your XML data search.

Distinguishing Tags Across DocTypes

In previous releases, the XML section group was unable to distinguish between tags in different DTD's. For instance, perhaps you have a DTD for storing contact information:

```
<!DOCTYPE contact>
<contact>
  <address>506 Blue Pool Road</address>
  <email>dudeman@radical.com</email>
</contact>
```

Appropriate sections might look like:

```
ctx_ddl.add_field_section('mysg','email','email');
ctx_ddl.add_field_section('mysg','address','address');
```

This is fine until you have a different kind of document in the same table:

```
<!DOCTYPE mail>
```

```
<mail>
  <address>dudeman@radical.com</address>
</mail>
```

Now your address section, originally intended for street addresses, starts picking up email addresses, because of tag collision.

Specifying Doctype Limiters to Distinguish Between Tags

Oracle8i release 8.1.5 and higher allow you to *specify doctype limiters* to distinguish between these tags across doctypes. Simply specify the doctype in parentheses before the tag as follows:

```
ctx_ddl.add_field_section('mysg','email','email');
ctx_ddl.add_field_section('mysg','address','(contact)address');
ctx_ddl.add_field_section('mysg','email','(mail)address');
```

Now when the XML section group sees an address tag, it will index it as the address section when the document type is `contact`, or as the email section when the document type is `mail`.

Doctype-Limited and Unlimited Tags in a Section Group

If you have both doctype-limited and unlimited tags in a section group:

```
ctx_ddl.add_field_section('mysg','sec1','(type1)tag1');
ctx_ddl.add_field_section('mysg','sec2','tag1');
```

Then the limited tag applies when in the doctype, and the unlimited tag applies in all other doctypes.

Querying is unaffected by this. The query is done on the section name, not the tag, so querying for an email address would be done like:

```
radical WITHIN email
```

which, since we have mapped two different kinds of tags to the same section name, finds documents independent of which tags are used to express the email address.

XML_SECTION_GROUP Attribute Sections

In Oracle8i Release 1(8.1.5) and higher, `XML_SECTION_GROUP` offers the ability to index and search within *attribute* values. Consider a document with the following lines:

```
<comment author="jeeves">
  I really like Oracle Text
</comment>
```

Now `XML_SECTION_GROUP` offers an attribute section. This allows the inclusion of attribute values to index. For example:

```
ctx_ddl.add_attr_section('mysg','author','comment@author');
```

The syntax is similar to other `add_section` calls. The first argument is the name of the section group, the second is the name of the section, and the third is the tag, in the form `<tag_name>@<attribute_name>`. This tells Oracle Text to index the contents of the `author` attribute of the `comment` tag as the section “author”.

Query syntax is just like for any other section:

```
WHERE CONTAINS ( ... , 'jeeves WITHIN author...', ... ) ...
```

and finds the document.

Attribute Value Sensitive Section Search

Attribute sections allow you to search the contents of attributes. They do not allow you to use attribute values to specify sections to search. For instance, given the document:

```
<comment author="jeeves">
  I really like Oracle Text
</comment>
```

You can find this document by asking:

```
jeeves within comment@author
```

which is equivalent to “find me all documents which have a comment element whose `author` attribute’s value includes the word `jeeves`”.

However, there you cannot currently request the following:

```
interMedia within comment where (@author = "jeeves")
```

in other words, “find me all documents where `interMedia` appears in a comment element whose `author` is `jeeves`”. This feature -- attribute value sensitive section searching -- is planned for future versions of the product.

Dynamic Add Section

Because the section group is defined before creating the index, Oracle8i Release 1 (8.1.5) is limited in its ability to cope with changing structured document sets; if your documents start coming with new tags, or you start getting new doctypes, you have to re-create the index to start making use of those tags.

With Oracle8i Release 2 (8.1.6) and higher you can add new sections to an existing index without rebuilding the index, using `alter index` and the new `add section` parameters string syntax:

```
add zone section <section_name> tag <tag>
add field section <section_name> tag <tag> [ visible | invisible ]
```

For instance, to add a new zone section named `tsec` using the tag `title`:

```
alter index <indexname> rebuild
parameters ('add zone section tsec tag title')
```

To add a new field section named `asec` using the tag `author`:

```
alter index <indexname> rebuild
parameters ('add field section asec tag author')
```

This field section would be invisible by default, just like when using `ADD_FIELD_SECTION`. To add it as visible field section:

```
alter index <indexname> rebuild
parameters ('add field section asec tag author visible')
```

Dynamic add section only modifies the index metadata, and does not rebuild the index in any way. This means that these sections take effect for any document indexed after the operation, and do not affect any existing documents -- if the index already has documents with these sections, they must be manually marked for re-indexing (usually with an update of the indexed column to itself).

This operation does not support addition of special sections. Those would require all documents to be re-indexed, anyway. This operation cannot be done using `rebuild online`, but it should be a fairly quick operation.

Constraints for Querying Attribute Sections

The following constraints apply to querying within attribute sections:

- Regular queries on attribute text do not hit the document unless qualified in a `within` clause. Assume you have an XML document as follows:

```
<book title="Tale of Two Cities">It was the best of times.</book>
```

A query on Tale by itself does not produce a hit on the document unless qualified with `WITHIN title@book`. This behavior is like field sections when you set the visible flag set to false.

- You cannot use attribute sections in a nested `WITHIN` query.
- Phrases ignore attribute text. For example, if the original document looked like:

```
Now is the time for all good <word type="noun"> men </word> to come to the aid.
```

Then this document would hit on the regular query `good men`, ignoring the intervening attribute text.

`WITHIN` queries can distinguish repeated attribute sections. This behavior is like zone sections but unlike field sections. For example, for the following document:

```
<book title="Tale of Two Cities">It was the best of times.</book>
<book title="Of Human Bondage">The sky broke dull and gray.</book>
```

Assume the book is a zone section and `book@author` is an attribute section. Consider the query:

```
'(Tale and Bondage) WITHIN book@author'
```

This query does not hit the document, because `tale` and `bondage` are in different occurrences of the attribute section `book@author`.

Repeated Zone Sections

Zone sections can repeat. Each occurrence is treated as a separate section. For example, if `<H1>` denotes a heading section, they can repeat in the same documents as follows:

```
<H1> The Brown Fox </H1>
<H1> The Gray Wolf </H1>
```

Assuming that these zone sections are named `Heading`.

The query:

```
WHERE CONTAINS (... , 'Brown WITHIN Heading' , ... )...
```

returns this document.

But the query:

```
WHERE CONTAINS (...,' (Brown and Gray) WITHIN Heading',...)
```

does not.

Overlapping Zone Sections

Zone sections can overlap each other. For example, if `` and `<I>` denote two different zone sections, they can overlap in document as follows:

```
plain <B> bold <I> bold and italic </B> only italic </I> plain
```

Nested Sections

Zone sections can nest, including themselves as follows:

```
<TD>
  <TABLE>
    <TD>nested cell</TD>
  </TABLE>
</TD>
```

Using the `WITHIN` operator, you can write queries to search for text in sections within sections.

Nested Section Query Example

For example, assume the `BOOK1`, `BOOK2`, and `AUTHOR` zone sections occur as follows in documents `doc1` and `doc2`:

`doc1`:

```
<book1><author>Scott Tiger</author> This is a cool book to read.</book1>
```

`doc2`:

```
<book2> <author>Scott Tiger</author> This is a great book to read.</book2>
```

Consider the nested query:

```
'Scott WITHIN author WITHIN book1'
```

This query returns only `doc1`.

Using Table CTX_OBJECTS and CTX_OBJECT_ATTRIBUTES View

The CTX_OBJECT_ATTRIBUTES view displays attributes that can be assigned to preferences of each object. It can be queried by all users.

Check out the structure of CTX_OBJECTS and CTX_OBJECT_ATTRIBUTE view, with the following DESCRIBE commands. Because we are only interested in querying XML documents in this chapter, we focus on XML_SECTION_GROUP and AUTO_SECTION_GROUP.

Describe ctx_objects

```
SELECT obj_class, obj_name FROM ctx_objects
ORDER BY obj_class, obj_name;
```

The result is:

```
...
SECTION_GROUP          AUTO_SECTION_GROUP    <<==
SECTION_GROUP          BASIC_SECTION_GROUP
SECTION_GROUP          HTML_SECTION_GROUP
SECTION_GROUP          NEWS_SECTION_GROUP
SECTION_GROUP          NULL_SECTION_GROUP
SECTION_GROUP          XML_SECTION_GROUP    <<==
...
```

Describe ctx_object_attributes

```
SELECT oat_attribute FROM ctx_object_attributes
WHERE oat_object = 'XML_SECTION_GROUP';
```

The result is:

```
OAT_ATTRIBUTE
-----
ATTR
FIELD
SPECIAL
ZONE
```

```
SELECT oat_attribute FROM ctx_object_attributes
WHERE oat_object = 'AUTO_SECTION_GROUP';
```

The result is:

```
OAT_ATTRIBUTE
-----
STOP
```

Example 7–19 Case Study: See the following section.

Case Study: Searching XML-Based Conference Proceedings

This case study uses `INPATH`, `HASPATH`, and `extract()` to search a XML-based conference proceedings.

Note: You can download this sample application from <http://otn.oracle.com/products/text>

Searching for Content and Structure in XML Documents

Documents are structured presentations of information. You can define a document as an asset that contains structure, content, and presentation. This case study describes how to search for content and structure at the same time. All features described here are available in Oracle9i Release 1 (9.0.1) and higher.

Consider an online conference proceedings search where the conference attendees can perform full text searches on the structure of the papers, for example search on title, author, abstract, and so on.

Searching XML-Based Conference Proceedings Using Oracle Text

Follow these tasks to build this conference proceedings search case study:

Task 1. Grant System Privileges. Set Initialization Parameters

You must be granted with `QUERY REWRITE` system privileges to create a Functional Index. You must also have the following initialization parameters defined to create a Functional Index:

- `QUERY_REWRITE_INTEGRITY` set to `TRUSTED`
- `QUERY_REWRITE_ENABLED` set to `TRUE`
- `COMPATIBLE` set to `8.1.0.0.0` or a greater value

Task 2. Create Table Proceedings

For example, create a table, `Proceedings` with two columns: `tk`, the paper's id, and `papers`, the content. Store the paper's content as an `XMLType`.

```
CREATE TABLE Proceedings (tk number, papers XMLTYPE);
```

Task 3. Populate Table with Data

Now populate table Proceedings with some conference papers:

```
INSERT INTO Proceedings(tk,papers) VALUES (1, XMLType.createXML(
'<?xml version="1.0"?>
<paper>
  <title>Accelerating Dynamic Web Sites using Edge Side Includes</title>
  <authors>Soo Yun and Scott Davies</authors>
  <company> Oracle Corporation </company>
  <abstract> The main focus of this presentation is on Edge Side Includes
(ESI). ESI is a simple markup language which is used to mark cacheable and
non-cacheable fragments of a web page. An "ESI aware server", such as Oracle Web
Cache and Akamai EdgeSuite, can take in ESI marked content and cache and
assemble pages closer to the users, at the edge of the network, rather than at
the application server level. This session will discuss the challenge many
dynamic websites face today, discuss what ESI is, explain how ESI can be used to
alleviate these issues. The session will also describe how to build pages with
ESI, and detail the ESI and JESI (Edge Side Includes for Java) libraries.
</abstract>
  <track> Fast Track to Oracle9i </track>
</paper>'));
```

Task 4. Create an Oracle Text Index on the XMLType Column

Create an Oracle Text index on the XMLType column, papers, using the usual CREATE INDEX statement:

```
CREATE INDEX proc_idx ON Proceedings(papers) INDEXTYPE IS ctxsys.context;
```

Task 5. Querying the Conference Proceedings with XPath and Contains()

Oracle9i Release 1 (9.0.1) introduced two new SQL functions existsNode() and extract() that operate on XMLType values as follows:

- existsNode(): given an XPath expression, checks if the XPath applied over the XML document can return any valid nodes.
- extract(): given an XPath expression, applies the XPath to the XML document and returns the fragment as an XMLType.

For example, select the authors only from the XML document:

```
SELECT p.papers.extract('/paper/authors/text()').getStringVal()
FROM Proceedings p;
```

You can use the Oracle Text `CONTAINS()` operator to search for content in a text or XML document. For example, to search for papers that contain “Dynamic” in the title you can use:

```
SELECT tk FROM Proceedings
WHERE CONTAINS(papers, 'Dynamic INPATH(paper/title)')>0
```

Using the `CONTAINS()` operator Oracle9i returns the columns selected. For an XML document it returns the entire document. To extract *fragments* of XML, you can combine the `extract()` function to manipulate the XML. For example, select the authors of papers that contain “Dynamic” in the title:

```
SELECT p.papers.extract('/paper/authors/text()').getStringVal()
FROM Proceedings p
WHERE CONTAINS(papers, 'Dynamic INPATH(paper/title)')>0
```

You can use all the functionality of an Oracle Text query for the content search. The following example selects the authors of papers containing “Dynamic” or “Edge” or “Libraries” in the title:

```
SELECT p.papers.extract('/paper/authors/text()').getStringVal()
FROM Proceedings p
WHERE CONTAINS(papers, 'Dynamic or Edge or Libraries INPATH(paper/title)')>0
```

Traditional databases allow searching of XML content or structure, but not both at the same time. Oracle provides unique features that enable querying of both XML content and structure at the same time.

[Figure 7-1](#) illustrates entering the search for “Libraries” in the structure of the Conference Proceedings documents. You can search for “Libraries” within Authors, abstract, title, company, or track. In this example, you are searching for the term “Libraries” in the abstracts only. Since it is an XML document you are searching, you can even select which fragment of the XML document you want to display. This example only displays the title of the paper.

[Figure 7-2](#) shows the search results.

For the .jsp code to build this look-up application, see ["Searching Conference Proceedings Example: jsp"](#) on page 7-60.

See Also:

- *Oracle Text Reference*
- *Oracle Text Application Developer's Guide*
- <http://otn.oracle.com/products/text>

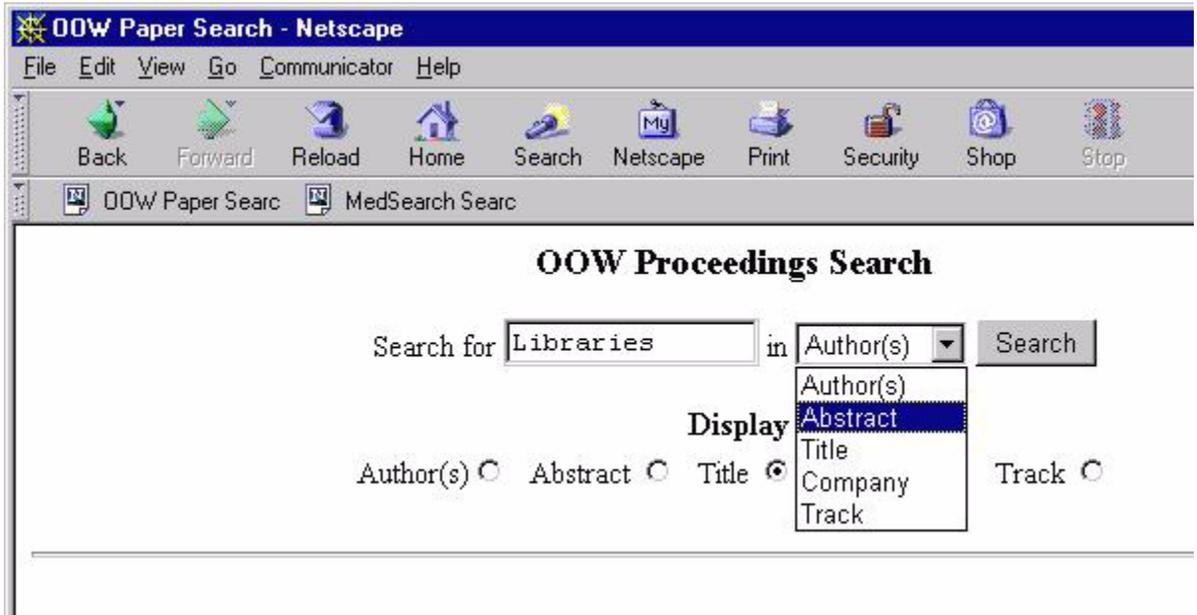
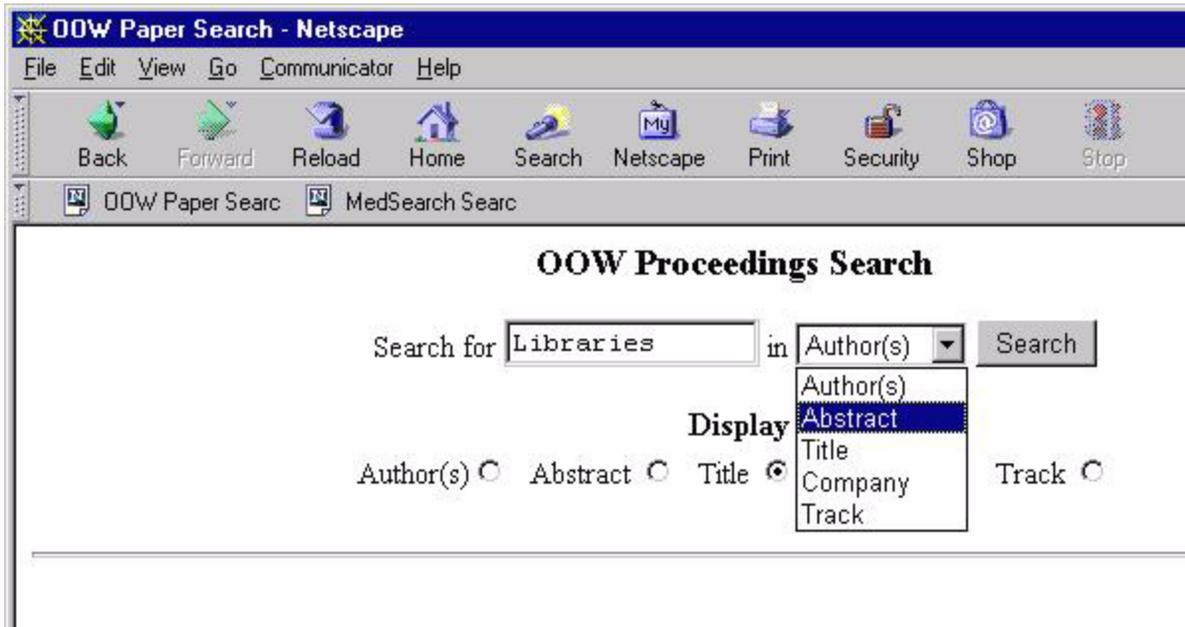
Figure 7-1 Using Oracle Text to Search for “Libraries” in the Conference Proceedings Abstracts

Figure 7-2 Oracle Text Search Results



Searching Conference Proceedings Example: jsp

Here is the full jsp example illustrating how you can use Oracle Text to create an online XML-based Conference Proceedings look-up application.

```
<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>
<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope="request" >
<jsp:setProperty name="name" property="value" param="query" />
</jsp:useBean>

<%
String connStr="jdbc:oracle:thin:@oalonso-sun:1521:betadev";
java.util.Properties info = new java.util.Properties();
Connection conn = null;
ResultSet rset = null;
Statement stmt = null;

if (name.isEmpty()) { %>
<html>
<title>OOO Paper Search</title>
```

```

<body>
<center>
  <h3>OOW Proceedings Search </h3>
  <form method=post>
  Search for
  <input type=text size=15 maxlength=25 name=query>
  in
  <select name="tagvalue">
    <option value="authors">Author(s)
    <option value="abstract">Abstract
    <option value="title">Title
    <option value="company">Company
    <option value="track">Track
  </select>
  <input type=submit value="Search">
  <p><b>Display</b><br>
  <table>
  <tr>
  <td>
    Author(s) <input type="radio" name="section" value="authors">
  </td>
  <td>
    Abstract <input type="radio" name="section" value="abstract">
  </td>
  <td>
    Title <input type="radio" name="section" value="title" checked>
  </td>
  <td>
    Company <input type="radio" name="section" value="company">
  </td>
  <td>
    Track <input type="radio" name="section" value="track">
  </td>
  </tr>
  </table>
  </form>
</center>
<hr>
</body>
</html>

<%
}
else {
%>

```

```
<html>
  <title>OOW Paper Search</title>
  <body>
    <center>
      <h3>OOW Proceedings Search </h3>
      <form method=post action="oowpapersearch.jsp">
        Search for
        <input type=text size=15 maxlength=25 name="query" value=<%=
name.getValue() %>>
        in
        <select name="tagvalue">
          <option value="authors">Author(s)
          <option value="abstract">Abstract
          <option value="title">Title
          <option value="company">Company
          <option value="track">Track
        </select>
        <input type=submit value="Search">
        <p><b>Display</b><br>
        Author(s) <input type="radio" name="section" value="authors">
        Abstract <input type="radio" name="section" value="abstract">
        Title <input type="radio" name="section" value="title" checked>
        Company <input type="radio" name="section" value="company">
        Track <input type="radio" name="section" value="track">
        </form>
      </center>
<%
  try {

    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver() );
    info.put ("user", "ctxdemo");
    info.put ("password", "ctxdemo");
    conn = DriverManager.getConnection(connStr,info);

    stmt = conn.createStatement();
    String theQuery = request.getParameter("query")+
INPATH(paper/" +request.getParameter("tagvalue")+");" );

    String tagValue = request.getParameter("tagvalue");
    String sectionValue = request.getParameter("section");

    // select p.papers.extract('/paper/authors').getStringVal()
    // from oowpapers p
```

```

// where contains(papers,'language inpath(paper/abstract)')>0

String myQuery = "select
p.papers.extract('/paper/'+request.getParameter("section")+"/text()').getStringV
al() from oowpapers p where contains(papers,'" + theQuery + "')>0";

rset = stmt.executeQuery(myQuery);
String color = "ffffff";
String myDesc = null;

int items = 0;
while (rset.next()) {
    myDesc = (String)rset.getString(1);
    items++;
    if (items == 1) {
%>

        <center>
            <table border="0">
                <tr bgcolor="#6699CC">
                    <th><%= sectionValue %></th>
                </tr>
%> } %>

                <tr bgcolor="#<%= color %>">
                    <td> <%= myDesc %></td>
                </tr>
%>

                if (color.compareTo("ffffff") == 0)
                    color = "eeeeee";
                else
                    color = "ffffff";

            }
        } catch (SQLException e) {
%>
            <b>Error: </b> <%= e %><p>
%>
        } finally {
            if (conn != null) conn.close();
            if (stmt != null) stmt.close();
            if (rset != null) rset.close();
        }
%>
</table>

```

```
        </center>
    </body></html>
    <%
  }
%>
```

Frequently Asked Questions About Oracle Text

This Frequently Asked Questions (FAQs) section is divided into the following categories:

- [FAQs: General Questions About Oracle Text](#)
- [FAQs: Searching Attribute Values with Oracle Text](#)
- [FAQs: Searching XML Documents in CLOBs Using Oracle Text](#)

FAQs: General Questions About Oracle Text

Can I Use a CONTAINS() Query with an XML Function to Extract an XML Fragment?

Answer: Yes you can. See "[Querying XML Data: Use CONTAINS or existsNode\(\)](#)" on page 7-38.

Can XML Documents Be Queried Like Table Data?

I know that an intact XML document can be stored in a CLOB in Oracle's XML solution.

Can XML documents stored in a CLOB or a BLOB be queried like table schema? For example:

```
[XML document stored in BLOB]...<name id="1111"><first>lee</first>
<second>jumee</second></name>...
```

Can value (lee, jumee) be queried by elements, attributes, and the structure of XML document?

Answer: Using Oracle Text, you can find this document with a queries such as:

```
lee within first
jumee within second
1111 within name@id
```

You can combine these like this:

```
lee within first and jumeo within second, or  
(lee within first) within name.
```

For more information, please read the Oracle Text Technical Overview available on OTN at <http://otn.oracle.com/products/text>

Can I Edit Individual XML Elements?

If some element or attribute is inserted, updated, or deleted, must the whole document be updated? Or can insert, update, and delete function as in table schema?

Answer: Oracle Text indexes CLOB and BLOB, and this has no knowledge about XML specifically, so you cannot really change individual elements. You have to edit the document as a whole.

How Are XML Files Locked in CLOBs and BLOBs?

About locking, if we manage an XML document stored in a CLOB or a BLOB, can anyone access the same XML document?

Answer: Just like any other CLOB, if someone is writing to the CLOB, they have it locked and nobody else can write to the CLOB. Other users can read it, but not write to it. This is basic LOB behavior.

An alternative is to decompose the XML document and store the information in relational fields. Then you can modify individual elements, have element-level simultaneous access, and so on. In this case, using something called the USER_DATASTORE and PL/SQL, you can reconstitute the document to XML for text indexing. Then, you can search the text as if it were XML, but manage the data as if it were relational data. Again, see the Oracle Text Technical Overview for more information at: <http://otn.oracle.com/products/text>.

How Can I Search XML Documents and Return a Zone?

I need to store a large XML file, search it, and return a specific tagged area. Using Oracle Text some of this is possible:

- I can store an XML file in a CLOB field
- I can index it with `ctxsys.context`

- I can create <Zones> and <Fields> to represent the tags in my XML file Ex.
`ctx_ddl.add_zone_section(xmlgroup,"dublincore",dc);`
- I can search for text within a zone or fieldEx. Select title from mytable where
`CONTAINS(textField,"some words WITHIN doubleness")`

How do I return a zone or a field based on a text search?

Answer: Oracle Text will only return the "hits". You can use Oracle Text doc service to highlight or mark up the section, or you can store the CLOB in an XMLType column and use the `extract()` function.

How Do I Load XML Documents into the Database?

How do I insert XML documents into a database? I need to insert the XML document as-is in column of datatype CLOB into a table.

Answer: Oracle's XML SQL Utility for Java offers a command-line utility that can be used for loading XML data. More information can be found on the XML SQL Utility at the following Web site:

<http://otn.oracle.com/tech/xml>

as well as in [Chapter 7, "XML SQL Utility \(XSU\)"](#).

You can insert the XML documents as you would any text file. There is nothing special about an XML-formatted file from a CLOB perspective.

How Do I Search XML Documents with Oracle Text?

Can Oracle Text be used to index and search XML stored in CLOBs? How can we get started?

Answer: Versions of Oracle Text before Oracle8i Release 2 (8.1.6) only allowed tag-based searching. The current version allows for XML structure and attribute based searching. There is documentation on how to have the index built and the SQL usage in the Oracle Text documentation.

See Also: *Oracle Text Reference*.

How Do I Search XML Using the WITHIN Operator?

I have this XML code:

```
<person>
  <name>efrat</name>
```

```
<childrens>
  <child>
    <id>1</id>
    <name>keren</name>
  </child>
</childrens>
</person>
```

How do I find the person who has a child name `keren` but not the person's name `keren`? This assumes I defined every tag with the `add_zone_section` that can be nested and can include themselves.

Answer: Use `'(keren within name) within child'`.

Where Can I Find Examples of Using Oracle Text to Search XML?

Answer: See the following manuals:

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

Does Oracle Text Automatically Recognize XML Tags?

Can Oracle Text recognize the tags in my XML document or do I have to use the `add_field_section` command for each tag in the XML document? My XML documents have hundreds of tags. Is there an easy way to do this?

Answer: Which version of the database are you using? I believe you need to use that command for Oracle8 release 8.1.5 but not in Oracle8i release 2 (8.1.6). You can use `AUTO_SECTION_GROUP` in Oracle8i release 2 (8.1.6).

XSQL Servlet ships with a complete (albeit simple from the Oracle Text standpoint) example of a SQL script that creates a complex XML datagram out of object types, and then creates an Oracle Text index on the XML document fragment stored in the `Insurance Claim` type.

If you download the XSQL Servlet, and look at the file `./xsql/demo/insclaim.sql` you'll be able to see the Oracle Text stuff at the bottom of the file. One of the key new features in Oracle Text in Oracle8i release 2 (8.1.6) was the `AUTO` Section for XML.

Can I Do Range Searching with Oracle Text?

I have an XML document that I have stored in CLOB. I have also created the indexes on the tags using `section_group`, and so on. One of the tags is `<SALARY></SALARY>`. I want to write a SQL statement to select all the records that have salary of greater than 5000. How do I do this? I cannot use the `WITHIN` operator. I want to interpret the value present in this tag as a number. This could be a floating point number also since this is salary.

Answer: You cannot do this in Oracle Text. Range search is not really a text operation. The best solution is to use the other Oracle XML parsing utilities to extract the salary into a `NUMBER` field. Then, you can use Oracle Text for text searching, and normal SQL operators for the more structured fields, and achieve the same results.

Can Oracle Text Do Section Extraction?

We are storing all our documents in XML format in a CLOB. Are there utilities available in Oracle, perhaps Oracle Text, to retrieve the contents a field at a time? That is, given a field name, can I get the text between tags, as opposed to retrieving the whole document and traversing the structure?

Answer: Oracle Text does not do section extraction. See the XML SQL Utility for this.

Can I Create a Text Index on Three Columns?

I have created a view based on seven to eight tables and it has columns like `custordnumber`, `product_dsc`, `qty`, `prdid`, `shipdate`, `ship_status`, and so on. I need to create an Oracle Text index on the three columns:

- `custordnumber`
- `product_dsc`
- `ship_status`

Is there a way to create a Text index on these columns?

Answer: The short answer is yes. You have two options:

1. Use the `USER_DATASTORE` object to create a concatenated field on the fly during indexing; or
2. Concatenate your fields and store them in an extra CLOB field in one of your tables. Then, create the index on the CLOB field. If you're using Oracle8i release

2 (8.1.6) or higher, then you also have the option of placing XML tags around each field prior to concatenation. This gives you the capability of searching within each field.

How Fast Is Oracle9i at Indexing Text? Can I Just Enable Boolean Searches?

We are using mySQL to do partial indexing of 9 million Web pages a day. We are running on a 4-processor Sparc 420 and are unable to do full text indexing. Can Oracle8i or Oracle9i do this?

We are not interested in transactional integrity, applying any special filters to the text pages, or in doing any other searching other than straight boolean word searches (no scoring, no stemming, no fuzzy searches, no proximity searches, and so on).

I have are the following questions:

- Will Oracle8i or Oracle9i be any faster at indexing text than mySQL?
- If so, is there a way to disable all the features of text indexing except for boolean word searches?

Answer: Yes. Oracle Text can create a full-text index on 9 million Web pages - and pretty quickly. In a benchmark on a large Sun box, we indexed 100 GB of Web pages (about 15 million) in 7 hours. We can also do partial indexing through regular DML or (in Oracle9i) through partitioning.

You can do “indexing lite” to some extent by disabling theme indexing. You do not need to filter documents if they are already in ASCII, HTML, or XML, and most common expansions, like fuzzy, stemming, and proximity, are done at query time.

FAQs: Searching Attribute Values with Oracle Text

Can I Build Text Indexes on Attribute Values?

Currently Oracle Text has the option to create indexes based on the content of a section group. But most XML elements are of the type `Element`. So, the only option for searching would be attribute values. Can I build indexes on attribute values?

Answer: Oracle8 release 8.1.6 and higher allow attribute indexing. See the following site:

http://otn.oracle.com/products/intermedia/htdocs/text_training_816/Samples/imt_816_techover.html#SCN

FAQs: Searching XML Documents in CLOBs Using Oracle Text

How Can I Search Different XML Documents Stored in CLOBs?

I store XML in CLOBs and use the DOM or SAX parsers to reparse the XML later as needed. How can I search this document repository? Oracle Text seems ideal. Do you have an example of setting this up using *interMedia* in Oracle8i, demonstrating how to define the XML_SECTION_GROUP and where to use a ZONE as opposed to a FIELD, and so on? For example:

How would I define *interMedia* parameters so that I would be able to search my CLOB column for records that had the values *aorta* and *damage* using the following XML (the DTD of which is implied)

```
WellKnownFileName.gif <keyword>echo</keyword>
<keyword>cardiogram aorta</keyword>
```

This is an image of the vessel damage.

Answer: Oracle8i release 2 (8.1.6) and higher allow searching within attribute text. That's something like: `state within book@author`. Oracle now offers attribute value sensitive search, more like the following:

```
state within book[@author = "Eric"]:

begin  ctx_ddl.create_section_group('mygrp','basic_section_group');
       ctx_ddl.add_field_section('mygrp','keyword','keyword');
       ctx_ddl.add_field_section('mygrp','caption','caption');
end;
create index myidx on mytab(mytxtcolumn) indextype is ctxsys.contextparameters
('section group mygrp');
select * from mytab where contains(mytxtcolumn, 'aorta within keyword')>0;
options:
```

- Use XML section group instead of basic section group if your tags have attributes or you need case-sensitive tag detection.
- Use zone sections instead of field sections if your sections overlap, or if you need to distinguish between instances. For instance, if *keywords* is a field section, then (*aorta* and *echo cardiogram*) within *keywords* finds the document. If it is a zone section, then it does not, because they are not in the SAME instance of *keywords*.

How Do I Store an XML Document in a CLOB Using Oracle Text?

I need to store XML files, which are currently on the file system, in the database. I want to store them as whole documents; that is, I do not want to break the document down by tags and then store the info in separate tables or fields. Rather, I want to have a universal table, that I can use to store different XML documents. I think internally it will be stored in a CLOB type of field. My XML files will always contain ASCII data.

Can this be done using Oracle Text? Should we be using Oracle Text or Oracle Text Annotator for this? I downloaded Annotator from OTN, but I could not store XML documents in the database.

I am trying to store XML documents in a CLOB column. Basically I have one table with the following definition:

```
CREATE TABLE xml_store_testing
(
  xml_doc_id NUMBER,
  xml_doc    CLOB )
```

I want to store my XML document in an xml_doc field.

I have written the following PL/SQL procedure, to read the contents of the XML document. The XML document is available on the file system and contains just ASCII data, no binary data.

```
CREATE OR REPLACE PROCEDURE FileExec
(
  p_Directory      IN VARCHAR2,
  p_FileName       IN VARCHAR2)
AS  v_CLOBLocator  CLOB;
    v_FileLocator  BFILE;
BEGIN
  SELECT  xml_doc
  INTO    v_CLOBLocator
  FROM    xml_store_testing
  WHERE   xml_doc_id = 1
  FOR    UPDATE;
  v_FileLocator := BFILENAME(p_Directory, p_FileName);
  DBMS_LOB.FILEOPEN(v_FileLocator, DBMS_LOB.FILE_READONLY);
  dbms_output.put_line(to_char(DBMS_LOB.GETLENGTH(v_FileLocator)));
  DBMS_LOB.LOADFROMFILE(v_CLOBLocator, v_FileLocator,
  DBMS_LOB.GETLENGTH(v_FileLocator));
  DBMS_LOB.FILECLOSE(v_FileLocator);
END FileExec;
```

Answer: Put the XML documents into your CLOB column, then add an Oracle Text index on it using the XML_SECTION_GROUP. See the documentation and overview material at this Web site: <http://otn.oracle.com/products/intermedia>.

Is Storing XML in CLOBs Affected by Character Set?

When I put my XML documents in a CLOB column, then add an Oracle Text index using the XML section-group, it executes successfully. But when I select from the table I see unknown characters in the table in CLOB field. Could this be because of the character set difference between operating system, where XML file resides, and database, where CLOB data resides?

Answer: Yes. If the character sets are different then you probably have to pass the data through UTL_RAW.CONVERT to do a character set conversion before writing to the CLOB.

Can I Only Insert Structured Data When the Table is Created?

I need to insert data in the database from an XML file. Currently I can only insert structured data with the table already created. Is this correct?

I am working in a law project where we need to store laws containing structured data and unstructured data, and then search the data using Oracle Text. Can I insert unstructured data too? Or do I need to develop a custom application to do it? Then, if I have the data stored with some structured parts and some unstructured parts, can I use Oracle Text to search it? If I stored the unstructured part in a CLOB, and the CLOB has tags, how can I search only data in a specific tag?

Answer: Consider using Oracle9iFS, which enables you to break up a document and store it across tables and in a LOB. Oracle Text can perform data searches with tags and is knowledgeable about the hierarchical XML structure. From Oracle8i release 2 (8.1.6), Oracle Text has had this capability, along with name/value pair attribute searches.

Can I Break an XML Document Without Creating a Custom Development?

Is document breaking possible if I don't create a custom development? Although Oracle Text does not understand hierarchical XML structure, can I do something like this?

```
<report>
  <day>yesterday</day> there was a disaster <cause>hurricane</cause>
</report>
```

Indexing with Oracle Text, I would like to search LOBs where cause was hurricane. Is this possible?

Answer: You can perform that level of searching with the current release of Oracle Text. Currently, to break a document up you have to use the XML Parser with XSLT to create a style sheet that transforms the XML into DDL. Oracle9iFS gives you a higher level interface.

Another technique is to use a JDBC program to insert the text of the document or document fragment into a CLOB or LONG column, then do the searching using the CONTAINS () operator after setting up the indexes.

What Is the Syntax for Creating a Substring Index with XML_SECTION_GROUP?

I have a CLOB column that has an existing XML_SECTION_GROUP index on certain tags within the XML content of the CLOB, as follows:

```
begin
  ctx_ddl.create_section_group('XMLDOC','XML_SECTION_GROUP');
end;
/
begin
  ctx_ddl.add_zone_section('XMLDOC','title','title');
  ctx_ddl.add_zone_section('XMLDOC','keywords','keywords');
  ctx_ddl.add_zone_section('XMLDOC','author','author');
end;
/
create index xmldoc_idx on xml_documents(xmldoc)
  indextype is ctxsys.context
  parameters ('section group xmldoc');
```

I need to search on the 'author' zone section by the first letter only. I believe I should use a substring index but I am unsure of the syntax to create a substring index. Especially when I have already declared a SECTION_GROUP preference on this column and I would also need to create a WORDLIST preference.

Answer. The main problem here is that you cannot apply that fancy substring processing just to the author section. It will apply to everything, which will probably blow up the index size. Anything you do will require reindexing the documents, so you cannot really get around having to rebuild the index entirely. Here are various ways to solve your problem:

1. Do nothing. Query just like: Z% WITHIN AUTHOR

Pro: You do not have to rebuild the index.

Con: The query is slow. Some queries cannot be executed due to wildcard maxterms limits.

2. Create a wordlist preference with PREFIX_INDEX set to TRUE, PREFIX_MIN_LENGTH set to 1, and PREFIX_MAX_LENGTH set to 1. The query looks like:
Z% WITHIN AUTHOR

Pro: This is a moderately fast query.

Con: You must use Oracle8i Release 3 (8.1.7) or higher or you will get 'junk' from words from other sections.

3. As in the preceding, plus make AUTHOR, KEYWORDS, TITLE field sections.

Pro: This faster query than 2.

Con: The field sections are less flexible with regards to. nesting and repeating occurrences.

4. Use a user_datastore or procedure_filter to transform the data so that:

```
<AUTHOR>Steven King</AUTHOR>
```

becomes

```
<AUTHORINIT>AIK</AUTHORINIT><AUTHOR>Steven King<AUTHOR>
```

Use field section for AUTHORINIT and query becomes:

```
AIK within AUTHORINIT
```

I used AIK instead of just K so that you do not have to make I and A non-stopwords.

Pro: This is the fastest query and the smallest index.

Con: It involves the most work as you have to massage the data so it slows down indexing.

Why Does the XML Search for Topic X with Relevance Y Give Wrong Results?

We are using Sun SPARC Solaris 5.8, Oracle8i Enterprise Edition Release 3 (8.1.7.2.0), Oracle Text. We are indexing XML documents that contain attributes within the XML tags. One of the sections in the XML is a list of subjects associated with the document. Each subject has a relevance associated with it. We want to

search for topic x with relevance y but we get the wrong results. For example: The data in some of the rows look like this, considering subject PA:

```
DOC 1 --> Story_seq_num = 561106
<ne-metadata.subjectlist>
  <ne-subject code="PA" source="NEWZ" relevance="50" confidence="100"/>
  <ne-subject code="CONW" source="NEWZ" relevance="100" confidence="100"/>
  <ne-subject code="LENF" source="NEWZ" relevance="100" confidence="100"/>
  <ne-subject code="TRAN" source="NEWZ" relevance="100" confidence="100"/>
</ne-metadata.subjectlist>
DOC 2 --> Story_seq_num =561107
<ne-metadata.subjectlist>
  <ne-subject code="CONW" source="NEWZ" relevance="100" confidence="100"/>
...
```

If users wants subject PA with relevance = 100, only DOC 2 should be returned. Here is a test case showing the results:

Are these the expected results?

TABLE

```
drop table t_stories1 ;
create table t_stories1 as select * from t_Stories_bck
where story_Seq_num in (561114,562571,562572,561106,561107);
```

INDEX SECTIONS

```
BEGIN
-- Drop the preference if it already exists
CTX_DDL.DROP_SECTION_GROUP('sg_nitf_story_body2');
END;
/
BEGIN
--Define a section group
ctx_ddl.create_section_group ('sg_nitf_story_body2','xml_section_group');
-- Create field sections for headline and body
ctx_ddl.add_field_section('sg_nitf_story_body2','HL','headline',true);
ctx_ddl.add_field_section('sg_nitf_story_body2','ST','body.content', true);
--Define attribute sections for the source fields
ctx_ddl.add_attr_section('sg_nitf_story_body2','P','ne-provider@id');
ctx_ddl.add_attr_section('sg_nitf_story_body2','C','ne-publication@id');
ctx_ddl.add_attr_section('sg_nitf_story_body2','S','ne-publication@section');
ctx_ddl.add_attr_section('sg_nitf_story_body2','D','date.issue@norm');
ctx_ddl.add_attr_section('sg_nitf_story_body2','SJ','ne-subject@code');
ctx_ddl.add_attr_section('sg_nitf_story_body2','SJR','ne-subject@relevance');
ctx_ddl.add_attr_section('sg_nitf_story_body2','CO','ne-company@code');
```

```
ctx_ddl.add_attr_section( 'sg_nitf_story_body2', 'TO', 'ne-topic@code');
ctx_ddl.add_attr_section( 'sg_nitf_story_body2', 'TK', 'ne-orgid@value');
```

```
END;
/
```

creating the index

```
drop index ix_stories ;
CREATE INDEX ix_stories on T_STORIES1(STORY_BODY)
  INDEXTYPE IS CTXSYS.CONTEXT
  PARAMETERS ('SECTION GROUP sg_nitf_story_body2 STORAGE ixst_story_body ');
```

-- testing the index

--We are looking for the subject PA with relevance = 100

--query that gives us the correct results

```
SELECT STORY_SEQ_NUM, STORY_BODY FROM T_STORIES1 WHERE CONTAINS(STORY_BODY, 'PA
WITHIN SJ')>0;
```

--Query that gives us the wrong results

```
SELECT STORY_SEQ_NUM, STORY_BODY FROM T_STORIES1 WHERE CONTAINS(STORY_BODY, 'PA
WITHIN SJ AND 100 within SJR')>0;
```

The data in some of the rows look like this:

```
Story_seq_num = 561106
<ne-metadata.subjectlist>
  <ne-subject code="PA" source="NEWZ" relevance="50" confidence="100"/>
  <ne-subject code="CONW" source="NEWZ" relevance="100" confidence="100"/>
  <ne-subject code="LENF" source="NEWZ" relevance="100" confidence="100"/>
  <ne-subject code="TRAN" source="NEWZ" relevance="100"
confidence="100"/>
</ne-metadata.subjectlist>

Story_seq_num =561107
<ne-metadata.subjectlist>
  <ne-subject code="CONW" source="NEWZ" relevance="100" confidence="100"/>
...
```

We are looking for the subject PA with relevance = 100

Only Story_seq_num = 561107 should be returned

The results are wrong because we wanted the subjects PA that have relevance =100. We get back story_seq_num=561106 that has relevance = 50 <ne-subject code="PA" source="NEWZ" relevance="50" confidence="100"/>

```
SQL> connect sosa/sosa
Connected.
SQL> select object_name, object_type from user_objects;
```

```
OBJECT_NAME
```

```
-----
OBJECT_TYPE
```

```
-----
IX_STORIES
INDEX
SYS_LOB0000025364C00005$$
LOB
SYS_LOB0000025364C00009$$
LOB
```

```
OBJECT_NAME
```

```
-----
OBJECT_TYPE
```

```
-----
SYS_LOB0000025364C00014$$
LOB
SYS_LOB0000025364C00016$$
LOB
T_STORIES1
TABLE
```

6 rows selected.

```
SQL> drop index ix_stories force;
```

Index dropped....

Answer. Oracle8i Release 3(8.1.7) is not able to this kind of search. You need the PATH section group in Oracle9i Release 1 (9.0.1), which has a much more sophisticated understanding of such relationships. To do this in 8.1.7 you would have to re-format the documents (possibly through a procedure filter or user datastore), use zone sections, and nested within, so that:

```
<A B="C" D="E">...
```

became

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

and queries are like:

(C within B and E within D) within A in 9.0.1, you should be able to use `PATH_SECTION_GROUP` on the unmodified data, with a query like:

```
haspath(/ne-subject[@code = "PA" and @relevance = "100"])
```

Part III

Using XMLType APIs to Manipulate XML Data

Part III of this manual introduces you to ways you can use Oracle XML DB XMLType PL/SQL and Java APIs to access and manipulate XML data. Part III contains the following chapters:

- [Chapter 8, "PL/SQL API for XMLType"](#)
- [Chapter 9, "Java and Java Bean APIs for XMLType"](#)

PL/SQL API for XMLType

This chapter describes the use of the APIs for XMLType in PL/SQL. It contains the following sections:

- [Introducing PL/SQL APIs for XMLType](#)
- [PL/SQL DOM API for XMLType \(DBMS_XMLDOM\)](#)
- [PL/SQL Parser API for XMLType \(DBMS_XMLPARSER\)](#)
- [PL/SQL XSLT Processor for XMLType \(DBMS_XSLPROCESSOR\)](#)

Introducing PL/SQL APIs for XMLType

This chapter describes the PL/SQL Application Program Interfaces (APIs) for XMLType. These include the following:

- **PL/SQL DOM API for XMLType (package DBMS_XMLDOM)**: For accessing XMLType objects. You can access both XML schema-based and non-schema-based documents. Before database startup, you must specify the read-from and write-to directories in the initialization.ORA file for example:

```
UTL_FILE_DIR=/mypath/insidemypath
```

The read-from and write-to files must be on the server file system.

- **PL/SQL XML Parser API for XMLType (package DBMS_XMLPARSER)**: For accessing the contents and structure of XML documents.
- **PL/SQL XSLT Processor for XMLType (package DBMS_XSLPROCESSOR)**: For transforming XML documents to other formats using XSLT.

Backward Compatibility with XDK for PL/SQL, Oracle9i Release 1 (9.0.1)

This release maintains support for the XDK for PL/SQL:

- XML Parser for PL/SQL
- XSLT Processor for PL/SQL

to ensure backward compatibility. Therefore, most applications written for Oracle9i Release 1 (9.0.1) XML Parser for PL/SQL and XSLT Processor for PL/SQL instances will need no changes. In this release, new applications built with the updated PL/SQL DOM and the extensions to XMLType API do not need the XDK's XML Parser for PL/SQL and XSLT Processor for PL/SQL.

See Also: *Oracle9i XML Developer's Kits Guide - XDK*

If Your Application Uses Character-Set Conversions and File Systems

Applications that extensively use character-set conversions and file system interaction require some changes. The changes needed are due to the UTL_FILE package limitations, such as read/write to files in the UTL_FILE_DIR specified at database start-up.

Note: In this release, the PL/SQL packages `DBMS_XMLDOM`, `DBMS_XMLPARSER`, and `DBMS_XSLPROCESSOR`, replace the previous XDK packages `XMLDOM`, `XMLPARSER`, and `XSLPROCESSOR`.

Differences Between PL/SQL API for XMLType and XDK for PL/SQL

This section explains differences between PL/SQL APIs native to Oracle XML DB and PL/SQL APIs available in XML Developer's Kits (XDK).

- *PL/SQL APIs for XMLType.* Use PL/SQL APIs for XMLType for developing applications that run on the server. PL/SQL APIs for XMLType in Oracle XML DB provide native XML support within the database.
- *Oracle XML XDK for PL/SQL.* Use Oracle XDK for PL/SQL for middle-tier and client-side XML support.

Note: Oracle XML DB APIs are natively integrated in Oracle9i Release 2 (9.2) and not available separately. Oracle XML DB APIs cannot be downloaded from Oracle Technology Network (OTN).

However, Oracle XDKs are available separately for download from OTN: <http://otn.oracle.com/tech/xml/content.html>.

See Also: ["PL/SQL DOM API for XMLType \(DBMS_XMLDOM\)"](#) on page 8-5

PL/SQL APIs For XMLType Features

The PL/SQL APIs for XMLType allow you to perform the following tasks:

- Create XMLType tables, columns, and views
- Access XMLType data
- Manipulate XMLType data

See Also:

- ["Key Features of Oracle XML DB"](#) on page 8-2 in [Chapter 1, "Introducing Oracle XML DB"](#), for an overview of the Oracle XML DB architecture and new features.
- [Chapter 4, "Using XMLType"](#)
- *Oracle9i XML API Reference - XDK and XDB*

Lazy XML Loading (Lazy Manifestation)

Because XMLType provides an in-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 as 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 Datatype Now Supports XML Schema

The XMLType datatype has been enhanced in this release to include support for XML schemas. You can create an XML schema and annotate it with XML to object-relational mappings. 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 XML schema you defined and registered with Oracle XML DB.

With PL/SQL APIs for XMLType You Can Modify and Store XML Elements

While typical XML parsers give read access to XML data in a standard way, they do not provide a way to modify and store individual XML elements.

What are Elements? An element is the basic logical unit of an XML document and acts as a container for other elements such as children, data, attributes, and their values. Elements are identified by start-tags, as in `<name>`, and end-tags, as in `</name>`, or in the case of empty elements, `<name/>`.

What is a DOM Parser? An embedded DOM parser accepts an XML-formatted document and constructs an in-memory DOM tree based on the document's structure. It then checks whether or not the document is well-formed and optionally whether it complies with a specific Document Type Definition (DTD). A DOM parser also provides methods for traversing the DOM tree and return data from it.

If you use the PL/SQL DOM API, you can use the `NamedNodeMap` methods to retrieve elements from an XML file.

Server-Side Support PL/SQL APIs for `XMLType` support processing on the server side only. Support for client-side processing is not provided in this release.

PL/SQL DOM API for XMLType (DBMS_XMLDOM)

Introducing W3C Document Object Model (DOM) Recommendation

Skip this section if you are already familiar with the generic DOM specifications recommended by the World Wide Web Consortium (W3C).

The Document Object Model (DOM) recommended by the W3C is a universal API to the structure of XML documents. It was originally developed to formalize Dynamic HTML, which allows animation, interaction and dynamic updating of Web pages. DOM provides a language and platform-neutral object model for Web pages and XML document structures in general. The DOM describes language and platform-independent interfaces to access and to 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. Additionally, the DOM makes it possible to write applications that work properly on all browsers and servers and on all platforms.

A brief summary of the state of the DOM Recommendations is provided in this section for your convenience.

W3C DOM Extensions Not Supported in This Release

The only extensions to the W3C DOM API not supported in this release are those relating to client-side file system input and output, and character set conversions. This type of procedural processing is available through the SAX interface.

Supported W3C DOM Recommendations

All Oracle XML DB APIs for accessing and manipulating XML comply with standard XML processing requirements as approved by the W3C. PL/SQL DOM supports Levels 1 and 2 from the W3C DOM specifications.

- In Oracle9i Release 1 (9.0.1), the XDK for PL/SQL implemented DOM Level 1.0 and parts of DOM Level 2.0.

- In Oracle9i Release 2 (9.2), the PL/SQL API for XMLType implements DOM Levels 1.0 and Level 2.0 Core, and is fully integrated in Oracle9i database through extensions to the XMLType API.

The following briefly describe each level:

- **DOM Level 1.0.** The first formal Level of the DOM specifications, completed in October 1998. Level 1.0 defines support for XML 1.0 and HTML.
- **DOM Level 2.0.** Completed in November 2000, Level 2.0 extends Level 1.0 with support for XML 1.0 with namespaces and adds support for Cascading Style Sheets (CSS) and events (user-interface events and tree manipulation events), and enhances tree manipulations (tree ranges and traversal mechanisms).
- **DOM Level 3.0.** Currently under development, Level 3.0 will extend Level 2.0 by finishing support for XML 1.0 with namespaces (alignment with the XML Infoset and support for XML Base) and will extend the user interface events (keyboard). It will also add support for abstract schemas (for DTDs and XML schema), and the ability to load and save a document or an abstract schema. It is exploring further mixed markup vocabularies and the implications on the DOM API (*Embedded DOM*), and it will support XPath.

Difference Between DOM and SAX

The generic APIs for XML can be classified in two main categories:

- *Tree-based.* The DOM is the primary generic tree-based API for XML.
- *Event-based.* SAX (Simple API for XML) is the primary generic event-based programming interface between an XML parser and an XML application.

The DOM works by creating objects. These objects have child objects and properties, and the child objects have child objects and properties, and so on. Objects are referenced either by moving down the object hierarchy or by explicitly giving an HTML element an ID attribute. For example:

```

```

Examples of structural manipulations are:

- Reordering elements
- Adding or deleting elements
- Adding or deleting attributes
- Renaming elements

PL/SQL DOM API for XMLType (DBMS_XMLDOM): Features

The default behavior for the PL/SQL DOM API for XMLType (DBMS_XMLDOM) is as follows:

- Produces a parse tree that can be accessed by DOM APIs.
- The parser is validating if a DTD is found; otherwise, it is non-validating.
- An application error is raised if parsing fails.
- The types and methods described in this document are made available by the PL/SQL package DBMS_XMLPARSER.

DTD validation follows the same rules that are exposed for the XML Parser available through the XDK in Oracle9i Release 1(9.0.1) with the only difference being that the validation occurs when the object document is manifested. For example, if lazy manifestation is used, the document will be validated when it is used.

Oracle XML DB extends the Oracle XML development platform beyond SQL support for XML text and storage and retrieval of XML data. In this release, you can operate on XMLType instances using the DOM in PL/SQL and Java. Thus, you can directly manipulate individual XML elements and data using the language best suited for your application or plug-in.

This release has updated the PL/SQL DOM API to exploit a C-based representation of XML in the server and to operate on XML schema-based XML instances. Oracle XML DB PL/SQL DOM API for XMLType and Java DOM API for XMLType comply with the W3C DOM Recommendations to define and implement structured storage of XML in relational or object-relational columns and as in-memory instances of XMLType. See "[Using PL/SQL DOM API for XMLType: Preparing XML Data](#)" on page 8-9, for a description of W3C DOM Recommendations.

XML Schema Support

PL/SQL DOM API for XMLType introduces XML schema support. Oracle XML DB uses annotations within an XML schema as metadata to determine both an XML document's structure and its mapping to a database schema.

Note: For backward compatibility and for flexibility, the PL/SQL DOM supports both XML schema-based documents and non-schema-based documents.

When an XML schema is registered with Oracle XML DB, the PL/SQL DOM API for XMLType builds an in-memory tree representation of the XML document 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. Some node types can have child nodes of various types, while some node types can only be leaf nodes and cannot have children nodes under them in the document structure.

Enhanced Performance

Additionally, Oracle XML DB uses the DOM to provide a standard way to translate data from multiple back-end data sources into XML and vice versa. 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 one native XML database to cache XML documents. Thus, Oracle XML DB can speed up application performance by acting as an intermediate cache between your Web applications and your back-end data sources, whether in relational databases or in disparate file systems.

See Also: [Chapter 9, "Java and Java Bean APIs for XMLType"](#)

Designing End-to-End Applications Using XDK and Oracle XML DB

When you build applications based on Oracle XML DB, you do not need the additional components in the XDKs. However, you can mix and match XDK components with Oracle XML DB to deploy a full suite of XML-enabled applications that run end-to-end. For example, you can use features in XDK for:

- Simple API for XML (SAX) interface processing. SAX is an XML standard interface provided by XML parsers and used by procedural and event-based applications.
- DOM interface processing for structural and recursive object-based processing.

Oracle XDKs contain the basic building blocks for creating applications that run on the client, in a browser or plug-in, for example, for reading, manipulating, transforming and viewing XML documents. To provide a broad variety of deployment options, Oracle XDKs are also available for Java, Java beans, C, C++, and PL/SQL. Unlike many shareware and trial XML components, Oracle XDKs are fully supported and come with a commercial redistribution license.

Oracle XDK for Java consists of these components:

- XML Parsers: Supports Java, C, C++ and PL/SQL, the components create and parse XML using industry standard DOM and SAX interfaces.

- XSLT Processor: Transforms or renders XML into other text-based formats such as HTML.
- XML Schema Processor: Supports Java, C, and C++, allows use of XML simple and complex datatypes.
- XML Class Generator: Automatically generates Java and C++ classes from DTDs and Schemas to send XML data from Web forms or applications.
- XML Transviewer Java Beans: Displays and transforms XML documents and data using Java components.
- XML SQL Utility: Supports Java, generates XML documents, DTDs and Schemas from SQL queries.
- TransXUtility. Loads data encapsulated in XML into the database with additional functionality useful for installations.
- XSQL Servlet: Combines XML, SQL, and XSLT in the server to deliver dynamic web content.

See Also: *Oracle9i XML Developer's Kits Guide - XDK*

Using PL/SQL DOM API for XMLType: Preparing XML Data

To take advantage of the Oracle XML DB DOM APIs, you must follow a few processes to allow Oracle XML DB to develop a data model from your XML data. This is true for any language, although PL/SQL is the focus of this chapter. The process you use depends on the state of your data and your application requirements.

To prepare data for using PL/SQL DOM APIs in Oracle XML DB, you must:

1. Create a standard XML schema if you do not already use one. Annotate the XML schema with definitions for the SQL objects defined in your relational or object-relational database.
2. Register your XML schema to generate the necessary database mappings.

You can then:

- Use XMLType views to wrap existing relational or object-relational data in XML formats. This enables an XML structure to be created that can be accessed by your application. See also "[Wrapping Existing Data into XML with XMLType Views](#)" on page 8-11.
- Insert XML documents (and fragments) into XMLType columns.

- Use Oracle XML DB DOM PL/SQL and Java APIs to access and manipulate XML data stored in XMLType columns and tables.

Creating and registering a standard XML schema allows your compliant XML documents to be inserted into the database where they can be decomposed, parsed, and stored in object-relational columns that can be accessed by your application.

Generating an XML Schema Mapping to SQL Object Types

An XML schema must be registered before it can be used or referenced in any context. When you register an XML schema, elements and attributes declared within it get mapped to separate attributes within the corresponding SQL object types within the database schema.

After the registration process is completed, XML documents conforming to this XML schema, and referencing it with its URL within the document, can be handled by Oracle XML DB. Tables and columns for storing the conforming documents can be created for root XML elements defined by this schema.

See Also: [Chapter 5, "Structured Mapping of XMLType"](#) for more information and examples.

An XML schema is registered by using the DBMS_XMLSCHEMA package and by specifying the schema document and its URL (also known as *schema location*). The URL used here is a name that uniquely identifies the registered schema within the database and need not be the physical URL where the schema document is located.

Additionally, the target namespace of the schema is another URL (different from the schema location URL) that specifies an *abstract* namespace within which the elements and types get declared. An instance of an XML document should specify both the namespace of the root element and the location (URL) of the schema that defines this element.

When instances of documents are inserted into Oracle XML DB using path-based protocols like HTTP or FTP, the XML schema to which the document conforms is registered implicitly, if its name and location are specified and if it has not been previously registered.

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *Oracle9i XML API Reference - XDK and XDB*

DOM Fidelity for XML Schema Mapping

While elements and attributes declared within the XML schema get mapped to separate attributes within the corresponding SQL object type, some encoded information in an XML document is not represented directly. In order to guarantee that the returned XML document is identical to the original document for purposes of DOM traversals (referred to as *DOM fidelity*), a binary attribute called `SYS_XDBPD$` is added to all generated SQL object types. This attribute stores all pieces of information that cannot be stored in any of the other attributes, thereby ensuring DOM fidelity for XML documents stored in Oracle XML DB.

Data handled by `SYS_XDBPD$` that is not represented in the XML schema mapping include:

- Comments
- Namespace declaration
- Prefix information

Note: In this document, the `SYS_XDBPD$` attribute has been omitted in many examples for simplicity. However, the attribute is always present in all SQL object types generated by the schema-registration process.

Wrapping Existing Data into XML with XMLType Views

To make existing relational and object-relational data available to your XML applications, you create `XMLType` views, which provide a mechanism for wrapping the existing data into XML formats. This exposes elements and entities, that can then be accessed using the PL/SQL DOM APIs.

You register an XML schema containing annotations that represent the bi-directional mapping from XML to SQL object types. Oracle XML DB can then create an `XMLType` view conforming to this XML schema.

See Also: [Chapter 11, "XMLType Views"](#)

PL/SQL DOM API for XMLType (DBMS_XMLDOM) Methods

Table 8–1 lists the PL/SQL DOM API for XMLType (DBMS_XMLDOM) methods.

Table 8–1 Summary DBMS_XMLDOM Methods

Group/Method	Description
Node methods	--
isNull()	Tests if the node is NULL .
makeAttr()	Casts the node to an Attribute.
makeCDataSection()	Casts the node to a CDataSection.
makeCharacterData()	Casts the node to CharacterData.
makeComment()	Casts the node to a Comment.
makeDocumentFragment()	Casts the node to a DocumentFragment.
makeDocumentType()	Casts the node to a Document Type.
makeElement()	Casts the node to an Element.
makeEntity()	Casts the node to an Entity.
makeEntityReference()	Casts the node to an EntityReference.
makeNotation()	Casts the node to a Notation.
makeProcessingInstruction()	Casts the node to a DOMProcessingInstruction.
makeText()	Casts the node to a DOMText.
makeDocument()	Casts the node to a DOMDocument.
writeToFile()	Writes the contents of the node to a file.
writeToBuffer()	Writes the contents of the node to a buffer.
writeToClob()	Writes the contents of the node to a clob.
getNodeName()	Retrieves the Name of the Node.
getNodeValue()	Retrieves the Value of the Node.
setNodeValue()	Sets the Value of the Node.
getNodeTypeInfo()	Retrieves the Type of the node.
getParentNode()	Retrieves the parent of the node.
getChildNodes()	Retrieves the children of the node.
getFirstChild()	Retrieves the first child of the node.

Table 8–1 Summary DBMS_XMLDOM Methods (Cont.)

Group/Method	Description
getLastChild()	Retrieves the last child of the node.
getPreviousSibling()	Retrieves the previous sibling of the node.
getNextSibling()	Retrieves the next sibling of the node.
getAttributes()	Retrieves the attributes of the node.
getOwnerDocument()	Retrieves the owner document of the node.
insertBefore()	Inserts a child before the reference child.
replaceChild()	Replaces the old child with a new child.
removeChild()	Removes a specified child from a node.
appendChild()	Appends a new child to the node.
hasChildNodes()	Tests if the node has child nodes.
cloneNode()	Clones the node.
Named node map methods	--
isNull()	Tests if the NodeMap is NULL .
getNamedItem()	Retrieves the item specified by the name.
setNamedItem()	Sets the item in the map specified by the name.
removeNamedItem()	Removes the item specified by name.
item()	Retrieves the item given the index in the map.
getLength()	Retrieves the number of items in the map.
Node list methods	--
isNull()	Tests if the Nodelist is NULL .
item()	Retrieves the item given the index in the nodelist.
getLength()	Retrieves the number of items in the list.
Attr methods	--
isNull()	Tests if the Attribute Node is NULL .
makeNode()	Casts the Attribute to a node.
getQualifiedName()	Retrieves the Qualified Name of the attribute.
getNamespace()	Retrieves the NS URI of the attribute.

Table 8–1 Summary DBMS_XMLDOM Methods (Cont.)

Group/Method	Description
getLocalName()	Retrieves the local name of the attribute.
getExpandedName()	Retrieves the expanded name of the attribute.
getName()	Retrieves the name of the attribute.
getSpecified()	Tests if attribute was specified in the owning element.
getValue()	Retrieves the value of the attribute.
setValue()	Sets the value of the attribute.
C data section methods	--
isNull()isNull()	Tests if the CDataSection is NULL .
makeNode()makeNode()	Casts the CDataSection to a node.
Character data methods	--
isNull()	Tests if the CharacterData is NULL .
makeNode()	Casts the CharacterData to a node.
getData()	Retrieves the data of the node.
setData()	Sets the data to the node.
getLength()	Retrieves the length of the data.
substringData()	Retrieves the substring of the data.
appendData()	Appends the given data to the node data.
insertData()	Inserts the data in the node at the given offSets.
deleteData()	Deletes the data from the given offSets.
replaceData()	Replaces the data from the given offSets.
Comment methods	--
isNull()	Tests if the comment is NULL .
makeNode()	Casts the Comment to a node.
DOM implementation methods	--
isNull()	Tests if the DOMImplementation node is NULL .
hasFeature()	Tests if the DOM implements a given feature.
Document fragment methods	--

Table 8–1 Summary DBMS_XMLDOM Methods (Cont.)

Group/Method	Description
isNull()	Tests if the DocumentFragment is NULL .
makeNode()	Casts the Document Fragment to a node.
Document type methods	--
isNull()	Tests if the Document Type is NULL .
makeNode()	Casts the document type to a node.
findEntity()	Finds the specified entity in the document type.
findNotation()	Finds the specified notation in the document type.
getPublicId()	Retrieves the public ID of the document type.
getSystemId()	Retrieves the system ID of the document type.
writeExternalDTDToFile()	Writes the document type definition to a file.
writeExternalDTDToBuffer()	Writes the document type definition to a buffer.
writeExternalDTDToClob()	Writes the document type definition to a clob.
getName()	Retrieves the name of the Document type.
getEntities()	Retrieves the nodemap of entities in the Document type.
getNotations()	Retrieves the nodemap of the notations in the Document type.
Element methods	--
isNull()	Tests if the Element is NULL .
makeNode()	Casts the Element to a node.
getQualifiedName()	Retrieves the qualified name of the element.
getNamespace()	Retrieves the NS URI of the element.
getLocalName()	Retrieves the local name of the element.
getExpandedName()	Retrieves the expanded name of the element.
getChildrenByTagName()	Retrieves the children of the element by tag name.
getElementsByTagName()	Retrieves the elements in the subtree by element.
resolveNamespacePrefix()	Resolve the prefix to a namespace uri.
getTagName()	Retrieves the Tag name of the element.

Table 8–1 Summary DBMS_XMLDOM Methods (Cont.)

Group/Method	Description
getAttribute()	Retrieves the attribute node specified by the name.
setAttribute()	Sets the attribute specified by the name.
removeAttribute()	Removes the attribute specified by the name.
getAttributeNode()	Retrieves the attribute node specified by the name.
setAttributeNode()	Sets the attribute node in the element.
removeAttributeNode()	Removes the attribute node in the element.
normalize()	Normalizes the text children of the element.
Entity methods	--
isNull()	Tests if the Entity is NULL .
makeNode()	Casts the Entity to a node.
getPublicId()	Retrieves the public Id of the entity.
getSystemId()	Retrieves the system Id of the entity.
getNotationName()	Retrieves the notation name of the entity.
Entity reference methods	--
isNull()	Tests if the entity reference is NULL .
makeNode()	Casts the Entity reference to NULL.
Notation methods	--
isNull()	Tests if the notation is NULL .
makeNode()	Casts the notation to a node.
getPublicId()	Retrieves the public Id of the notation.
getSystemId()	Retrieves the system Id of the notation.
Processing instruction methods	--
isNull()	Tests if the processing instruction is NULL .
makeNode()	Casts the Processing instruction to a node.
getData()	Retrieves the data of the processing instruction.
getTarget()	Retrieves the target of the processing instruction.
setData()	Sets the data of the processing instruction.

Table 8–1 Summary DBMS_XMLDOM Methods (Cont.)

Group/Method	Description
Text methods	--
isNull()	Tests if the text is NULL .
makeNode()	Casts the text to a node.
splitText()	Splits the contents of the text node into 2 text nodes.
Document methods	--
isNull()	Tests if the document is NULL.
makeNode()	Casts the document to a node.
newDOMDocument()	Creates a new document.
freeDocument()	Frees the document.
getVersion()	Retrieves the version of the document.
setVersion()	Sets the version of the document.
getCharset()	Retrieves the Character set of the document.
setCharset()	Sets the Character set of the document.
getStandalone()	Retrieves if the document is specified as standalone.
setStandalone()	Sets the document standalone.
writeToFile()	Writes the document to a file.
writeToBuffer()	Writes the document to a buffer.
writeToClob()	Writes the document to a clob.
writeExternalDTDToFile()	Writes the DTD of the document to a file.
writeExternalDTDToBuffer()	Writes the DTD of the document to a buffer.
writeExternalDTDToClob()	Writes the DTD of the document to a clob.
getDoctype()	Retrieves the DTD of the document.
getImplementation()	Retrieves the DOM implementation.
getDocumentElement()	Retrieves the root element of the document.
createElement()	Creates a new element.
createDocumentFragment()	Creates a new document fragment.
createTextNode()	Creates a Text node.

Table 8–1 Summary DBMS_XMLDOM Methods (Cont.)

Group/Method	Description
createComment()	Creates a comment node.
createCDATASection()	Creates a CDATAsection node.
createProcessingInstruction()	Creates a processing instruction.
createAttribute()	Creates an attribute.
createEntityReference()	Creates an Entity reference.
getElementsByTagName()	Retrieves the elements in the by tag name.

PL/SQL DOM API for XMLType (DBMS_XMLDOM) Exceptions

The following lists the PL/SQL DOM API for XMLType (DBMS_XMLDOM) exceptions. For further information, see *Oracle9i XML Developer's Kits Guide - XDK*.

The exceptions have not changed since the prior release:

- INDEX_SIZE_ERR
- DOMSTRING_SIZE_ERR
- HIERARCHY_REQUEST_ERR
- WRONG_DOCUMENT_ERR
- INVALID_CHARACTER_ERR
- NO_DATA_ALLOWED_ERR
- NO_MODIFICATION_ALLOWED_ERR
- NOT_FOUND_ERR
- NOT_SUPPORTED_ERR
- INUSE_ATTRIBUTE_ERR

PL/SQL DOM API for XMLType: Node Types

In the DOM specification, the term “document” is used to describe a container for many different kinds of information or data, which the DOM objectifies. The DOM specifies the way elements within an XML document container are used to create an object-based tree structure and to define and expose interfaces to manage and use the objects stored in XML documents. Additionally, the DOM supports storage of documents in diverse systems.

When a request such as `getNodeType(myNode)` is given, it returns `myNodeType`, which is the node type supported by the parent node. These constants represent the different types that a node can adopt:

- ELEMENT_NODE
- ATTRIBUTE_NODE
- TEXT_NODE
- CDATA_SECTION_NODE
- ENTITY_REFERENCE_NODE
- ENTITY_NODE
- PROCESSING_INSTRUCTION_NODE
- COMMENT_NODE
- DOCUMENT_NODE
- DOCUMENT_TYPE_NODE
- DOCUMENT_FRAGMENT_NODE
- NOTATION_NODE

Table 8–2 shows the node types for XML and HTML and the allowed corresponding children node types.

Table 8–2 XML and HTML DOM Node Types and Corresponding Children Node Types

Node Type	Children Node Types
Document	Element (maximum of one), ProcessingInstruction, Comment, DocumentType (maximum of one)
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	No children
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Text, EntityReference
ProcessingInstruction	No children
Comment	No children
Text	No children

Table 8–2 XML and HTML DOM Node Types and Corresponding Children Node Types (Cont.)

Node Type	Children Node Types
CDATASection	No children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	No children

Oracle XML DB DOM API for XMLType also specifies these interfaces:

- **A NodeList interface** to handle ordered lists of Nodes, for example:
 - The children of a Node
 - Elements returned by the `getElementsByTagName` method of the element interface
- **A NamedNodeMap interface** to handle unordered sets of nodes, referenced by their name attribute, such as the attributes of an element.

Working with XML Schema-Based XML Instances

This release introduces several extensions for character-set conversion and input and output to and from a file system. As stated earlier in this chapter, applications written against the PL/SQL Parser APIs in the previous release continue to work, but require some modifications that are described in the following sections.

PL/SQL API for XMLType is optimized to operate on XML schema-based XML instances.

A new function is provided, `newDOMDocument` that constructs a DOM Document handle given an XMLType value.

A typical usage scenario would be for a PL/SQL application to:

1. Fetch or construct an XMLType instance
2. Construct a DOMDocument node over the XMLType instance
3. Use the DOM API to access and manipulate the XML data

Note: For `DOMDocument`, node types represent handles to XML fragments but do not represent the data itself.

For example, if you copy a node value, `DOMDocument` simply 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 actual data and reflects the results of all DOM operations on it.

DOM NodeList and NamedNodeMap Objects

`NodeList` and `NamedNodeMap` objects in the DOM are live; that is, changes to the underlying document structure are reflected in all relevant `NodeList` and `NamedNodeMap` objects.

For example, if a DOM user gets a `NodeList` object containing the children of an element, and then subsequently adds more children to that element (or removes children, or modifies them), those changes are automatically propagated in the `NodeList`, without further 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.

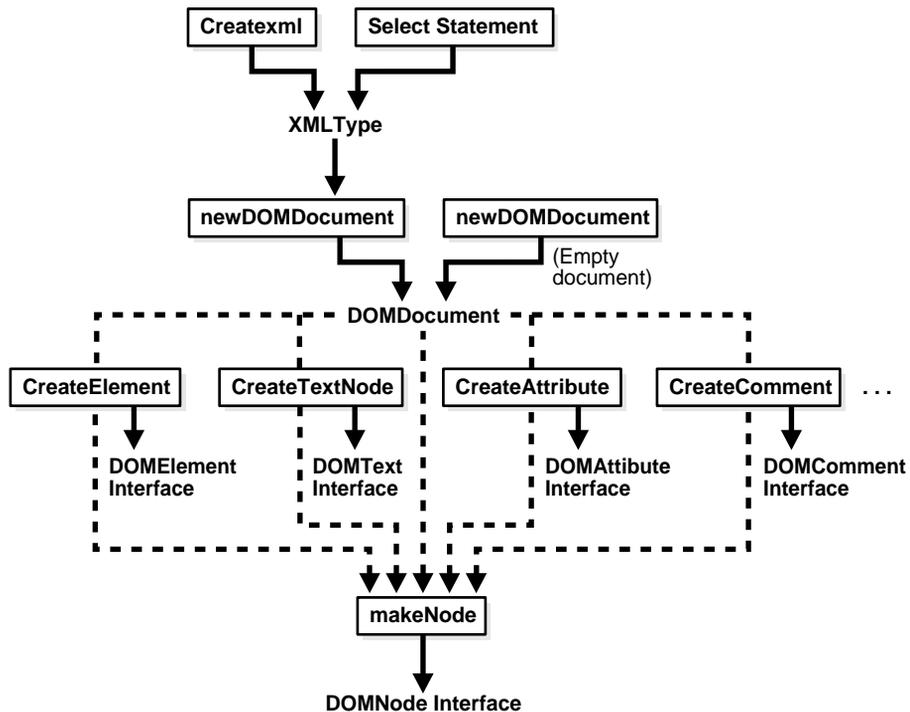
PL/SQL DOM API for XMLType (DBMS_XMLDOM): Calling Sequence

[Figure 8-1](#) illustrates the PL/SQL DOM API for `XMLType` (`DBMS_XMLDOM`) calling sequence.

You can create a DOM document (`DOMDocument`) from an existing `XMLType` or as an empty document.

1. The `newDOMDocument` procedure processes the `XMLType` or empty document. This creates a `DOMDocument`.
2. You can use the DOM API methods such as, `createElement`, `createText`, `createAttribute`, and `createComment`, and so on, to traverse and extend the DOM tree. See [Table 8-1](#) for a full list of available methods.
3. The results of these methods (`DOMElement`, `DOMText`, and so on) can also be passed to `makeNode` to obtain the `DOMNode` interface.

Figure 8-1 PL/SQL DOM API for XMLType: Calling Sequence



PL/SQL DOM API for XMLType Examples

Example 8-1 Creating and Manipulating a DOM Document

This example illustrates how to create a `DOMDocument` handle for an example element `PERSON`:

```

-- This example illustrates how to create a DOMDocument handle for an example
element PERSON:
declare
  var      XMLType;
  doc      dbms_xmldom.DOMDocument;
  ndoc     dbms_xmldom.DOMNode;
  docelem  dbms_xmldom.DOMELEMENT;
  node     dbms_xmldom.DOMNode;
  childnode dbms_xmldom.DOMNode;

```

```

nodelist dbms_xmldom.DOMNodelist;
buf      varchar2(2000);
begin
  var := xmltype('<PERSON> <NAME> ramesh </NAME> </PERSON>');

  -- Create DOMDocument handle:
  doc := dbms_xmldom.newDOMDocument(var);
  ndoc := dbms_xmldom.makeNode(doc);

  dbms_xmldom.writetobuffer(ndoc, buf);
  dbms_output.put_line('Before: ' || buf);

  docelem := dbms_xmldom.getDocumentElement( doc );

  -- Access element:
  nodelist := dbms_xmldom.getElementsByTagName(docelem, 'NAME');
  node := dbms_xmldom.item(nodelist, 0);
  childnode := dbms_xmldom.getFirstChild(node);

  -- Manipulate:
  dbms_xmldom.setNodeValue(childnode, 'raj');

  dbms_xmldom.writetobuffer(ndoc, buf);
  dbms_output.put_line('After: ' || buf);
end;
/

```

Example 8–2 Creating a DOM Document Using sys.xmltype

This example creates a DOM document from an XMLType:

```

declare
  doc dbms_xmldom.DOMDocument;

  buf  varchar2(32767);

begin
  -- new document
  doc := dbms_xmldom.newDOMDocument(sys.xmltype('<person> <name>Scott</name>
</person>'));
  dbms_xmldom.writeToBuffer(doc, buf);
  dbms_output.put_line(buf);
end;
/

```

Example 8–3 Creating an Element Node

```
-- This example creates an element node starting from an empty DOM document:
declare
  doc          dbms_xmlDOM.DOMDocument;
  elem         dbms_xmlDOM.DOMELEMENT;
  nelem        dbms_xmlDOM.DOMNode;
begin
  -- new document
  doc := dbms_xmlDOM.newDOMDocument;

  -- create a element node
  elem := dbms_xmlDOM.createElement(doc, 'ELEM');

  -- make node
  nelem := dbms_xmlDOM.makeNode(elem);
  dbms_output.put_line(dbms_xmlDOM.getNodeName(nelem));
  dbms_output.put_line(dbms_xmlDOM.getNodeValue(nelem));
  dbms_output.put_line(dbms_xmlDOM.getNodeType(nelem));
end;
/
```

PL/SQL Parser API for XMLType (DBMS_XMLPARSER)

XML documents are made up of storage units, called *entities*, that contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism for imposing constraints on the storage layout and logical structure.

A software module called an XML parser or processor reads XML documents and provides access to their content and structure. An XML parser usually does its work on behalf of another module, typically the application.

PL/SQL Parser API for XMLType: Features

In general, PL/SQL Parser API for XMLType (DBMS_XMLPARSER) performs the following tasks:

- Builds a result tree that can be accessed by PL/SQL APIs
- Raises an error if the parsing fails

[Table 8–3](#) lists the PL/SQL Parser API for XMLType (DBMS_XMLPARSER) methods.

Table 8–3 DBMS_XMLPARSER Methods

Method	Arguments, Return Values, and Results
parse	Argument: (url VARCHAR2) Result: Parses XML stored in the given URL or file and returns the built DOM Document
newParser	Returns: A new parser instance
parse	Argument: (p Parser, url VARCHAR2) Result: Parses XML stored in the given URL or file
parseBuffer	Argument: (p Parser, doc VARCHAR2) Result: Parses XML stored in the given buffer
parseClob	Argument: (p Parser, doc CLOB) Result: Parses XML stored in the given CLOB
parseDTD	Argument: (p Parser, url VARCHAR2, root VARCHAR2) Result: Parses XML stored in the given URL or file
parseDTDBuffer	Argument: (p Parser, dtd VARCHAR2, root VARCHAR2) Result: Parses XML stored in the given buffer
parseDTDClob	Argument: (p Parser, dtd CLOB, root VARCHAR2) Result: Parses XML stored in the given clob
setBaseDir	Argument: (p Parser, dir VARCHAR2) Result: Sets base directory used to resolve relative URLs
showWarnings	Argument: (p Parser, yes BOOLEAN) Result: Turns warnings on or off
setErrorLog	Argument: (p Parser, fileName VARCHAR2) Result: Sets errors to be sent to the specified file
setPreserveWhitespace	Argument: (p Parser, yes BOOLEAN) Result: Sets white space preserve mode
setValidationMode	Argument: (p Parser, yes BOOLEAN) Result: Sets validation mode

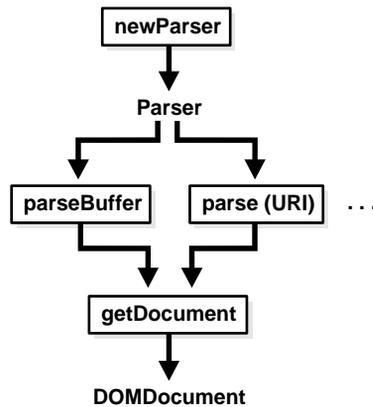
Table 8–3 DBMS_XMLPARSER Methods (Cont.)

Method	Arguments, Return Values, and Results
getValidationMode	Argument: (p Parser) Result: Gets validation mode
setDoctype	Argument: (p Parser, dtd DOMDocumentType) Result: Sets DTD
getDoctype	Argument: (p Parser) Result: Gets DTD
getDocument	Argument: (p Parser) Result: Gets DOM document
freeParser	Argument: (p Parser) Result: Frees a Parser object

PL/SQL Parser API for XMLType (DBMS_XMLPARSER): Calling Sequence

Figure 8–2 illustrates the PL/SQL Parser for XMLType (DBMS_XMLPARSER) calling sequence:

1. newParser method can be used to construct a Parser instance.
2. XML documents can then be parsed using the Parser with methods such as, parseBuffer, parseClob, parse (URI), and so on. See Table 8–3 for a full list of Parser methods.
3. An error is raised if the input is not a valid XML document.
4. To use the PL/SQL DOM API for XMLType on the parsed XML document instance, you need to call getDocument on the Parser to obtain a DOMDocument interface.

Figure 8–2 PL/SQL Parser API for XMLType: Calling Sequence

PL/SQL Parser API for XMLType Example

Example 8–4 Parsing an XML Document

This example parses a simple XML document and enables DOM APIs to be used.

```

declare
  indoc      VARCHAR2(2000);
  indomdoc   dbms_xmldom.domdocument;
  innode     dbms_xmldom.domnode;
  myParser   dbms_xmlparser.Parser;
begin
  indoc      := '<emp><name> Scott </name></emp>';
  myParser   := dbms_xmlparser.newParser;
  dbms_xmlparser.parseBuffer(myParser, indoc);
  indomdoc   := dbms_xmlparser.getDocument(myParser);
  innode     := dbms_xmldom.makeNode(indomdoc);
  -- DOM APIs can be used here
end;
/

```

PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)

W3C XSL Recommendation describes rules for transforming a source tree into a result tree. A transformation expressed in eXtensible Stylesheet Language Transformation (XSLT) is called an XSL stylesheet. The transformation specified is achieved by associating patterns with templates defined in the XSL stylesheet. A template is instantiated to create part of the result tree.

Enabling Transformations and Conversions with XSLT

The Oracle XML DB PL/SQL DOM API for XMLType also supports eXtensible Stylesheet Language Transformation (XSLT). This enables transformation from one XML document to another, or conversion into HTML, PDF, or other formats. XSLT is also widely used to convert XML to HTML for browser display.

The embedded XSLT processor follows eXtensible Stylesheet Language (XSL) statements and traverses the DOM tree structure for XML data residing in XMLType. Oracle XML DB applications do not require a separate parser as did the prior release's XML Parser for PL/SQL. However, applications requiring external processing can still use the XML Parser for PL/SQL first to expose the document structure.

Note: The XML Parser for PL/SQL in Oracle XDK parses an XML document (or a standalone DTD) so that the XML document can be processed by an application, typically running on the client. PL/SQL APIs for XMLType are used for applications that run on the server and are natively integrated in the database. Benefits include performance improvements and enhanced access and manipulation options.

See Also: [Appendix D, "XSLT Primer"](#)

PL/SQL XSLT Processor for XMLType: Features

PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) is Oracle XML DB's implementation of the XSL processor. This follows the W3C XSLT recommendation working draft (rev WD-xslt-19990813). It includes the required behavior of an XSL processor in terms of how it must read XSL stylesheets and the transformations it must achieve.

The types and methods of PL/SQL XSLT Processor are made available by the PL/SQL package, `DBMS_XSLPROCESSOR`.

PL/SQL XSLT Processor API (DBMS_XSLPROCESSOR): Methods

The methods in PL/SQL XSLT Processor API (`DBMS_XSLPROCESSOR`) use two PL/SQL types specific to the XSL Processor implementation. These are the `Processor` type and the `Stylesheet` type.

[Table 8–4](#) lists PL/SQL XSLT Processor (`DBMS_XSLPROCESSOR`) methods.

Note: There is no space between the method declaration and the arguments, for example: `processXSL(p Processor, ss Stylesheet, xmldoc DOMDocument)`

Table 8–4 *DBMS_XSLPROCESSOR Methods* (Page 1 of 2)

Method	Argument or Return Values or Result
<code>newProcessor</code>	Returns: a new processor instance
<code>processXSL</code>	Argument: (p Processor, ss Stylesheet, xmldoc DOMDocument) Result: Transforms input XML document using given DOMDocument and stylesheet
<code>processXSL</code>	Argument: (p Processor, ss Stylesheet, xmldoc DOMDocumentFragment) Result: Transforms input XML document using given DOMDocumentFragment and stylesheet
<code>showWarnings</code>	Argument: (p Processor, yes BOOLEAN) Result: Turn warnings on or off
<code>setErrorLog</code>	Argument: (p Processor, Filename VARCHAR2) Result: Sets errors to be sent to the specified file
<code>NewStylesheet</code>	Argument: (Input VARCHAR2, Reference VARCHAR2) Result: Sets errors to be sent to the specified file
<code>transformNode</code>	Argument: (n DOMNode, ss Stylesheet) Result: Transforms a node in a DOM tree using the given stylesheet
<code>selectNodes</code>	Argument: (n DOMNode, pattern VARCHAR2) Result: Selects nodes from a DOM tree which match the given pattern

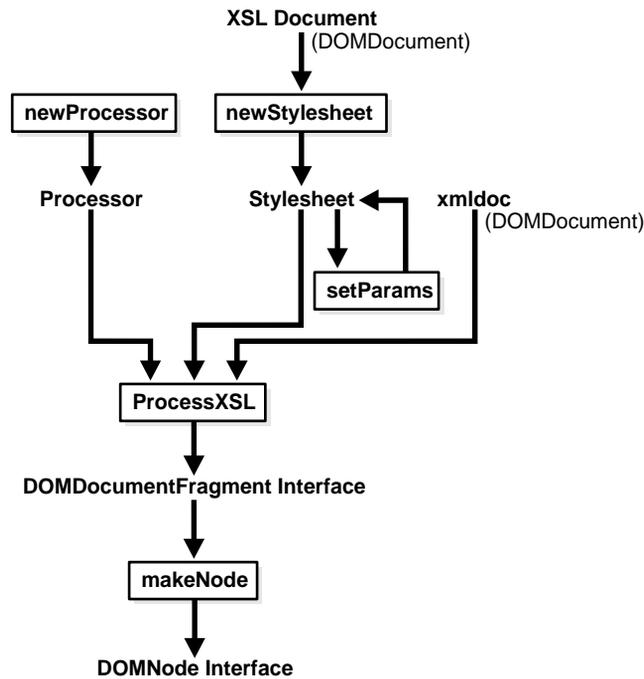
Table 8–4 DBMS_XSLPROCESSOR Methods (Cont.) (Page 2 of 2)

Method	Argument or Return Values or Result
selectSingleNode	Argument: (n DOMNode, pattern VARCHAR2) Result: Selects the first node from the tree that matches the given pattern
valueOf	Argument: (n DOMNode, pattern VARCHAR2) Result: Retrieves the value of the first node from the tree that matches the given pattern
setParam	Argument: (ss Stylesheet, name VARCHAR2, value VARCHAR2) Result: Sets a top level paramter in the stylesheet
removeParam	Argument: (ss Stylesheet, name VARCHAR2) Result: Removes a top level stylesheet parameter
ResetParams	Argument: (ss Stylesheet) Result: Resets the top-level stylesheet parameters
freeStylesheet	Argument: (ss Stylesheet) Result: Frees a Stylesheet object
freeProcessor	Argument: (p Processor) Result: Frees a Processor object

PL/SQL Parser API for XMLType (DBMS_XSLPROCESSOR): Calling Sequence

Figure 8–2 illustrates the XSLT Processor for XMLType (DBMS_XSLPROCESSOR) calling sequence:

1. An XSLT Processor can be constructed using the method `newProcessor`.
2. To build a `Stylesheet` from a DOM document use method `newStylesheet`.
3. Optionally, you can set parameters to the `Stylesheet` using the call `setParams`.
4. The XSLT processing can then be executed with the call `processXSL` using the processor and `Stylesheet` created in Steps 1 - 3.
5. Pass the XML document to be transformed to the call `processXSL`.
6. The resulting `DOMDocumentFragment` interface can be operated on using the PL/SQL DOM API for XMLType.

Figure 8–3 PL/SQL XSLT Processor for XMLType: Calling Sequence

PL/SQL XSLT Processor for XMLType Example

Example 8–5 Transforming an XML Document Using an XSL Stylesheet

This example transforms an XML document by using the `processXSL` call. Expect the following output (XML with tags ordered based on tag name):

```

<emp>
  <empno>1</empno>
  <fname>robert</fname>
  <job>engineer</job>
  <lname>smith</lname>
  <sal>1000</sal>
</emp>

```

```

declare
  indoc      VARCHAR2(2000);
  xsldoc    VARCHAR2(2000);

```

```

myParser      dbms_xmlparser.Parser;
indomdoc      dbms_xmldom.domdocument;
xsltDOMdoc    dbms_xmldom.domdocument;
xsl           dbms_xslprocessor.stylesheet;
outDOMdocf    dbms_xmldom.domdocumentfragment;
outnode       dbms_xmldom.domnode;
proc          dbms_xslprocessor.processor;
buf           varchar2(2000);
begin
    indoc       := '<emp><empno> 1</empno> <fname> robert </fname> <lname>
smith</lname> <sal>1000</sal> <job> engineer </job> </emp>';
    xslDoc      :=
'<?xml version="1.0"?>
  <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output encoding="utf-8"/>
  <!-- alphabetizes an xml tree -->
  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates select="*|text()">
        <xsl:sort select="name(.)" data-type="text" order="ascending"/>
      </xsl:apply-templates>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="text()">
    <xsl:value-of select="normalize-space(.)"/>
  </xsl:template>
</xsl:stylesheet>';

    myParser := dbms_xmlparser.newParser;
    dbms_xmlparser.parseBuffer(myParser, indoc);
    indomdoc := dbms_xmlparser.getDocument(myParser);

    dbms_xmlparser.parseBuffer(myParser, xslDoc);
    xsltDOMdoc := dbms_xmlparser.getDocument(myParser);

    xsl       := dbms_xslprocessor.newstylesheet(xsltDOMdoc, '');
    proc      := dbms_xslprocessor.newProcessor;

    --apply stylesheet to DOM document
    outDOMdocf := dbms_xslprocessor.processxsl(proc, xsl, indomdoc);
    outnode    := dbms_xmldom.makeNode(outDOMdocf);
    -- PL/SQL DOM API for XMLType can be used here
    dbms_xmldom.writeToBuffer(outnode, buf);
    dbms_output.put_line(buf);

```

```
end;  
/
```

Java and Java Bean APIs for XMLType

This chapter describes how to use `XMLType` in Java, including fetching `XMLType` data through JDBC and manipulating them using the Java Bean API.

- [Introducing Java DOM and Java Bean APIs for XMLType](#)
- [Java DOM API for XMLType](#)
- [Java DOM API for XMLType Features](#)
- [Java DOM API for XMLType Classes](#)
- [Java Bean API for XMLType](#)
- [Guidelines for Using Java Bean API for XMLType](#)

Introducing Java DOM and Java Bean APIs for XMLType

Oracle XML DB supports the following Java Application Program Interfaces (APIs):

- *Java Document Object Model (DOM) API for XMLType*. This is a generic API for client and server, for both XML schema-based and non-schema-based documents. It is implemented using the Java package `oracle.xml.db.dom`.

To access XMLType data using JDBC use the class `oracle.xml.db.XMLType`.

- *Java Bean API for XMLType*. This is a high performance API for the server for XML schema-based documents. It is implemented using Java package `oracle.xml.db.bean`. The Java Bean API for XMLType offers:
 - An optimized memory footprint. Because in Oracle XML DB, XML schema-based documents offer reduced storage requirements, the Java Bean API for XMLType has a highly optimized memory footprint.
 - Fast data retrieval. Because the Java Bean API for XMLType is XML schema-aware, XML data can be retrieved quickly.

Java beans also have the advantage of being programmer-friendly, because the API uses the XML schema names from which the bean was generated. Java Bean API for XMLType supports any form of XML schema.

For XML documents that do not conform to any XML schema, you can use the Java DOM API for XMLType as it can handle *any* valid XML document.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

Java DOM API for XMLType

Java DOM API for XMLType handles all kinds of valid XML documents irrespective of how they are stored in Oracle XML DB. It presents to the application a uniform view of the XML document irrespective of whether it is XML schema-based or non-schema-based, whatever the underlying storage. Java DOM API works on client and server.

As discussed in [Chapter 8, "PL/SQL API for XMLType"](#), the Oracle XML DB DOM APIs are compliant with W3C DOM Level 1.0 and Level 2.0 Core Recommendation.

Accessing XML Documents in Repository

Oracle XML DB Resource API for Java/JNDI API allows Java applications to access XML documents stored in the Oracle XML DB Repository. Naming conforms to the Java binding for DOM as specified by the W3C DOM Recommendation. Oracle

XML DB Repository hierarchy can store both XML schema-based and non-schema-based documents.

See: [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#)

Accessing XML Documents Stored in Oracle9i Database (Java)

Oracle XML DB provides two ways (part of the Java/JNDI resource APIs) for Java applications to access XML data stored in a database:

- **Using JNDI.** This is a directory-based navigational access to XML documents in Oracle XML DB.
- **Using JDBC.** This is an SQL-based approach for Java applications for accessing any data in Oracle9i database, including XML documents in Oracle XML DB. Use the `oracle.xml.db.dom.XMLType` class, `createXML()` method.

How Java Applications Use JNDI to Access XML Documents in Oracle XML DB

JNDI allows navigational access to XML documents stored in Oracle XML DB Repository hierarchy. This is described in [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#). Depending on the access type specified to JNDI, either a document object or a bean object is returned. The `JNDI.lookup()` method obtains an XML object from Oracle XML DB Repository when given a path name, if the object is an XML schema-based document:

- And a bean has been generated for that XML schema, an instance of the bean class is returned.
- And if no bean has been generated for that XML schema or if the XML is non-XML schema-based, then a DOM document interface is returned.

The Java application can also call the Java DOM API for `XMLType` for objects for which beans have been generated. This is because the Java Beans interface is based on an extension to the DOM. Once the application has the object returned by `lookup()`, it can call either the Java DOM API for `XMLType` methods or Java Beans methods on the class to access XML data. For Java DOM API for `XMLType`, an `XMLDocument` class, which implements the DOM interface, is returned.

For example:

```
Document x = (org.w3c.dom.Document)ctx.lookup("/usr/local/bkhaladk/po.xml");
```

How Java Applications Use JDBC to Access XML Documents in Oracle XML DB

JDBC users can query an XMLType table to obtain a JDBC XMLType interface that supports all methods supported by the SQL XMLType data type. The Java (JDBC) API for XMLType interface can implement the DOM document interface.

JavaBean support is not available through JDBC because JDBC can work on both client and server, and the Java Bean API for XMLType is designed to work only in the server.

Example 9–1 XMLType Java: Using JDBC to Query an XMLType Table

The following is an example that illustrates using JDBC to query an XMLType table:

```
import oracle.xdb.XMLType;
...
OraclePreparedStatement stmt = (OraclePreparedStatement)
conn.prepareStatement("select e.poDoc from po_xml_tab e");
ResultSet rset = stmt.executeQuery();
OracleResultSet orset = (OracleResultSet) rset;

while(orset.next())
{
    // get the XMLType
    XMLType poxml = XMLType.createXML(orset.getOPAQUE(1));
    // get the XMLDocument as a string...
    Document podoc = (Document)poxml.getDOM();
}
```

Example 9–2 XMLType Java: Selecting XMLType Data

You can select the XMLType data in JDBC in one of two ways:

- Use the getClobVal() or getStringVal() in SQL and get the result as a oracle.sql.CLOB or java.lang.String in Java. The following Java code snippet shows how to do this:

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

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

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

```

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

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

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

    // now use the CLOB inside the program
    }

```

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

```

import oracle.xdb.XMLType;
...

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

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

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

// get the XML as a string...
String poString = poxml.getStringVal();

```

Example 9-3 XMLType Java: Directly Returning XMLType Data

This example shows the use of `getObject` to directly get the `XMLType` from the `ResultSet`. This is the easiest way to get the `XMLType` from the `ResultSet`.

```

import oracle.xdb.XMLType;
...

```

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

ResultSet rset = stmt.executeQuery();
OracleResultSet orset = (OracleResultSet) rset;
while(orset.next())
    {

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

// get the XML as a string...
String poString = poxml.getStringVal();
    }
```

Using JDBC to Manipulate XML Documents Stored in a Database

You can also update, insert, and delete XMLType data using JDBC.

Example 9-4 XMLType Java: Updating/Inserting/Deleting XMLType Data

You can insert an XMLType in java in one of two ways:

- Bind a CLOB or a string to an INSERT/UPDATE/DELETE statement, and use the XMLType constructor inside SQL to construct the XML instance:

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

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

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

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

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

```

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

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

```

Example 9–5 XMLType Java: Getting Metadata on XMLType

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

```

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

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

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

// Describe the column = the column type comes out as OPAQUE
// and column type name comes out as XMLTYPE
if (mdata.getColumnType(1) == OracleTypes.OPAQUE &&
    mdata.getColumnTypeName(1).compareTo("SYS.XMLTYPE") == 0)
{
    // we know it is an XMLtype
}

```

Example 9–6 XMLType Java: Updating an Element in an XMLType Column

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

```
-- create po_xml_hist table to store old PurchaseOrders
```

```
create table po_xml_hist (
  xpo xmltype
);

/*
  DESCRIPTION
    Example for oracle.xdb.XMLType

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

import java.sql.*;
import java.io.*;

import oracle.xml.parser.v2.*;
import org.xml.sax.*;
import org.w3c.dom.*;

import oracle.jdbc.driver.*;
import oracle.sql.*;

import oracle.xdb.XMLType;

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

    static String updateXML(String xmlTypeStr)
    {
        System.out.println("\n=====");
        System.out.println("xmlType.getStringVal():");
        System.out.println(xmlTypeStr);
        System.out.println("=====");
        String outXML = null;
        try{
            DOMParser parser = new DOMParser();
            parser.setValidationMode(false);
```

```

parser.setPreserveWhitespace (true);

parser.parse(new StringReader(xmlTypeStr));
System.out.println("xmlType.getStringVal(): xml String is well-formed");

XMLDocument doc = parser.getDocument();

NodeList nl = doc.getElementsByTagName("DISCOUNT");

for(int i=0;i<nl.getLength();i++){
    XMLElement discount = (XMLElement)nl.item(i);
    XMLNode textNode = (XMLNode)discount.getFirstChild();
    textNode.setNodeValue("10");
}

StringWriter sw = new StringWriter();
doc.print(new PrintWriter(sw));

outXML = sw.toString();

//print modified xml
System.out.println("\n=====");
System.out.println("Updated PurchaseOrder:");
System.out.println(outXML);
System.out.println("=====");
}
catch ( Exception e )
{
    e.printStackTrace(System.out);
}
return outXML;
}

public static void main(String args[]) throws Exception
{
    try{

        System.out.println("qryStr="+ qryStr);

        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

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

        Statement s = conn.createStatement();

```

```
OraclePreparedStatement stmt;

ResultSet rset = s.executeQuery(qryStr);
OracleResultSet orset = (OracleResultSet) rset;

while(orset.next()){

//retrieve PurchaseOrder xml document from database
XMLType xt = XMLType.createXML(orset.getOPAQUE(1));

    //store this PurchaseOrder in po_xml_hist table
    stmt = (OraclePreparedStatement)conn.prepareStatement(
        "insert into po_xml_hist values(?)");

    stmt.setObject(1,xt); // bind the XMLType instance
    stmt.execute();

//update "DISCOUNT" element
    String newXML = updateXML(xt.getStringVal());

    // create a new instance of an XMLtype from the updated value
    xt = XMLType.createXML(conn,newXML);

// update PurchaseOrder xml document in database
    stmt = (OraclePreparedStatement)conn.prepareStatement(
        "update po_xml_tab x set x.poDoc =? where "+
        "x.poDoc.extract('/PO/PONO/text()').getNumberVal()=200");

    stmt.setObject(1,xt); // bind the XMLType instance
    stmt.execute();

    conn.commit();
    System.out.println("PurchaseOrder 200 Updated!");

}

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

```

    }
  }
}

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

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

```

Here is the resulting updated purchase order in XML:

```

<?xml version = '1.0'?>
<PurchaseOrder>
  <PONO>200</PONO>
  <CUSTOMER>
    <CUSTINO>2</CUSTINO>
    <CUSTNAME>John Nike</CUSTNAME>
    <ADDRESS>
      <STREET>323 College Drive</STREET>
      <CITY>Edison</CITY>
      <STATE>NJ</STATE>
      <ZIP>08820</ZIP>
    </ADDRESS>
    <PHONELIST>
      <VARCHAR2>609-555-1212</VARCHAR2>
      <VARCHAR2>201-555-1212</VARCHAR2>
    </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>20-APR-97</ORDERDATE>
  <SHIPDATE>20-MAY-97 12.00.00.000000 AM</SHIPDATE>
  <LINEITEMS>
    <LINEITEM_TYP LineItemNo="1">
      <ITEM StockNo="1004">
        <PRICE>6750</PRICE>
        <TAXRATE>2</TAXRATE>
      </ITEM>
      <QUANTITY>1</QUANTITY>
      <DISCOUNT>10</DISCOUNT>
    </LINEITEM_TYP>
    <LINEITEM_TYP LineItemNo="2">
      <ITEM StockNo="1011">

```

```
<PRICE>4500.23</PRICE>
<TAXRATE>2</TAXRATE>
</ITEM>
<QUANTITY>2</QUANTITY>
<DISCOUNT>10</DISCOUNT>
</LINEITEM_TYP>
</LINEITEMS>
<SHIPTOADDR>
  <STREET>55 Madison Ave</STREET>
  <CITY>Madison</CITY>
  <STATE>WI</STATE>
  <ZIP>53715</ZIP>
</SHIPTOADDR>
</PurchaseOrder>
```

Example 9–7 Manipulating an XMLType Column

This example performs the following:

- Selects an XMLType from an XMLType table
- Extracts portions of the XMLType based on an XPath expression
- Checks for the existence of elements
- Transforms the XMLType to another XML format based on XSL
- Checks the validity of the XMLType document against an XML schema

```
import java.sql.*;
import java.io.*;
import java.net.*;
import java.util.*;

import oracle.xml.parser.v2.*;
import oracle.xml.parser.schema.*;
import org.xml.sax.*;
import org.w3c.dom.*;

import oracle.xml.sql.dataset.*;
import oracle.xml.sql.query.*;
import oracle.xml.sql.docgen.*;
import oracle.xml.sql.*;

import oracle.jdbc.driver.*;
import oracle.sql.*;
```

```

import oracle.xdb.XMLType;

public class tkxmtpk1
{
    static String conStr = "jdbc:oracle:oci8:@";
    static String user = "tpjc";
    static String pass = "tpjc";
    static String qryStr = "select x.resume from t1 x where id<3";
    static String xslStr =
        "<?xml version='1.0' ?> " +
        "<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1
999/XSL/Transform'> " +
        "<xsl:template match='ROOT'> " +
        "<xsl:apply-templates/> " +
        "</xsl:template> " +
        "<xsl:template match='NAME'> " +
        "<html> " +
        "  <body> " +
        "    This is Test " +
        "  </body> " +
        "</html> " +
        "</xsl:template> " +
        "</xsl:stylesheet>";

    static void parseArg(String args[])
    {
        conStr = (args.length >= 1 ? args[0]:conStr);
        user = (args.length >= 2 ? args[1].substring(0, args[1].indexOf("/")):user);
        pass = (args.length >= 2 ? args[1].substring(args[1].indexOf("/")+1):pass);
        qryStr = (args.length >= 3 ? args[2]:qryStr);
    }
    /**
     * Print the byte array contents
     */
    static void showValue(byte[] bytes) throws SQLException
    {
        if (bytes == null)
            System.out.println("null");
        else if (bytes.length == 0)
            System.out.println("empty");
        else
        {
            for(int i=0; i<bytes.length; i++)
                System.out.print((bytes[i]&0xff)+" ");
        }
    }
}

```

```
        System.out.println();
    }
}

public static void main(String args[]) throws Exception
{
    tkxmjnd1 util = new tkxmjnd1();

    try{

        if( args != null )
            parseArg(args);

        //      System.out.println("conStr=" + conStr);
        System.out.println("user/pass=" + user + "/" + pass );
        System.out.println("qryStr="+ qryStr);

        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

        Connection conn = DriverManager.getConnection(conStr, user, pass);
        Statement s = conn.createStatement();

        ResultSet rset = s.executeQuery(qryStr);
        OracleResultSet orset = (OracleResultSet) rset;
        OPAQUE xml;

        while(orset.next()){
            xml = orset.getOPAQUE(1);
            oracle.xdb.XMLType xt = oracle.xdb.XMLType.createXML(xml);

            System.out.println("Testing getDOM() ...");
            Document doc = xt.getDOM();
            util.printDocument(doc);

            System.out.println("Testing getBytesValue() ...");
            showValue(xt.getBytesValue());

            System.out.println("Testing existsNode() ...");
            try {
                System.out.println("existsNode(/)" + xt.existsNode("/", null));
            }
            catch (SQLException e) {
                System.out.println("Thin driver Expected exception: " + e);
            }
        }
    }
}
```

```

System.out.println("Testing extract() ...");
try {
    XMLType xt1 = xt.extract("/RESUME", null);
    System.out.println("extract RESUME: " + xt1.getStringVal());
    System.out.println("should be Fragment: " + xt1.isFragment());
}
catch (SQLException e) {
    System.out.println("Thin driver Expected exception: " + e);
}

System.out.println("Testing isFragment() ...");
try {
    System.out.println("isFragment = " + xt.isFragment());
}
catch (SQLException e) {
    System.out.println("Thin driver Expected exception: " + e);
}

System.out.println("Testing isSchemaValid() ...");
try {
    System.out.println("isSchemaValid(): " + xt.isSchemaValid(null, "RES
UME"));
}
catch (SQLException e) {
    System.out.println("Thin driver Expected exception: " + e);
}

System.out.println("Testing transform() ...");
System.out.println("XSLDOC: \n" + xslStr + "\n");
try {
    /* XMLType xslDoc = XMLType.createXML(conn, xslStr);
    System.out.println("XSLDOC Generated");
    System.out.println("After transformation:\n" + (xt.transform(xslDoc,
null)).getStringVal()); */
    System.out.println("After transformation:\n" + (xt.transform(null, n
ull)).getStringVal());
}
catch (SQLException e) {
    System.out.println("Thin driver Expected exception: " + e);
}

System.out.println("Testing createXML(conn, doc) ...");
try {
    XMLType xt1 = XMLType.createXML(conn, doc);
    System.out.println(xt1.getStringVal());
}

```

```
        }
        catch (SQLException e) {
            System.out.println("Got exception: " + e);
        }
    }
}
catch( Exception e )
{
    e.printStackTrace(System.out);
}
}
```

Java DOM API for XMLType Features

When you use the JNDI API to get XML data from Oracle XML DB, you get an `XMLDocument` object that represents the XML data or file you retrieve. The `XMLDocument` object returned by JNDI implements the DOM interface. From this document interface you can get the elements of the document and perform all the operations specified in the W3C DOM specification. The DOM works on:

- Any type of XML document:
 - XML schema-based
 - Non-XML schema-based
- Any type of underlying storage used by the document:
 - CLOB
 - BLOB
 - Object-relational.

The Java DOM API for `XMLType` supports deep or shallow searching in the document to retrieve children and properties of XML objects such as name, namespace, and so on. Conforming to the DOM 2.0 recommendation, Java DOM API for `XMLType` is namespace aware.

Creating XML Documents Programmatically

Java API for `XMLType` also allows applications to create XML documents programmatically. This way applications can create XML documents on the fly (or dynamically) that either conform to a preregistered XML schema or are non-XML schema-based documents.

Creating XML Schema-Based Documents

To create XML schema-based documents, Java DOM API for XMLType uses an extension to specify which XML schema URL to use. For XML schema-based documents, it also verifies that the DOM being created conforms to the specified XML schema, that is, that the appropriate children are being inserted under the appropriate documents.

Note: In this release, 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's JNDI API `bind()`. The XML document is stored in the appropriate format:

- As a CLOB or BLOB for non-XML schema-based documents
- In the format specified by the XML schema for XML schema-based documents

Example 9–8 Java DOM API for XMLType: Creating a DOM Object and Storing It in the Format Specified by the XML Schema

The following example shows how you can use Java DOM API for XMLType to create a DOM object and store it in the format specified by the XML schema. Note that the validation against the XML schema is not shown here.

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

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

Element rootElem = poDOM.createElement("PO");
poDOM.insertBefore(poDOM, rootElem, null);

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

JDBC/SQLJ

An `XMLType` instance is represented in Java by `oracle.xdb.XMLType`. When an instance of `XMLType` is fetched using JDBC, it is automatically manifested as an object of the provided `XMLType` class. Similarly, objects of this class can be bound as values to Data Manipulation Language (DML) statements where an `XMLType` is expected. The same behavior is supported in SQLJ clients.

Java DOM API for XMLType Classes

Oracle XML DB supports the W3C DOM Level 2 recommendation.

In addition to the W3C recommendation, Oracle XML DB DOM API also provides Oracle-specific extensions, mainly to facilitate your application interfacing with Oracle XDK for Java. A list of the Oracle extensions is found at:

http://otn.oracle.com/docs/tech/xml/xdk_java/content.html

`XMLDocument()` is a class that represents the DOM for the instantiated XML document. You can retrieve the `XMLType` from the XML document using the function `getXMLType()` on `XMLDocument()` class.

Table 9-1 lists the Java DOM API for `XMLType` classes and the W3C DOM interfaces they implement.

Table 9-1 Java DOM API for XMLType: Classes

Java DOM API for XMLType Class	W3C DOM Interface Recommendation Class
<code>oracle.xdb.dom.XMLDocument</code>	<code>org.w3c.dom.Document</code>
<code>oracle.xdb.dom.XMLCDATA</code>	<code>org.w3c.dom.CDataSection</code>
<code>oracle.xdb.dom.XMLComment</code>	<code>org.w3c.dom.Comment</code>
<code>oracle.xdb.dom.XMLPI</code>	<code>org.w3c.dom.ProcessingInstruction</code>
<code>oracle.xdb.dom.XMLText</code>	<code>org.w3c.dom.Text</code>
<code>oracle.xdb.dom.XMLEntity</code>	<code>org.w3c.dom.Entity</code>
<code>oracle.xdb.dom.DTD</code>	<code>org.w3c.dom.DocumentType</code>
<code>oracle.xdb.dom.XMLNotation</code>	<code>org.w3c.dom.Notation</code>
<code>oracle.xdb.dom.XMLNodeList</code>	<code>org.w3c.dom.NodeList</code>
<code>oracle.xdb.dom.XMLAttribute</code>	<code>org.w3c.dom.Attribute</code>

Table 9–1 Java DOM API for XMLType: Classes (Cont.)

Java DOM API for XMLType Class	W3C DOM Interface Recommendation Class
oracle.xdb.dom.XMLDOMImplementation	org.w3c.dom.DOMImplementation
oracle.xdb.dom.XMLElement	org.w3c.dom.Element
oracle.xdb.dom.XMLNamedNodeMap	org.w3c.dom.NamedNodeMap
oracle.xdb.dom.XMLNode	org.w3c.dom.Node

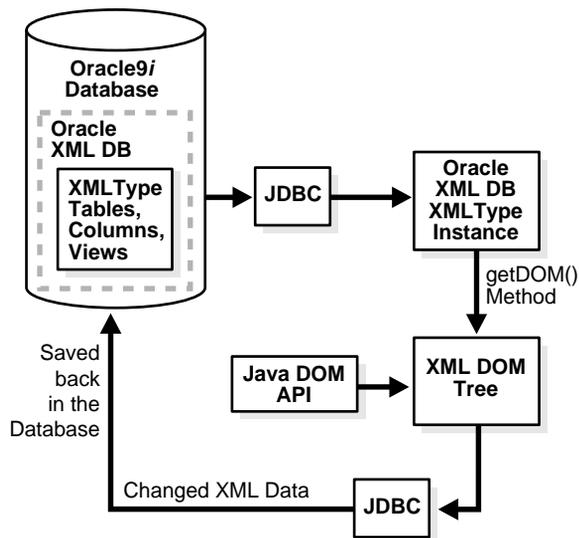
Java DOM API for XMLType: Calling Sequence

The following Java DOM API for XMLType calling sequence description assumes that your XML data is pre-registered with an XML schema and that it is stored in an XMLType datatype column. To use the Java DOM API for XMLType, follow these steps:

1. Retrieve the XML data from the XMLType table or XMLType column in the table. When you fetch XML data, Oracle XML DB creates a DOMDocument instance of XMLType, parsing the document into a DOM tree. You can then manipulate elements in the DOM tree using Java DOM API for XMLType.
2. Use the Java DOM API for XMLType to perform operations and manipulations on elements of the DOM tree.
3. The Java DOM API for XMLType sends the changed XML data back to Oracle XML DB.

Figure 9–1 illustrates the Java DOM API for XMLType calling sequence.

Figure 9–1 Java DOM API for XMLType: Calling Sequence



Java Bean API for XMLType

The Java Bean API for XMLType is a specialized API for working with XML schema-based data. The XML schema must be registered with Oracle XML DB. Java Bean API for XMLType is optimized for processing XML data by Java applications in the server.

See Also: [Chapter 5, "Structured Mapping of XMLType"](#) for details on registering XML schema

Use the Java Bean API for XMLType for applications that:

- Execute in the server
- Use XML schema-based documents

The Java bean source file is stored in your home directory in Oracle XML DB Repository under the `bean/` directory. You can use this source file for the list of beans generated for the specific XML schema.

Java Beans Names Are Always Unique

The rule described here is used for XML schema generation. The package name, `oracle.xml.db.bean`, is prepended to the name of the XML schema.

- If you have specified `BeanName` in the XML schema definition, then that name is used as the class name for the XML schema.
- If you have not specified `BeanName` in the XML schema definition, the name of the element is used as the class name. Thus the Element name inside the package name ensures the uniqueness of the name in the XML schema.

The bean name is always unique and is generated as the XML schema name, appended with its number in the XML schema file. For example, if a file, `emp.xml`, contains three schemas: `Emp`, `Def`, and `Manager`, then:

- The package name for the XML schema `Emp` will be `Emp1`.
- The name of the bean will be `Emp`.

Thus, class `Emp()` will be unique in package `Emp1`. This way Java bean names are always unique.

Use `registerSchema()` to Register Your XML Schema and Generate Java Beans

Java beans are generated when the XML schema is registered with Oracle XML DB using `registerSchema()` with `genbean` flag set to `TRUE`.

Note: You can also generate Java Beans independently using the `generateBean()` function in `DBMS_XMLSCHEMA` package. This can be done as many times as you wish.

The `deleteSchema()` PLSQL call removes the Java bean class files from Oracle XML DB. See [Chapter 5, "Structured Mapping of XMLType"](#).

Guidelines for Using Java Bean API for XMLType

The following are guidelines for using Java Bean API for `XMLType`:

1. First register the XML schema with Oracle XML DB. The XML Schema registration API has a flag that enables the caller to generate a bean for the XML schema. The flag is optional since not all XML schema users use Java Bean API for `XMLType`.

See Also: [Chapter 5, "Structured Mapping of XMLType"](#).

The Java bean generation process generates the Java bean code and compiles it into the server where your XML schema resides. A copy of the source file is also placed in the Oracle XML DB Repository hierarchy in your home directory. This is needed for you to later compile your source code.

2. The Java bean classes are generated in a package with the name of the XML schema file. Different XML schema types are mapped to Java types based on rules specified in the XML schema. See [Table 9–3, "Mapping Between XML Schema, SQL, and Java DataTypes"](#). For example:
 - Oracle XML DB for Java Bean API uses the reflection mechanism to generate class names. For all attributes and children there will be get and set methods in the API. These get and set methods will be suffixed with the name of the child.
 - For server side applications, Java Bean API for XMLType objects are accessed using JNDI `lookup()` API. In this case the `XDB_ACCESS_TYPE` flag specifies `BEAN` as the access type.
3. Java Bean API for XMLType can also be used for creating new schema-based XML documents. The applications can instantiate bean classes and use the `set` methods to set the appropriate data in the document. Similar to DOM, the JNDI `bind()` method is Java bean aware and can be used to save the XML schema document in Oracle XML DB Repository. See also [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#).

[Figure 9–2](#) illustrates the Java Bean API for XMLType calling sequence.

Figure 9–2 Java Bean API for XMLType: Calling Sequence

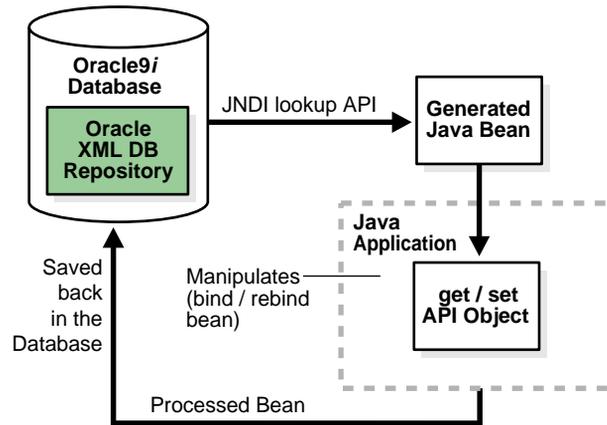


Table 9–2 describes the mapping of XML schema entities to the Java bean classes/methods.

Table 9–2 JavaBeans API for XMLType: Mapping of XML Schema Entities

XML Schema Entity	JavaBeans Classes/Methods	Description
Attribute	get/setAttribute{attrname}	
Complextype	get/set{complextype classname}.	For complex children separate bean classes get generated. Extras (processing instructions/comments). DOM classes for processing-instruction/comments are returned.
Scalar data	get/set{scalar type name}	Children with maxoccurs > 1 Get/set List of type of child

Table 9–3 describes the mapping used by Java Bean API For XMLType between XML schema, SQL, and Java datatypes.

Table 9–3 Mapping Between XML Schema, SQL, and Java DataTypes

XML Schema Data Type	SQL DataType	Java Data Type
Boolean	boolean	boolean

Table 9–3 Mapping Between XML Schema, SQL, and Java DataTypes (Cont.)

XML Schema Data Type	SQL DataType	Java Data Type
String	URI reference	ID
IDREF	ENTITY	NOTATION
Language	NCName	Name
java.lang.String	String	DECIMAL
INTEGER	LONG	SHORT
INT	POSITIVEINTEGER	NONPOSITIVEINTEGER
oracle.sql.Number	int	FLOAT
DOUBLE	oracle.sql.Number	float
TIMEDURATION	TIMEPERIOD	RECURRINGDURATION
DATE	TIME	MONTH, YEAR
RECURRINGDATE	java.sql.Timestamp	Time
REF	oracle.sql.Ref	Ref
BINARY	oracle.sql.RAW	Byte[]
QNAME	java.lang.String	String

Example 9–9 Java Bean API for XMLType

This example illustrates the use of Java Bean API for XMLType. Assume the following XML schema:

```
<schema targetNamespace="http://www.oracle.com/txxmschl.xsd" version="1.0">
<element name = "Employee">
<complexType>
<sequence>
<element name = "EmployeeId" type = "positiveInteger"/>
<element name = "FirstName" type = "string"/>
<element name = "LastName" type = "string"/>
<element name = "Salary" type = "positiveInteger"/>
</sequence>
</complexType>
</element>
</schema>
```

The generated Java bean for this XML schema looks like this:

```
/* bean file generated by Oracle XML DB */
package oracle.xdb.beam.tkxmschl;
import org.w3c.dom.*;
import oracle.xdb.dom.*;
import oracle.xdb.beam.*;
import java.util.Vector;

public class Employee extends XMLTypeBean
{
public Employee(XMLType owner, long xob, long kidnum)
{
super(owner, xob, kidnum);
}

public Employee()
{
super(null, 0, 0);
}

public int getEmployeeId()
{
return super.getScalarint(0);
}

public void setEmployeeId(int val)
{
super.setScalar(0, val);
return;
}

public String getFirstName()
{
return super.getScalarString(1);
}

public void setFirstName(String val)
{
super.setScalar(1, val);
return;
}

public String getLastName()
{
return super.getScalarString(2);
}
}
```

```
public void setLastName(String val)
{
    super.setScalar(2, val);
    return;
}

public int getSalary()
{
    return super.getScalarint(3);
}

public void setSalary(int val)
{
    super.setScalar(3, val);
    return;
}
}
```

Here is the Java Stored Procedure:

```
Hashtable env = new Hashtable();
env.put(Context.PROVIDER_URL, "/");
env.put(Context.INITIAL_CONTEXT_FACTORY, "oracle.xdb.spi.XDBCContextFactory");
Context ctx = (Context)new InitialContext(env);
Employee obj = (Employee)ctx.lookup("/tkxms1dl.xml");
return obj.getFirstName();
```

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

Part IV

Viewing Existing Data as XML

Part IV of this manual introduces you to ways you can view your existing data as XML. It contains the following chapters:

- [Chapter 10, "Generating XML Data from the Database"](#)
- [Chapter 11, "XMLType Views"](#)
- [Chapter 12, "Creating and Accessing Data Through URLs"](#)

Generating XML Data from the Database

This chapter describes Oracle XML DB options for generating XML from the database. It explains in detail, the SQLX standard functions and Oracle-provided functions and packages for generating XML data from relational content.

It contains these sections:

- [Oracle XML DB Options for Generating XML Data From Oracle9i Database](#)
- [Generating XML from the Database Using SQLX Functions](#)
- [XMLElement\(\) Function](#)
- [XMLForest\(\) Function](#)
- [XMLColAttVal\(\) Function](#)
- [XMLSEQUENCE\(\) Function](#)
- [XMLConcat\(\) Function](#)
- [XMLAgg\(\) Function](#)
- [Generating XML from Oracle9i Database Using DBMS_XMLGEN](#)
- [Generating XML Using Oracle-Provided SQL Functions](#)
- [SYS_XMLGEN\(\) Function](#)
- [SYS_XMLAGG\(\) Function](#)
- [Generating XML Using XSQL Pages Publishing Framework](#)
- [Generating XML Using XML SQL Utility \(XSU\)](#)

Oracle XML DB Options for Generating XML Data From Oracle9i Database

Oracle9i supports native XML generation. In this release, Oracle provides you with several new options for generating or regenerating XML data when stored in:

- Oracle9i database, in general
- Oracle9i database in `XMLTypes` columns and tables

[Figure 10-1](#) illustrates the Oracle XML DB options you can use to generate XML from Oracle9i database.

Generating XML Using SQLX Functions

The following SQLX functions are supported in Oracle XML DB:

- ["XMLElement\(\) Function"](#) on page 10-5
- ["XMLForest\(\) Function"](#) on page 10-9
- ["XMLConcat\(\) Function"](#) on page 10-15
- ["XMLAgg\(\) Function"](#) on page 10-17

Generating XML Using Oracle Extensions to SQLX

The following are Oracle extension functions to SQLX:

- ["XMLColAttVal\(\) Function"](#) on page 10-19

Generating XML Using DBMS_XMLGEN

Oracle XML DB supports `DBMS_XMLGEN`, a PL/SQL supplied package. `DBMS_XMLGEN` generates XML from SQL queries. See ["Generating XML from Oracle9i Database Using DBMS_XMLGEN"](#) on page 10-20.

Generating XML Using SQL Functions

Oracle XML DB also supports the following Oracle-provided SQL functions that generate XML from SQL queries:

- ["SYS_XMLGEN\(\) Function"](#) on page 10-41. This operates on rows, generating XML documents.

- ["SYS_XMLAGG\(\) Function"](#) on page 10-50. This operates on groups of rows, aggregating several XML documents into one.
- ["XMLSEQUENCE\(\) Function"](#) on page 10-11. Note that only the cursor version of this function generates XML. This function is also classified as an SQLX function.

Generating XML with XSQL Pages Publishing Framework

["Generating XML Using XSQL Pages Publishing Framework"](#) on page 10-51 can also be used to generate XML from Oracle9i database.

XSQL Pages Publishing Framework, also known as XSQL Servlet, is part of the XDK for Java.

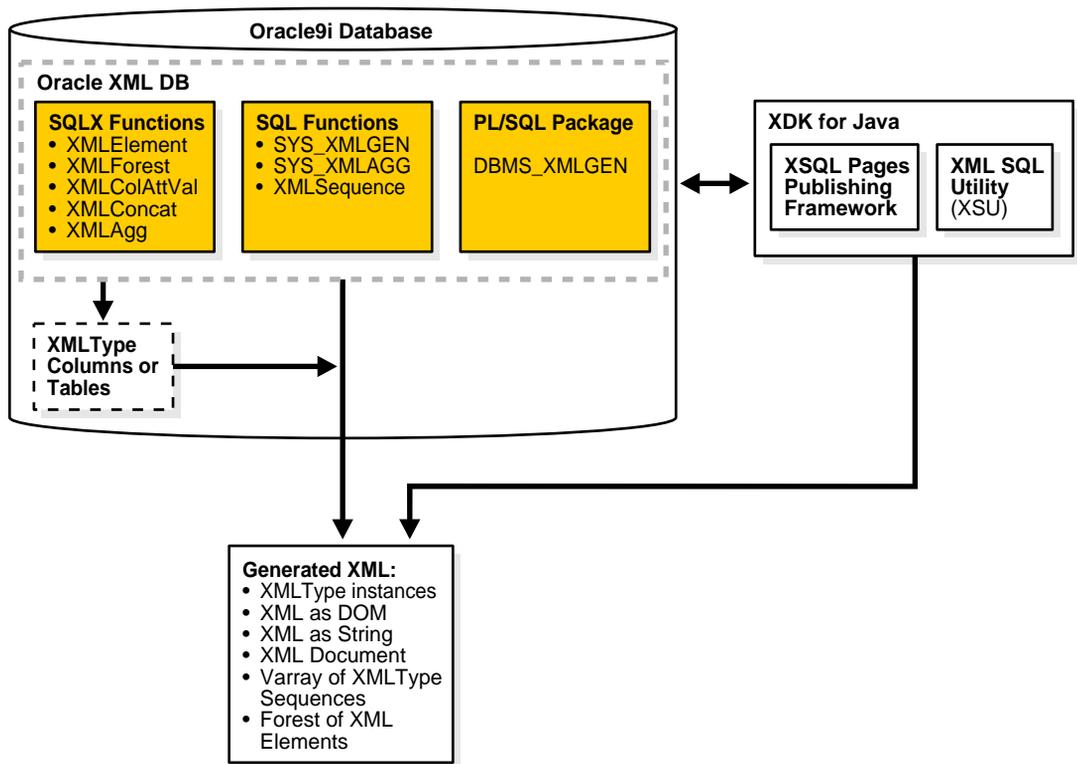
Generating XML Using XML SQL Utility (XSU)

XML SQL Utility (XSU) enables you to perform the following tasks on data in `XMLType` tables and columns:

- Transform data retrieved from object-relational database tables or views into XML.
- Extract data from an XML document, and using a canonical mapping, insert the data into appropriate columns or attributes of a table or a view.
- Extract data from an XML document and apply this data to updating or deleting values of the appropriate columns or attributes.

See Also: ["Generating XML Using XML SQL Utility \(XSU\)"](#) on page 10-54

Figure 10-1 Oracle XML DB Options for Generating XML from Oracle9i Database



See Also:

- [Chapter 6, "Transforming and Validating XMLType Data"](#)
- [Chapter 8, "PL/SQL API for XMLType"](#)
- [Chapter 9, "Java and Java Bean APIs for XMLType"](#)
- *Oracle9i XML API Reference - XDK and Oracle XML DB*

Generating XML from the Database Using SQLX Functions

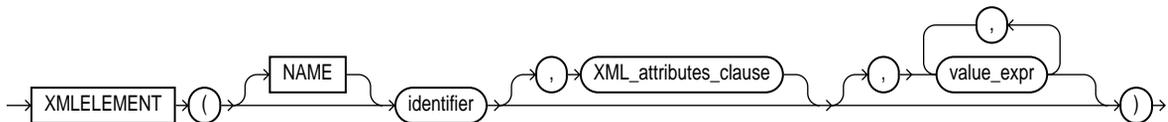
`XMLElement()`, `XMLForest()`, `XMLConcat()`, and `XMLAgg()` belong to the SQLX standard, an emerging SQL standard for XML. Because these are emerging standards the syntax and semantics of these functions are subject to change in the future in order to conform to the standard.

All of the generation functions convert user-defined types (UDTs) to their canonical XML format. In the canonical mapping the user-defined type's attributes are mapped to XML elements.

XMLElement() Function

`XMLElement()` function is based on the emerging SQL XML standard. It takes an element name, an optional collection of attributes for the element, and zero or more arguments that make up the element's content and returns an instance of type `XMLType`. See [Figure 10-2](#). The `XML_attributes_clause` is described in the following section.

Figure 10-2 *XMLElement() Syntax*



It is similar to `SYS_XMLGEN()`, but unlike `SYS_XMLGEN()`, `XMLElement()` does not create an XML document with the prolog (the XML version information). It allows multiple arguments and can include attributes in the XML returned.

`XMLElement()` is primarily used to construct XML instances from relational data. It takes an identifier that is partially escaped to give the name of the root XML element to be created. The identifier does not have to be a column name, or column reference, and cannot be an expression. If the identifier specified is `NULL`, then no element is returned.

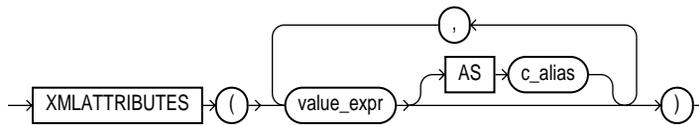
As part of generating a valid XML element name from an SQL identifier, characters that are disallowed in an XML element name are escaped. Partial escaping implies that SQL identifiers other than the ":" sign which are not representable in XML, are escaped using the # sign followed by the character's unicode representation in hexadecimal format. This can be used to specify namespace prefixes for the

elements being generated. The fully escaped mapping escapes all non-XML characters in the SQL identifier name, including the " : " character.

XML_Attributes_Clause

XMLElement() also takes an optional XMLAttributes() clause, which specifies the attributes of that element. This can be followed by a list of values that make up the children of the newly created element. See [Figure 10-3](#).

Figure 10-3 XML_attributes_clause Syntax



In the XMLAttributes() clause, the value expressions are evaluated to get the values for the attributes. For a given value expression, if the AS clause is omitted, the fully escaped form of the column name is used as the name of the attribute. If the AS clause is specified, then the partially escaped form of the alias is used as the name of the attribute. If the expression evaluates to NULL, then no attribute is created for that expression. The type of the expression cannot be an object type or collection.

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

Example 10-1 XMLElement(): Generating an Element for Each Employee

The following example produces an Emp XML element for each employee, with the employee's name as its content:

```
SELECT e.employee_id, XMLELEMENT ( "Emp", e.fname || ' ' || e.lname ) AS "result"
FROM employees e
WHERE employee_id > 200;
```

```
-- This query produces the following typical result:
-- ID      result
-- -----
-- 1001 <Emp>John Smith</Emp>
-- 1206 <Emp>Mary Martin</Emp>
```

XMLElement() can also be nested to produce XML data with a nested structure.

Example 10–2 XMLElement(): Generating Nested XML

To produce an Emp element for each employee, with elements that provide the employee's name and start date:

```
SELECT XMLELEMENT("Emp", XMLELEMENT("name", e.fname || ' ' || e.lname),
                                XMLELEMENT("hiredate", e.hire)) AS "result"
FROM employees e
WHERE employee_id > 200 ;
```

This query produces the following typical XML result:

```
result
-----
<Emp>
  <name>John Smith</name>
  <hiredate>2000-05-24</hiredate>
</Emp>
<Emp>
  <name>Mary Martin</name>
  <hiredate>1996-02-01</hiredate>
</Emp>
```

Note: Attributes, if they are specified, appear in the second argument of XMLElement() as:

“XMLATTRIBUTES (attribute, ...)”.

Example 10–3 XMLElement(): Generating an Element for Each Employee with ID and Name Attribute

This example produces an Emp element for each employee, with an id and name attribute.

```
SELECT XMLELEMENT ("Emp",
                  XMLATTRIBUTES (e.id,e.fname || ' ' || e.lname AS "name")) AS "result"
FROM employees e
WHERE employee_id > 200;
```

This query produces the following typical XML result fragment:

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

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

If the name of the element or attribute is being created from the ALIAS specified in the AS clause, then partially escaped mapping is used. If the name of the element or attribute is being created from a column reference, then the fully escaped mapping is used.

The following example illustrates these mappings:

```
SELECT XMLELEMENT ( "Emp:Exempt",
  XMLATTRIBUTES ( e.fname, e.lname AS "name:last", e."name:middle")) AS "result"
  FROM employees e
  WHERE ... ;
```

This query could produce the following XML result:

```
<Emp:Exempt FNAME="John" name:last="Smith" name_x003A_middle="Quincy" /> ...
```

Note: XMLElement() does not validate the document produced with these namespace prefixes and it is the responsibility of the user to ensure that the appropriate namespace declarations are included as well. A full description of partial and full escaping has been specified as part of the emerging SQL XML standard.

Example 10-4 XMLElement(): Using Namespaces to Create a Schema-Based XML Document

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

```
SELECT XMLELEMENT ( "Employee",
  XMLATTRIBUTES ( 'http://www.w3.org/2001/XMLSchema' AS "xmlns:xsi",
    'http://www.oracle.com/Employee.xsd' AS
      "xsi:nonamespaceSchemaLocation" ),
  XMLForest(empno, ename, sal)) AS "result"
  FROM scott.emp
  WHERE deptno = 100;
```

This creates an XML document that conforms to the Employee.xsd XMLSchema, result:

```
-----
```

```

<Employee xmlns:xsi="http://www.w3.org/2001/XMLSchema"
          xsi:nonamespaceSchemaLocation="http://www.oracle.com/Employee.xsd">
  <EMPNO>1769</EMPNO>
  <ENAME>John</ENAME>
  <SAL>200000</SAL>
</Employee>

```

Example 10-5 XMLElement(): Generating an Element from a UDT

Using the same example as given in the following DBMS_XMLGEN section ([Example 10-18, "DBMS_XMLGEN: Generating Complex XML"](#) on page 10-29), you can generate a hierarchical XML for the employee, department example as follows:

```

SELECT XMLElement("Department",
  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;

```

This produces an XML document which contains the `Department` element and the canonical mapping of the `dept_t` type.

```

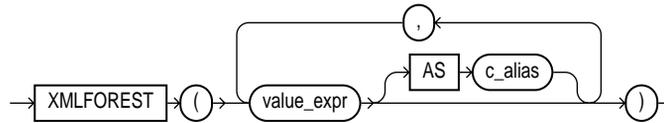
<Department>
  <DEPT_T DEPTNO="100">
    <DNAME>Sports</DNAME>
    <EMPLIST>
      <EMP_T EMPNO="200">
        <ENAME>John</ENAME>
      <EMP_T>
      <EMP_T>
        <ENAME>Jack</ENAME>
      </EMP_T>
    </EMPLIST>
  </DEPT_T>
</Department>

```

XMLForest() Function

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

[Figure 10-4](#) describes the `XMLForest()` syntax.

Figure 10–4 XMLForest() Syntax

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

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

Example 10–6 XMLForest(): Generating Elements for Each Employee with Name Attribute, Start Date, and Dept as Content

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

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

This query might produce the following XML result:

```
<Emp name="John Smith">
  <HIRE>2000-05-24</HIRE>
  <department>Accounting</department>
</Emp>
<Emp name="Mary Martin">
  <HIRE>1996-02-01</HIRE>
  <department>Shipping</department>
</Emp>
```

Example 10–7 XMLForest(): Generating an Element from an UDT

You can also use XMLForest() to generate XML from user-defined types (UDTs). Using the same example as given in the following DBMS_XMLGEN section ([Example 10–18, "DBMS_XMLGEN: Generating Complex XML"](#) on page 10-29), you can generate a hierarchical XML for the employee, department example as follows:

```
SELECT XMLForest(
```

```

dept_t(deptno,dname,
      CAST(MULTISET(
        select empno, ename
        from emp e
        where e.deptno = d.deptno) AS emplist_t)) AS "Department")
AS deptxml
FROM dept d;

```

This produces an XML document which contains the Department element and the canonical mapping of the *dept_t* type.

Note: Unlike in the `XMLElement()` case, the `DEPT_T` element is missing.

```

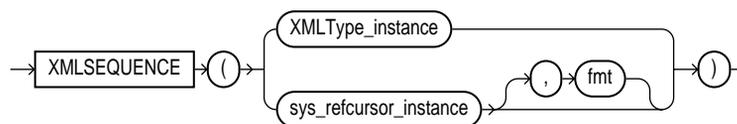
<Department DEPTNO="100">
  <DNAME>Sports</DNAME>
  <EMPLIST>
    <EMP_T EMPNO="200">
      <ENAME>John</ENAME>
    </EMP_T>
    <EMP_T>
      <ENAME>Jack</ENAME>
    </EMP_T>
  </EMPLIST>
</Department>

```

XMLSEQUENCE() Function

`XMLSequence()` function returns a sequence of `XMLType`. The function returns an `XMLSequenceType` which is a `VARRAY` of `XMLType` instances. Since this function returns a collection, it can be used in the `FROM` clause of SQL queries. See [Figure 10-5](#).

Figure 10-5 *XMLSequence()* Syntax



The `XMLSequence()` function has two forms

- The first form inputs an `XMLType` instance and returns a `VARRAY` of top-level nodes. This form can be used to shred XML fragments into multiple rows.
- The second form takes as input a `REFCURSOR` argument, with an optional instance of the `XMLFormat` object and returns the varray of `XMLTypes` corresponding to each row of the cursor. This form can be used to construct `XMLType` instances from arbitrary SQL queries. Note that in this release, this use of `XMLFormat` does not support XML schemas.

`XMLSequence()` is essential for effective SQL queries involving `XMLTypes`.

Example 10–8 XMLSequence(): Generating One XML Document from Another

Suppose you had the following XML document containing employee information:

```
<EMPLOYEES>
  <EMP>
    <EMPNO>112</EMPNO>
    <EMPNAME>Joe</EMPNAME>
    <SALARY>50000</SALARY>
  </EMP>
  <EMP>
    <EMPNO>217</EMPNO>
    <EMPNAME>Jane</EMPNAME>
    <SALARY>60000</SALARY>
  </EMP>
  <EMP>
    <EMPNO>412</EMPNO>7
    <EMPNAME>Jack</EMPNAME>
    <SALARY>40000</SALARY>
  </EMP>
</EMPLOYEES>
```

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

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

This returns the following XML document:

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

```

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

```

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

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

Example 10–9 XMLSequence(): Generating An XML Document for Each Row of a Cursor Expression, Using SYS_REFCURSOR Argument

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

```

SELECT value(e).getClobVal()
FROM TABLE(XMLSequence(Cursor(SELECT * FROM emp))) e;

```

might return the following XML:

```

XMLType
-----
<ROW>
  <EMPNO>300</EMPNO>
  <ENAME>John</ENAME>
</ROW>

<ROW>
  <EMPNO>413</EMPNO>
  <ENAME>Jane</ENAME>
</ROW>

<ROW>

```

```

    <EMPNO>968</EMPNO>
    <ENAME>Jack</ENAME>
  </ROW>
  ...

```

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

Example 10–10 XMLSequence(): Unnesting Collections inside XML Documents into SQL Rows

`XMLSequence()` being a `TABLE` function, can be used to unnest the elements inside an XML document. If you have a XML documents such as:

```

<Department deptno="100">
  <DeptName>Sports</DeptName>
  <EmployeeList>
    <Employee empno="200">
      <Ename>John</Ename>
      <Salary>33333</Salary>
    </Employee>
    <Employee empno="300">
      <Ename>Jack</Ename>
      <Salary>333444</Salary>
    </Employee>
  </EmployeeList>
</Department>

```

```

<Department deptno="200">
  <DeptName>Garment</DeptName>
  <EmployeeList>
    <Employee empno="400">
      <Ename>Marlin</Ename>
      <Salary>20000</Salary>
    </Employee>
  </EmployeeList>
</Department>

```

stored in an `XMLType` table `dept_xml_tab`, you can use the `XMLSequence()` function to unnest the Employee list items as top level SQL rows:

```

CREATE TABLE dept_xml_tab OF XMLTYPE;

INSERT INTO dept_xml_tab VALUES(
  xmltype(' <Department deptno="100">
    <DeptName>Sports</DeptName><EmployeeList>
    <Employee empno="200"><Ename>John</Ename><Salary>33333</Salary></Employee>

```

```

<Employee empno="300"><Ename>Jack</Ename><Salary>333444</Salary></Employee>
</EmployeeList></Department>');

INSERT INTO dept_xml_tab VALUES (
  xmltype('<Department deptno="200">
    <DeptName>Sports</DeptName><EmployeeList>
    <Employee empno="400"><Ename>Marlin</Ename><Salary>20000</Salary></Employee>
    </EmployeeList></Department>'));

SELECT extractvalue(value(d), '/Department/@deptno') as deptno,
       extractvalue(value(e), '/Employee/@empno') as empno,
       extractvalue(value(e), '/Employee/Ename') as ename
FROM   dept_xml_tab d,
       TABLE(XMLSequence(extract(value(d), '/Department/EmployeeList/Employee'))) e;

```

This returns the following:

DEPTNO	EMPNO	ENAME
100	200	John
100	300	Jack
200	400	Marlin

3 rows selected

For each row in table `dept_xml_tab`, the `TABLE` function is evaluated. Here, the `extract()` function creates a new `XMLType` instance that contains a fragment of all employee elements. This is fed to the `XMLSequence()` which creates a collection of all employees.

The `TABLE` function then explodes the collection elements into multiple rows which are correlated with the parent table `dept_xml_tab`. Thus you get a list of all the parent `dept_xml_tab` rows with the associated employees.

The `extractValue()` functions extract out the scalar values for the department number, employee number, and name.

XMLConcat() Function

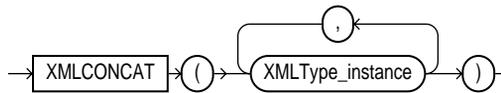
`XMLConcat()` function concatenates all the arguments passed in to create a `XML` fragment. [Figure 10-6](#) shows the `XMLConcat()` syntax. `XMLConcat()` has two forms:

- The first form takes an `XMLSequenceType`, which is a `VARRAY` of `XMLType` and returns a single `XMLType` instance that is the concatenation of all of the

elements of the varray. This form is useful to collapse lists of XMLTypes into a single instance.

- The second form takes an arbitrary number of XMLType values and concatenates them together. If one of the value is null, it is ignored in the result. If all the values are NULL, the result is NULL. This form is used to concatenate arbitrary number of XMLType instances in the same row. XMLAgg () can be used to concatenate XMLType instances across rows.

Figure 10–6 XMLConcat() Syntax



Example 10–11 XMLConcat(): Returning a Concatenation of XML Elements Used in the Argument Sequence

This example shows how XMLConcat () returns the concatenation of XMLTypes from the XMLSequenceType:

```

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

returns a single fragment of the form:

```

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

Example 10–12 XMLConcat(): Returning XML Elements By Concatenating the Elements in the Arguments

The following example creates an XML element for the first and the last names and then concatenates the result:

```

SELECT XMLConcat ( XMLElement ("first", e.fname), XMLElement ("last", e.lname))
AS "result"
FROM employees e ;
  
```

This query might produce the following XML document:

```

<first>Mary</first>
<last>Martin</last>

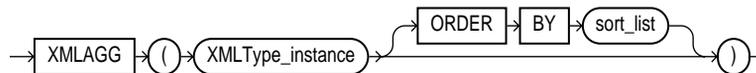
<first>John</first>
<last>Smith</last>

```

XMLAgg() Function

`XMLAgg()` function is an aggregate function that produces a forest of XML elements from a collection of XML elements. [Figure 10–7](#) describes the `XMLAgg()` syntax. As with `XMLConcat()`, any arguments that are null are dropped from the result. `XMLAgg()` function is similar to the `SYS_XMLAGG()` function except that it returns a forest of nodes, and does not take the `XMLFormat()` parameter. This function can be used to concatenate `XMLType` instances across multiple rows.

Figure 10–7 *XMLAgg() Syntax*



Example 10–13 *XMLAgg(): Generating Department Elements with a List of Employee Elements*

This example produces department elements, with the list of employees belonging to that department:

```

SELECT XMLELEMENT( "Department", XMLATTRIBUTES ( e.dept AS "name" ),
                 XMLAGG (XMLELEMENT ("emp", e.lname))) AS "dept_list"
FROM employees e
GROUP BY dept ;

```

This query might produce the following XML result:

```

<Department name="Accounting">
  <emp>Yates</emp>
  <emp>Smith</emp>
</Department>

<Department name="Shipping">
  <emp>Oppenheimer</emp>
  <emp>Martin</emp>
</Department>

```

Example 10–14 XMLAgg(): Generating Department Elements, Employee Elements Per Department, and Employee Dependents

XMLAgg() can be used to reflect the hierarchical nature of some relationships that exist in tables. The following example generates a department element for each department. Within this it creates elements for all employees of the department. Within each employee, it lists their dependents:

```
SELECT XMLELEMENT( "Department", XMLATTRIBUTES ( d.dname AS "name" ),
    (SELECT XMLAGG(XMLELEMENT ("emp", XMLATTRIBUTES (e.ename AS name),
        ( SELECT XMLAGG(XMLELEMENT( "dependent",
            XMLATTRIBUTES(de.name AS "name")))
        FROM dependents de
        WHERE de.empno = e.empno ) ))
    FROM emp e
    WHERE e.deptno = d.deptno ) AS "dept_list"
FROM dept d ;
```

The query might produce a row containing the XMLType instance for each department.

```
<Department name="Accounting">
  <emp name="Smith">
    <dependent name="Sara Smith"/d>
    <dependent name="Joyce Smith"/>
  </emp>
  <emp name="Yates"/>
</Department>

<Department name="Shipping">
  <emp name="Martin">
    <dependent name="Alan Martin"/>
  </emp>
  <emp name="Oppenheimer">
    <dependent name="Ellen Oppenheimer"/>
  </emp>
</Department>
```

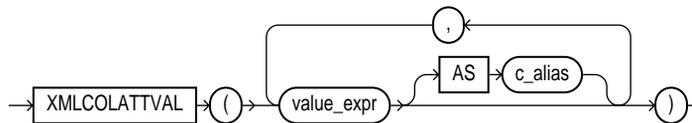
Generating XML from the Database Using SQLX Functions

XMLColAttVal() is an Oracle SQLX extension function.

XMLColAttVal() Function

XMLColAttVal() function generates a forest of XML column elements containing the value of the arguments passed in. Figure 10–8 shows the XMLColAttVal() syntax.

Figure 10–8 XMLColAttVal() Syntax



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

Example 10–15 XMLColAttVal(): Generating an Emp Element Per Employee with Name Attribute and Elements with Start Date and Dept as Content

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

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

This query might produce the following XML result:

```
<Emp name="John Smith">
  <column name="HIRE">2000-05-24</column>
  <column name="department">Accounting</column>
</Emp>
<Emp name="Mary Martin">
  <column name="HIRE">1996-02-01</column>
  <column name="department">Shipping</column>
</Emp>
<Emp name="Samantha Stevens">
```

```
<column name="HIRE">1992-11-15</column>
<column name="department">Standards</column>
</Emp>
```

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

Generating XML from Oracle9i Database Using DBMS_XMLGEN

`DBMS_XMLGEN` creates XML documents from any SQL query by mapping the database query results into XML. It gets the XML document as a CLOB or `XMLType`. It provides a “fetch” interface whereby you can specify the maximum rows and rows to skip. This is useful for pagination requirements in Web applications. `DBMS_XMLGEN` also provides options for changing tag names for `ROW`, `ROWSET`, and so on.

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

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

See Also: ["Generating XML with XSU's OracleXMLQuery"](#), in [Chapter 7, "XML SQL Utility \(XSU\)"](#), and compare the functionality of `OracleXMLQuery` with `DBMS_XMLGEN`.

Sample DBMS_XMLGEN Query Result

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

```
<?xml version="1.0"?>
<ROWSET>
<ROW>
  <EMPNO>30</EMPNO>
  <ENAME>Scott</ENAME>
  <SALARY>20000</SALARY>
</ROW>
<ROW>
```

```
<EMPNO>30</EMPNO>  
<ENAME>Mary</ENAME>  
<AGE>40</AGE>  
</ROW>  
</ROWSET>
```

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

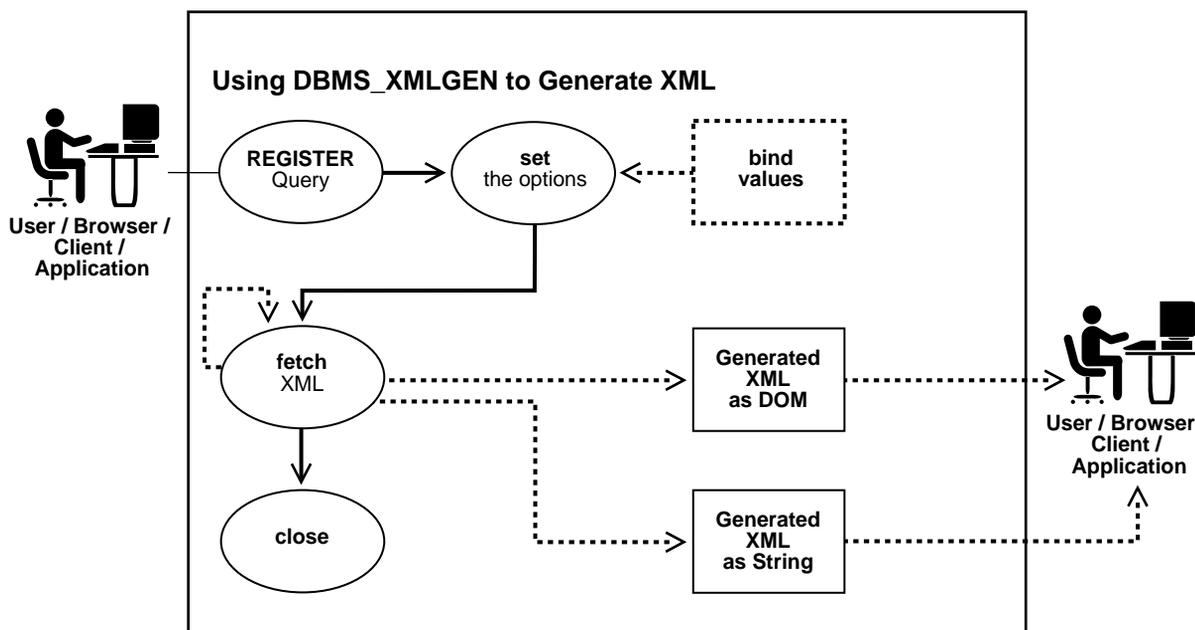
- Every row of the query result maps to an XML element with the default tag name `ROW`.
- The entire result is enclosed in a `ROWSET` element. These names are both configurable, using the `setRowTagName()` and `setRowSetTagName()` procedures in `DBMS_XMLGEN`.
- Each column in the SQL query result, maps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

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

DBMS_XMLGEN Calling Sequence

[Figure 10-9](#) summarizes the `DBMS_XMLGEN` calling sequence.

Figure 10–9 DBMS_XMLGEN Calling Sequence



Here is DBMS_XMLGEN's calling sequence:

1. Get the context from the package by supplying a SQL query and calling the `newContext()` call.
2. Pass the context to all the procedures/functions in the package to set the various options. For example to set the ROW element's name, use `setRowTag(ctx)`, where `ctx` is the context got from the previous `newContext()` call.
3. Get the XML result, using the `getXML()` or `getXMLType()`. By setting the maximum rows to be retrieved for each fetch using the `setMaxRows()` call, you can call this function repeatedly, getting the maximum number of row set for each call. The function returns null if there are no rows left in the query.

`getXML()` and `getXMLType()` always return an XML document, even if there were no rows to retrieve. If you want to know if there were any rows retrieved, use the function `getNumRowsProcessed()`.

4. You can reset the query to start again and repeat step 3.

5. Close the `closeContext()` to free up any resource allocated inside.

Table 10-1 summarizes DBMS_XMLGEN functions and procedures.

Table 10-1 DBMS_XMLGEN Functions and Procedures

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

Table 10–1 DBMS_XMLGEN Functions and Procedures (Cont.)

Function or Procedure	Description
<p>PROCEDURE setRowSetTag(ctx IN ctxHandle, rowSetTag IN VARCHAR2);</p>	<p>PARAMETERS: ctx (IN) - the context handle obtained from the newContext call, rowsetTag (IN) - the name of the document element. NULL indicates that you do not want the ROW element to be present. Call this to set the name of the document root element, if you do not want the default "ROWSET" name in the output. You can also set this to NULL to suppress the printing of this element. However, this is an error if both the row and the rowset are null and there is more than one column or row in the output.</p>
getXML()	<p>Gets the XML document by fetching the maximum number of rows specified. It appends the XML document to the CLOB passed in.</p>
<p>PROCEDURE getXML(ctx IN ctxHandle, clobval IN OUT NCOPY clob, dtdOrSchema IN number:= NONE);</p>	<p>PARAMETERS: ctx (IN) - The context handle obtained from the newContext() call, clobval (IN/OUT) - the clob to which the XML document is to be appended, dtdOrSchema (IN) - whether you should generate the DTD or Schema. This parameter is NOT supported. Use this version of the getXML function, to avoid any extra CLOB copies and if you want to reuse the same CLOB for subsequent calls. This getXML call is more efficient than the next flavor, though this involves that you create the lob locator. When generating the XML, the number of rows indicated by the setSkipRows call are skipped, then the maximum number of rows as specified by the setMaxRows call (or the entire result if not specified) is fetched and converted to XML. Use the getNumRowsProcessed function to check if any rows were retrieved or not.</p>
getXML()	<p>Generates the XML document and returns it as a CLOB.</p>
<p>FUNCTION getXML(ctx IN ctxHandle, dtdOrSchema IN number:= NONE) RETURN clob</p>	<p>PARAMETERS: ctx (IN) - The context handle obtained from the newContext() call, dtdOrSchema (IN) - whether we should generate the DTD or Schema. This parameter is NOT supported. RETURNS: A temporary CLOB containing the document. Free the temporary CLOB obtained from this function using the dbms_lob.freetemporary call.</p>
<p>FUNCTION getXMLType(ctx IN ctxHandle, dtdOrSchema IN number:= NONE) RETURN XMLTYPE</p>	<p>PARAMETERS: ctx (IN) - The context handle obtained from the newContext() call, dtdOrSchema (IN) - whether we should generate the DTD or Schema. This parameter is NOT supported. RETURNS: An XMLType instance containing the document.</p>

Table 10–1 DBMS_XMLGEN Functions and Procedures (Cont.)

Function or Procedure	Description
FUNCTION getXML(sqlQuery IN VARCHAR2, dtdOrSchema IN NUMBER := NONE) RETURN CLOB;	Converts the query results from the passed in SQL query string to XML format, and returns the XML as a CLOB.
FUNCTION getXMLType(sqlQuery IN VARCHAR2, dtdOrSchema IN NUMBER := NONE) RETURN XMLTYPE;	Converts the query results from the passed in SQL query string to XML format, and returns the XML as a CLOB.
getNumRowsProcessed()	Gets the number of SQL rows processed when generating the XML using the <code>getXML</code> call. This count does not include the number of rows skipped before generating the XML.
FUNCTION getNumRowsProcessed(ctx IN ctxHandle) RETURN number	PARAMETERS: <code>queryString</code> (IN) - the query string, the result of which needs to be converted to XML RETURNS: This gets the number of SQL rows that were processed in the last call to <code>getXML</code> . You can call this to find out if the end of the result set has been reached. This does not include the number of rows skipped. Use this function to determine the terminating condition if you are calling <code>getXML</code> in a loop. Note that <code>getXML</code> would always generate a 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 <code>getXML</code> call.
PROCEDURE setMaxRows(ctx IN ctxHandle, maxRows IN NUMBER);	PARAMETERS: <code>ctx</code> (IN) - the context handle corresponding to the query executed, <code>maxRows</code> (IN) - the maximum number of rows to get for each call to <code>getXML</code> . The <code>maxRows</code> 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.
setSkipRows()	Skips a given number of rows before generating the XML output for every call to the <code>getXML</code> routine.
PROCEDURE setSkipRows(ctx IN ctxHandle, skipRows IN NUMBER);	PARAMETERS: <code>ctx</code> (IN) - the context handle corresponding to the query executed, <code>skipRows</code> (IN) - the number of rows to skip for each call to <code>getXML</code> . The <code>skipRows</code> 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 <code>skipRows</code> to zero. For the next set, you can set the <code>skipRows</code> to the number of rows that you got in the first case.

Table 10–1 DBMS_XMLGEN Functions and Procedures (Cont.)

Function or Procedure	Description
setConvertSpecialChars()	Sets whether special characters in the XML data need to be converted into their escaped XML equivalent or not. For example, the "<" sign is converted to <. The default is to perform conversions.
PROCEDURE setConvertSpecialChars(ctx IN ctxHandle, conv IN boolean);	PARAMETERS: ctx (IN) - the context handle to use, conv (IN) - true indicates that conversion is needed. You can use this function to speed up the XML processing whenever you are sure that the input data cannot contain any special characters such as <, >, ", ', and so on, which need to be escaped. Note that it is expensive to actually scan the character data to replace the special characters, particularly if it involves a lot of data. So in cases when the data is XML-safe, then this function can be called to improve performance.
useItemTagsForColl()	Sets the name of the collection elements. The default name for collection elements is the type name itself. You can override that to use the name of the column with the <code>_ITEM</code> tag appended to it using this function.
PROCEDURE useItemTagsForColl(ctx IN ctxHandle);	PARAMETERS: ctx (IN) - the context handle. If you have a collection of NUMBER, say, the default tag name for the collection elements is NUMBER. You can override this behavior and generate the collection column name with the <code>_ITEM</code> tag appended to it, by calling this procedure.
restartQuery()	Restarts the query and generate the XML from the first row again.
PROCEDURE restartQuery(ctx IN ctxHandle);	PARAMETERS: 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.
PROCEDURE closeContext(ctx IN ctxHandle);	PARAMETERS: ctx (IN) - the context handle to close. Closes all resources associated with this handle. After this you cannot use the handle for any other <code>DBMS_XMLGEN</code> function call.
Conversion Functions	
FUNCTION convert(xmlData IN varchar2, flag IN NUMBER := ENTITY_ENCODE) return varchar2;	Encodes or decodes the passed in XML data string. <ul style="list-style-type: none"> ■ Encoding refers to replacing entity references such as '<' to their escaped equivalent, such as '&lt;'. ■ Decoding refers to the reverse conversion.
FUNCTION convert(xmlData IN CLOB, flag IN NUMBER := ENTITY_ENCODE) return CLOB;	Encodes or decodes the passed in XML CLOB data. <ul style="list-style-type: none"> ■ Encoding refers to replacing entity references such as '<' to their escaped equivalent, such as '&lt;'. ■ Decoding refers to the reverse conversion.

Example 10–16 DBMS_XMLGEN: Generating Simple XML

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

```
CREATE TABLE temp_clob_tab(result CLOB);

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

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

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

  INSERT INTO temp_clob_tab VALUES(result);

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

This query example generates the following XML:

```
SELECT * FROM temp_clob_tab;
```

RESULT

```
-----
<?xml version='1.0'?>
<ROWSET>
  <EMPLOYEE>
    <EMPNO>7369</EMPNO>
    <ENAME>SMITH</ENAME>
    <JOB>CLERK</JOB>
    <MGR>7902</MGR>
    <HIREDATE>17-DEC-80</HIREDATE>
    <SAL>800</SAL>
    <DEPTNO>20</DEPTNO>
  </EMPLOYEE>
  <EMPLOYEE>
    <EMPNO>7499</EMPNO>
    <ENAME>ALLEN</ENAME>
```

```

    <JOB>SALESMAN</JOB>
    <MGR>7698</MGR>
    <HIREDATE>20-FEB-81</HIREDATE>
    <SAL>1600</SAL>
    <COMM>300</COMM>
    <DEPTNO>30</DEPTNO>
  </EMPLOYEE>
  ...
</ROWSET>

```

Example 10-17 DBMS_XMLGEN: Generating Simple XML with Pagination

Instead of generating all the XML for all rows, you can use the `fetch` interface that `DBMS_XMLGEN` provides to retrieve a fixed number of rows each time. This speeds up response time and also can help in scaling applications that need a DOM API on the resulting XML, particularly if the number of rows is large.

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

```

-- create a table to hold the results..!
CREATE TABLE temp_clob_tab ( result clob);

declare
  qryCtx dbms_xmlgen.ctxHandle;
  result CLOB;
begin

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

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

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

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

    -- do some processing with the lob data..!
    -- Here, we are inserting the results
    -- into a table. You can print the lob out, output it to a stream,
    -- put it in a queue
    -- or do any other processing.

```

```

        insert into temp_clob_tab values(result);

    end loop;
    --close context
    dbms_xmlgen.closeContext(qryCtx);
end;
/

```

Here, for each set of 5 rows, you generate an XML document.

Example 10–18 DBMS_XMLGEN: Generating Complex XML

Complex XML can be generated using object types to represent nested structures:

```

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

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

CREATE TYPE emp_t AS OBJECT(
    "@employee_id" NUMBER,
    last_name VARCHAR2(20)
);
/

CREATE TYPE emplist_t AS TABLE OF emp_t;
/

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

qryCtx := dbms_xmlgen.newContext
('SELECT dept_t(department_id, department_name,
    CAST(MULTISET
        (SELECT e.employee_id, e.last_name

```

```

                FROM new_employees e
                WHERE e.department_id = d.department_id)
                AS emplist_t)) AS deptxml
FROM new_departments d');
DBMS_XMLGEN.setRowTag(qryCtx, NULL);

-- Here is the resulting XML:
-- <ROWSET>
--   <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>
--   <DEPTXML DEPARTMENT_ID="20">
--     ...
-- </ROWSET>

```

Now, you can select the LOB data from the `temp_clob_Tab` table and verify the results. The result looks like the sample result shown in the previous section, ["Sample DBMS_XMLGEN Query Result"](#) on page 10-20.

With relational data, the results are a flat non-nested XML document. To obtain *nested* XML structures, you can use object-relational data, where the mapping is as follows:

- *Object types* map as an XML element -- see [Chapter 5, "Structured Mapping of XMLType"](#).
- *Attributes of the type*, map to sub-elements of the parent element

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

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

Example 10–19 DBMS_XMLGEN: Generating Complex XML #2 - Inputting User Defined Types For Nested XML Documents

When you input a user-defined type (UDT) value to DBMS_XMLGEN functions, the user-defined type is mapped to an XML document using canonical mapping. In the canonical mapping, user-defined type's *attributes* are mapped to XML *elements*. Attributes with names starting with "@" are mapped to attributes of the preceding element.

User-defined types can be used for nesting in the resulting XML document. For example, consider tables, EMP and DEPT:

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

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

To generate a hierarchical view of the data, that is, departments with employees in them, you can define suitable object types to create the structure inside the database as follows:

```
CREATE TYPE EMP_T AS OBJECT
(
  "@empno" number, -- empno defined as an attribute!
  ename varchar2(20)
);
/
-- You have defined the empno with an @ sign in front, to denote that it must
-- be mapped as an attribute of the enclosing Employee element.
```

```

CREATE TYPE EMPLIST_T AS TABLE OF EMP_T;
/
CREATE TYPE DEPT_T AS OBJECT
(
  "@deptno" number,
  dname varchar2(20),
  emplist emplist_t
);
/

-- Department type, DEPT_T, denotes the department as containing a list of
-- employees. You can now query the employee and department tables and get
-- the result as an XML document, as follows:
declare
  qryCtx dbms_xmlgen.ctxHandle;
  result CLOB;
begin

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

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

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

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

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

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

```

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

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

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

Example 10–20 DBMS_XMLGEN: Generating a Purchase Order from the Database in XML Format

This example uses `DBMS_XMLGEN.getXMLType()` to generate `PurchaseOrder` in XML format from a relational database using object views. Note that the example is five pages long.

```
-- Create relational schema and define Object Views
-- Note: DBMS_XMLGEN Package maps UDT attributes names
--       starting with '@' to xml attributes
-----
-- Purchase Order Object View Model

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

-- Address object type
CREATE TYPE Address_typ AS OBJECT (
```

```

        Street          VARCHAR2(200),
        City            VARCHAR2(200),
        State           CHAR(2),
        Zip             VARCHAR2(20)
    )
/

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

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

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

-- LineItems Nested table
CREATE TYPE LineItems_ntabtyp AS TABLE OF LineItem_typ
/

-- Purchase Order object type
CREATE TYPE PO_typ AUTHID CURRENT_USER AS OBJECT (
    PONO               NUMBER,
    Cust_ref            REF Customer_typ,
    OrderDate           DATE,
    ShipDate            TIMESTAMP,
    LineItems_ntab      LineItems_ntabtyp,
    ShipToAddr          Address_typ
)

```

```

/

-- Create Purchase Order Relational Model tables

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

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

--Stock Table
CREATE TABLE Stock_tab (
  StockNo        NUMBER constraint stock_uk UNIQUE,
  Price          NUMBER,
  TaxRate        NUMBER
);

--Line Items Table
CREATE TABLE LineItems_tab(
  LineItemNo     NUMBER,
  PONO           NUMBER constraint LI_PO_FK REFERENCES po_tab,
  StockNo        NUMBER ,

```

```

Quantity          NUMBER,
Discount          NUMBER,
constraint LI_PK PRIMARY KEY (PONo, LineItemNo)
);

-- create Object Views

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

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

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

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

INSERT INTO Stock_tab VALUES(1004, 6750.00, 2) ;

```

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

-- Register Customers

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

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

-- Place Orders

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

INSERT INTO PO_tab
  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 DBMS_XMLGEN Package to generate PO in XML format
-- and store XMLTYPE in po_xml table
-----

declare
  qryCtx dbms_xmlgen.ctxHandle;
  pxml XMLTYPE;
  cxml clob;
begin
```

```

-- get the query context;
qryCtx := dbms_xmlgen.newContext('
            select pono,deref(cust_ref) customer,p.OrderDate,p.shipdate,
            lineitems_ntab lineitems,shiptoaddr
            from po p'
        );

-- set the maximum number of rows to be 1,
dbms_xmlgen.setMaxRows(qryCtx, 1);
-- set rowset tag to null and row tag to PurchaseOrder
dbms_xmlgen.setRowSetTag(qryCtx,null);
dbms_xmlgen.setRowTag(qryCtx,'PurchaseOrder');

loop
    -- now get the po in xml format
    pxml := dbms_xmlgen.getXMLType(qryCtx);

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

    -- Store XMLTYPE po in po_xml table (get the pono out)
    insert into po_xml_tab (poid, poDoc)
        values(
            pxml.extract('//PONO/text()').getNumberVal(),
            pxml);
end loop;
end;
/

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

set long 100000
set pages 100
select x.podoc.getClobVal() xpo
from   po_xml_tab x;

```

This produces the following purchase order XML documents:

PurchaseOrder 1001:

```

<?xml version="1.0"?>
<PurchaseOrder>
  <PONO>1001</PONO>
  <CUSTOMER>

```

```

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

```

PurchaseOrder 2001:

```

<?xml version="1.0"?>
<PurchaseOrder>
  <PONO>2001</PONO>
  <CUSTOMER>
    <CUSTNO>2</CUSTNO>
    <CUSTNAME>John Nike</CUSTNAME>
    <ADDRESS>
      <STREET>323 College Drive</STREET>

```

```

    <CITY>Edison</CITY>
    <STATE>NJ</STATE>
    <ZIP>08820</ZIP>
  </ADDRESS>
  <PHONELIST>
    <VARCHAR2>609-555-1212</VARCHAR2>
    <VARCHAR2>201-555-1212</VARCHAR2>
  </PHONELIST>
</CUSTOMER>
<ORDERDATE>20-APR-97</ORDERDATE>
<SHIPDATE>20-MAY-97 12.00.00.000000 AM</SHIPDATE>
<LINEITEMS>
  <LINEITEM_TYP LineItemNo="1">
    <ITEM StockNo="1004">
      <PRICE>6750</PRICE>
      <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>1</QUANTITY>
    <DISCOUNT>0</DISCOUNT>
  </LINEITEM_TYP>
  <LINEITEM_TYP LineItemNo="2">
    <ITEM StockNo="1011">
      <PRICE>4500.23</PRICE>
      <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>2</QUANTITY>
    <DISCOUNT>1</DISCOUNT>
  </LINEITEM_TYP>
</LINEITEMS>
<SHIPTOADDR>
  <STREET>55 Madison Ave</STREET>
  <CITY>Madison</CITY>
  <STATE>WI</STATE>
  <ZIP>53715</ZIP>
</SHIPTOADDR>
</PurchaseOrder>

```

Example 10–21 DBMS_XMLGEN: Generating a New Context Handle from a Passed in PL/SQL Ref Cursor

```

CREATE OR REPLACE FUNCTION joe3 RETURN CLOB
IS
  ctx1 number := 2;
  ctx2 number;
  xmldoc CLOB;

```

```

page NUMBER := 0;
xmlpage boolean := true;
refcur SYS_REFCURSOR;
BEGIN
  OPEN refcur FOR 'select * from emp where rownum < :1' USING ctx1;

  ctx2 := DBMS_XMLGEN.newContext( refcur);

  ctx1 := 4;
  OPEN refcur FOR 'select * from emp where rownum < :1' USING ctx1;
  ctx1 := 5;
  OPEN refcur FOR 'select * from emp where rownum < :1' USING ctx1;
  dbms_lob.createtemporary(xmldoc, TRUE);
  -- xmlDoc will have 4 rows
  xmlDoc := DBMS_XMLGEN.getXML(ctx2,DBMS_XMLGEN.NONE);
  DBMS_XMLGEN.closeContext(ctx2);
  return xmlDoc;
END;
/

```

Generating XML Using Oracle-Provided SQL Functions

In addition to the SQL standard functions, Oracle9i provides the `SYS_XMLGEN` and `SYS_XMLAGG` functions to aid in generating XML.

SYS_XMLGEN() Function

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

Example 10–22 Using SQL_XMLGEN to Create XML

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

```

SELECT SYS_XMLGEN(employee_id)
       FROM employees WHERE last_name LIKE 'Scott%';

```

The resulting XML document is:

```

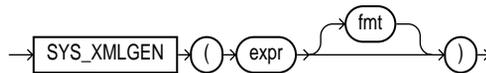
<?xml version='1.0'?>
<employee_id>60</employee_id>

```

SYS_XMLGEN Syntax

`SYS_XMLGEN()` takes in a scalar value, object type, or `XMLType` instance to be converted to an XML document. It also takes an optional `XMLFormat` (the old name was `XMLGenFormatType`) object that you can use to specify formatting options for the resulting XML document. See [Figure 10-10](#).

Figure 10-10 *SYS_XMLGEN Syntax*



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

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

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

In this release, the formatting argument for `SYS_XMLGEN()` accepts the schema and element name, and generates the XML document conforming to that registered schema.

```

SELECT sys_xmlgen(
    dept_t(d.deptno, d.dname, d.loc,
        cast(multiset(
            SELECT emp_t(e.empno, e.ename, e.job, e.mgr, e.hiredate,e.sal, e.comm)
            FROM emp e
            WHERE e.deptno = d.deptno) AS emp_list_t),
    xmlformat.createformat('Department', 'http://www.oracle.com/dept.xsd'))
FROM dept d;
  
```

Example 10–23 SYS_XMLGEN(): Retrieving Employee Email ID From Employees Table and Generating XML with EMAIL Element

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

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

```
SYS_XMLGEN(EMAIL).GETSTRINGVAL()
```

```
-----
<EMAIL>SHIGGENS</EMAIL>
```

Why is SYS_XMLGEN() so Powerful?

`SYS_XMLGEN()` is powerful for the following reasons:

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

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

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

and returns an `XMLType` instance contained in the document.

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

Using XMLFormat Object Type

You can use `XMLFormat` to specify formatting arguments for `SYS_XMLGEN()` and `SYS_XMLAGG()` functions.

`SYS_XMLGEN()` returns an instance of type `XMLType` containing an XML document. Oracle9i provides the `XMLFormat` object, which lets you format the output of the `SYS_XMLGEN` function.

[Table 10-2](#) lists the `XMLFormat` attributes, of the `XMLFormat` object. The function that implements this type follows the table.

Table 10-2 Attributes of the XMLFormat Object

Attribute	Datatype	Purpose
<code>enclTag</code>	<code>VARCHAR2(100)</code>	The name of the enclosing tag for the result of the <code>SYS_XMLGEN</code> function. If the input to the function is a column name, the default is the column name. Otherwise the default is <code>ROW</code> . When <code>schemaType</code> is set to <code>USE_GIVEN_SCHEMA</code> , this attribute also gives the name of the <code>XMLSchema</code> element.
<code>schemaType</code>	<code>VARCHAR2(100)</code>	The type of schema generation for the output document. Valid values are <code>'NO_SCHEMA'</code> and <code>'USE_GIVEN_SCHEMA'</code> . The default is <code>'NO_SCHEMA'</code> .
<code>schemaName</code>	<code>VARCHAR2(4000)</code>	The name of the target schema Oracle uses if the value of the <code>schemaType</code> is <code>'USE_GIVEN_SCHEMA'</code> . If you specify <code>schemaName</code> , then Oracle uses the enclosing tag as the element name.
<code>targetNameSpace</code>	<code>VARCHAR2(4000)</code>	The target namespace if the schema is specified (that is, <code>schemaType</code> is <code>GEN_SCHEMA_*</code> , or <code>USE_GIVEN_SCHEMA</code>)
<code>dburl</code>	<code>VARCHAR2(2000)</code>	The URL to the database to use if <code>WITH_SCHEMA</code> is specified. If this attribute is not specified, the Oracle declares the URL to the types as a relative URL reference.
<code>processingIns</code>	<code>VARCHAR2(4000)</code>	User-provided processing instructions, which are appended to the top of the function output before the element.

Example 10-24 Creating a Formatting Object with `createFormat`

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

```

STATIC FUNCTION createFormat(
    enclTag IN varchar2 := 'ROWSET',
    schemaType IN varchar2 := 'NO_SCHEMA',
    schemaName IN varchar2 := null,
    targetNameSpace IN varchar2 := null,
    dburlPrefix IN varchar2 := null,
    processingIns IN varchar2 := null) RETURN XMLGenFormatType,
MEMBER PROCEDURE genSchema (spec IN varchar2),
MEMBER PROCEDURE setSchemaName(schemaName IN varchar2),
MEMBER PROCEDURE setTargetNameSpace(targetNameSpace IN varchar2),
MEMBER PROCEDURE setEnclosingElementName(enclTag IN varchar2),
MEMBER PROCEDURE setDbUrlPrefix(prefix IN varchar2),

```

```

MEMBER PROCEDURE setProcessingIns(pi IN varchar2),
CONSTRUCTOR FUNCTION XMLGenFormatType (
    enclTag IN varchar2 := 'ROWSET',
    schemaType IN varchar2 := 'NO_SCHEMA',
    schemaName IN varchar2 := null,
    targetNameSpace IN varchar2 := null,
    dbUrlPrefix IN varchar2 := null,
    processingIns IN varchar2 := null) RETURN SELF AS RESULT

```

Note: XMLFormat object is the new name for XMLGenFormatType. You can use both names.

Example 10–25 SYS_XMLGEN(): Converting a Scalar Value to an XML Document Element's Contents

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

```
select sys_xmlgen(empno) from scott.emp where rownum < 2;
```

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

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

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

Example 10–26 Generating Default Column Name, ROW

In the last example, you used the column name EMPNO for the document. If the column name cannot be derived directly, then the default name ROW is used. For example, in the following case:

```
SELECT sys_xmlgen(empno).getclobval()
FROM scott.emp
WHERE rownum < 2;
```

you get the following XML output:

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

since the function cannot infer the name of the expression. You can override the default ROW tag by supplying an `XMLFormat` (the old name was "XMLGenFormatType") object to the first argument of the operator.

Example 10–27 Overriding the Default Column Name: Supplying an XMLFormat Object to the Operator's First Argument

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

```
SELECT sys_xmlgen(empno *2,
                 xmlformat.createformat('EMPNO')).getClobVal()
FROM emp;
```

This results in the following XML:

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

Example 10–28 SYS_XMLGEN(): Converting a User-Defined Type to XML

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

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

Using the same example as given in the `DBMS_XMLGEN` section ([Example 10–18, "DBMS_XMLGEN: Generating Complex XML"](#) on page 10-29), you can generate a hierarchical XML for the employee, department example as follows:

```
SELECT SYS_XMLGEN(
  dept_t(deptno,dname,
        CAST(MULTISET(
          select empno, ename
          from emp e
          where e.deptno = d.deptno) AS emplist_t)).getClobVal()
  AS deptxml
FROM dept d;
```

The `MULTISET` operator treats the result of the subset of employees working in the department as a list and the `CAST` around it, cast's it to the appropriate collection

type. You then create a department instance around it and call `SYS_XMLGEN()` to create the XML for the object instance.

The result is:

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

for each row of the department. The default name `ROW` is present because the function cannot deduce the name of the input operand directly.

Note: The difference between `SYS_XMLGEN()` function and `DBMS_XMLGEN` package is apparent from the preceding example:

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

Example 10–29 *SYS_XMLGEN(): Converting an XMLType Instance*

If you pass an XML document into `SYS_XMLGEN()`, `SYS_XMLGEN()` encloses the document (or fragment) with an element, whose tag name is the default `ROW`, or the name passed in through the formatting object. This functionality can be used to turn document fragments into well formed documents.

For example, the `extract()` operation on the following document, can return a fragment. If you extract out the `EMPNO` elements from the following document:

```
<DOCUMENT>
  <EMPLOYEE>
    <ENAME>John</ENAME>
    <EMPNO>200</EMPNO>
  </EMPLOYEE>
</EMPLOYEE>
```

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

using the following statement:

```
SELECT e.podoc.extract('/DOCUMENT/EMPLOYEE/ENAME')
FROM po_xml_tab e;
```

you get an XML document fragment such as the following:

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

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

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

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

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

Note: If the input was a column, then the column name would have been used as default. You can override the enclosing element name using the formatting object that can be passed in as an additional argument to the function. See ["Using XMLFormat Object Type"](#) on page 10-43.

Example 10–30 SYS_XMLGEN(): Using SYS_XMLGEN() with Object Views

```
-- create Purchase Order object type
CREATE OR REPLACE TYPE PO_typ AUTHID CURRENT_USER AS OBJECT (
```

```

PONO                NUMBER,
Customer            Customer_typ,
OrderDate           DATE,
ShipDate            TIMESTAMP,
LineItems_ntab     LineItems_ntabtyp,
ShipToAddr          Address_typ
)
/

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

-----
-- Use SYS_XMLGEN() to generate PO in XML format
-----

set long 20000
set pages 100
SELECT SYS_XMLGEN(value(p),
                 sys.xmlformat.createFormat('PurchaseOrder')).getClobVal() PO
FROM po p
WHERE p.pono=1001;

```

This returns the Purchase Order in XML format:

```

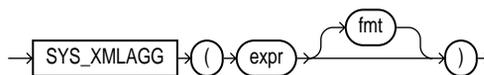
<?xml version="1.0"?>
<PurchaseOrder>
  <PONO>1001</PONO>
  <CUSTOMER>
    <CUSTNO>1</CUSTNO>
    <CUSTNAME>Jean Nance</CUSTNAME>
  </CUSTOMER>
</PurchaseOrder>

```

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

SYS_XMLAGG() Function

`SYS_XMLAGG()` function aggregates all XML documents or fragments represented by `expr` and produces a single XML document. It adds a new enclosing element with a default name, `ROWSET`. To format the XML document differently then specify `fmt`, the instance of `XMLFORMAT` object

Figure 10–11 SYS_XMLAGG() Syntax

See Also: *Oracle9i SQL Reference*

Generating XML Using XSQL Pages Publishing Framework

Oracle9i introduced `XMLType` for use with storing and querying XML-based database content. You can use these database XML features to produce XML for inclusion in your XSQL pages by using the `<xsql:include-xml>` action element.

The `SELECT` statement that appears inside the `<xsql:include-xml>` element should return a single row containing a single column. The column can either be a `CLOB` or a `VARCHAR2` value containing a well-formed XML document. The XML document will be parsed and included in your XSQL page.

Example 10–31 Using XSQL Servlet's `<xsql:include-xml>` and Nested `XMLAgg()` Functions to Aggregate the Results Into One XML Document

The following example uses nested `xmlagg()` functions to aggregate the results of a dynamically-constructed XML document containing departments and nested employees into a *single* XML "result" document, wrapped in a `<DepartmentList>` element:

```
<xsql:include-xml connection="orcl92" xmlns:xsql="urn:oracle-xsql">
  select XmlElement("DepartmentList",
    XmlAgg(
      XmlElement("Department",
        XmlAttributes(deptno as "Id"),
        XmlForest(dname as "Name"),
        (select XmlElement("Employees",
          XmlAgg(
            XmlElement("Employee",
              XmlAttributes(empno as "Id"),
              XmlForest(ename as "Name",
                sal as "Salary",
                job as "Job")
            )
          )
        )
      )
    )
  )
from emp e
```

```

                where e.deptno = d.deptno
            )
        )
    )
    ).getClobVal()
from dept d
order by dname
</xsql:include-xml>

```

Example 10–32 Using XSQL Servlet's <xsql:include-xml>, XMLElement(), and XMLAgg() to Generate XML from Oracle9i Database

Since it is more efficient for the database to aggregate XML fragments into a single result document, the <xsql:include-xml> element encourages this approach by only retrieving the first row from the query you provide.

For example, if you have a number of <Movie> XML documents stored in a table of XmlType called MOVIES, each document might look something like this:

```

<Movie Title="The Talented Mr.Ripley" RunningTime="139" Rating="R">
  <Director>
    <First>Anthony</First>
    <Last>Minghella</Last>
  </Director>
  <Cast>
    <Actor Role="Tom Ripley">
      <First>Matt</First>
      <Last>Damon</Last>
    </Actor>
    <Actress Role="Marge Sherwood">
      <First>Gwenyth</First>
      <Last>Paltrow</Last>
    </Actress>
    <Actor Role="Dickie Greenleaf">
      <First>Jude</First>
      <Last>Law</Last>
      <Award From="BAFTA" Category="Best Supporting Actor"/>
    </Actor>
  </Cast>
</Movie>

```

You can use the built-in Oracle9i XPath query features to extract an aggregate list of all cast members who have received Oscar awards from any movie in the database using a query like this:

```

SELECT xmlelement("AwardedActors",
    xmlagg(extract(value(m),
        '/Movie/Cast/*[Award[@From="Oscar"]]''))
    FROM movies m;

-- To include this query result of XMLType into your XSQL page,
-- simply paste the query inside an <xsql:include-xml> element, and add
-- a getClobVal() method call to the query expression so that the result will
-- be returned as a CLOB instead of as an XMLType to the client:
<xsql:include-xml connection="orcl92" xmlns:xsql="urn:oracle-xsql">
    select xmlelement("AwardedActors",
        xmlagg(extract(value(m),
            '/Movie/Cast/*[Award[@From="Oscar"]]''))).getClobVal()
    from movies m
</xsql:include-xml>

```

Note: Again we use the combination of `XMLElement()` and `XMLAgg()` to have the database aggregate all of the XML fragments identified by the query into a single, well-formed XML document.

Failing to do this results in an attempt by the XSQL page processor to parse a CLOB that looks like:

```

<Actor>...</Actor>
<Actress>...</Actress>

```

Which is not well-formed XML because it does not have a single document element as required by the XML 1.0 specification. The combination of `xmlelement()` and `xmlagg()` work together to produce a well-formed result like this:

```

<AwardedActors>
  <Actor>...</Actor>
  <Actress>...</Actress>
</AwardedActors>

```

This well-formed XML is then parsed and included in your XSQL page.

See Also: *Oracle9i XML Developer's Kits Guide - XDK*, the chapter in "XDK for Java" on XSQL Page Publishing Framework.

Generating XML Using XML SQL Utility (XSU)

The Oracle XML SQL Utility (XSU) can still be used with Oracle9i to generate XML. This might be useful if you want to generate XML on the middle-tier or the client. XSU now additionally supports generating XML on tables with XMLType columns.

Example 10–33 *Generating XML Using XSU for Java getXML*

For example, if you have table, parts:

```
CREATE TABLE parts ( PartNo number, PartName varchar2(20), PartDesc xmltype );
```

You can generate XML on this table using Java with the call:

```
java OracleXML getXML -user "scott/tiger" -rowTag "Part" "select * from parts"
```

This produces the result:

```
<Parts>
  <Part>
    <PartNo>1735</PartNo>
    <PartName>Gizmo</PartName>
    <PartDesc>
      <Description>
        <Title>Description of the Gizmo</Title>
        <Author>John Smith</Author>
        <Body>
          The <b>Gizmo</b> is <i>grand</i>.
        </Body>
      </Description>
    </PartDesc>
  </Part>
  ...
</Parts>
```

See Also : *Oracle9i XML Developer's Kits Guide - XDK* for more information on XSU

XMLType Views

This chapter describes how to create and use `XMLType` views. It contains the following sections:

- [What Are XMLType Views?](#)
- [Creating Non-Schema-Based XMLType Views](#)
- [Creating XML Schema-Based XMLType Views](#)
- [Creating XMLType Views by Transforming XMLType Tables](#)
- [Referencing XMLType View Objects Using REF\(\)](#)
- [DML \(Data Manipulation Language\) on XMLType Views](#)
- [Query Rewrite on XMLType Views](#)
- [Ad-Hoc Generation of XML Schema-Based XML](#)
- [Validating User-Specified Information](#)

What Are XMLType Views?

XMLType views wrap existing relational and object-relational data in XML formats. The major advantages of using XMLType views are:

- You can exploit the new Oracle XML DB XML features that use XMLSchema functionality without having to migrate your base legacy data.
- With XMLType views, you can experiment with various other forms of storage, besides the object-relational or CLOB storage alternatives available to XMLType tables.

XMLType views are similar to object views. Each row of an XMLType view corresponds to an XMLType instance. The object identifier for uniquely identifying each row in the view can be created using an expression such as `extract()` on the XMLType value.

Similar to XMLType tables, XMLType views can conform to an XML schema. This provides stronger typing and enables optimization of queries over these views.

To use XMLType views with XML schemas, you must first register your XML schema with annotations that represent the bi-directional mapping from XML to SQL object types. An XMLType view conforming to this registered XML schema can then be created by providing an underlying query that constructs instances of the appropriate SQL object type.

See Also:

- [Chapter 5, "Structured Mapping of XMLType"](#)
- [Appendix B, "XML Schema Primer"](#)

This chapter describes the two main ways you can create XMLType views:

- Based on XML generation functions
- Based on object types

Creating Non-Schema-Based XMLType Views

You can create a view of XMLType or a view with one or more XMLType columns, by using the SQL XML generation functions, particularly those that comply with the emerging SQLX standards.

See Also: [Chapter 10, "Generating XML Data from the Database"](#), for details on SQLX generation functions.

Example 11–1 XMLType View: Creating XMLType View Using XMLElement() Function

The following statement creates an XMLType view using XMLElement() generation function:

```
DROP TABLE employees;
CREATE TABLE employees
(empno number(4), fname varchar2(20), lname varchar2(20), hire date, salary
number(6));

INSERT INTO employees VALUES
(2100, 'John', 'Smith', Date'2000-05-24', 30000);

INSERT INTO employees VALUES
(2200, 'Mary', 'Martin', Date'1996-02-01', 30000);

CREATE OR REPLACE VIEW Emp_view OF XMLTYPE WITH OBJECT ID
(EXTRACT(sys_nc_rowinfo$, '/Emp/@empno').getnumberval())
AS SELECT XMLELEMENT("Emp", XMLAttributes(empno),
XMLForest(e.fname || ' ' || e.lname AS "name",
e.hire AS "hiredate")) AS "result"
FROM employees e
WHERE salary > 20000;
```

A query against the XMLType view returns the following employee data in XML format:

```
SELECT * FROM Emp_view;
<Emp empno="2100">
  <name>John Smith</name>
  <hiredate>2000-05-24</hiredate>
</Emp>

<Emp empno="2200">
  <name>Mary Martin</name>
  <hiredate>1996-02-01</hiredate>
</Emp>
```

empno attribute in the document should become the unique identifier for each row. SYS_NC_ROWINFO\$ is a virtual column that references the row XMLType instance.

Note: In prior releases, the object identifier clause only supported attributes of the object type of the view to be specified. In this release, this has been enhanced to support any expression returning a scalar value.

You can perform DML operations on these XMLType views, but, in general, you must write instead-of triggers to handle the DML operation.

XMLType Views can also be created using SYS_XMLGEN. An equivalent query that produces the same query results using SYS_XMLGEN is as follows :

```
CREATE TYPE Emp_t AS OBJECT ("@empno" number(4), fname varchar2(2000),
lname      varchar2(2000), hiredate date);

CREATE VIEW employee_view OF XMLTYPE WITH OBJECT ID
(EXTRACT(sys_nc_rowinfo$, '/Emp/@empno').getnumberval())
AS SELECT SYS_XMLGEN(emp_t(empno, fname, lname, hire),
XMLFORMAT('EMP'))
FROM employees e
WHERE salary > 20000;
```

Existing data in relational or object-relational tables or views can be exposed as XML using this mechanism. In addition, queries involving simple XPath traversal over SYS_XMLGEN views are candidates for query rewrite to directly access the object attributes.

Creating XML Schema-Based XMLType Views

To wrap relational data with strongly-typed XML, that is, XML based on an XML schema, perform these steps:

1. Create object types
2. Create or generate and then register an XML schema document that contains the XML structures, along with its mapping to the SQL object types and attributes. See [Chapter 5, "Structured Mapping of XMLType"](#).
3. Create the XMLType view and specify the XML schema URL and the root element name. The underlying view query first constructs the object instances and then converts them to XML. This step can also be done in two steps:
 - Create an object view
 - Create an XMLType view over the object view

The mechanism for creating XMLType views is more convenient when you already have an object-relational schema and want to map it directly to XML. Also, since the view is based on XML schema, it derives several performance (memory and access) optimizations.

You can create XML schema-based XMLType views without creating object types. For this, you can use the SQL XML generation functions or transformation functions to generate an XML schema conformant XMLType instance. The use of object types with schemas however, enables Query Rewrite functionality.

Consider the following examples based on the canonical employee -department relational tables and XML views of this data:

- [Creating Non-Schema-Based XMLType Views](#)
- [XMLType View: View 2, Wrapping Relational Department Data with Nested Employee Data as XML](#)

Example 11–2 Creating Non-Schema-Based XMLType Views

For the first example view, to wrap the relational employee data with nested department information as XML, follow these steps:

Step 1. Create Object Types

```
CREATE OR REPLACE TYPE dept_t AS OBJECT
(
  DEPTNO      NUMBER(2),
  DNAME       VARCHAR2(14),
  LOC         VARCHAR2(13)
);
/
CREATE OR REPLACE TYPE emp_t AS OBJECT
(
  EMPNO       NUMBER(4),
  ENAME       VARCHAR2(10),
  JOB         VARCHAR2(9),
  MGR         NUMBER(4),
  HIREDATE    DATE,
  SAL         NUMBER(7,2),
  COMM        NUMBER(7,2),
  DEPT        DEPT_T
);
/
```

Step 2. Create or Generate XMLSchema, emp.xsd

You can create the XML schema by hand or you can use the DBMS_XMLSchema package to generate the XML schema automatically from the existing object types. For example:

```
SELECT DBMS_XMLSchema.generateSchema('SCOTT','EMP_T') AS result FROM DUAL;
```

generates the XML schema for the employee type. You can supply various arguments to this function to add namespaces, and so on. You can further edit the XML schema to change the various default mappings that were generated. generateSchemas() function in the package generates a list of XML schemas one for each different SQL database schema referenced by the object type and its attributes.

Step 3. Register XML Schema, emp.xsd

XML schema, emp.xsd also specifies how the XML elements and attributes are mapped to their corresponding attributes in the object types, as follows:

```
BEGIN
  dbms_xmlschema.deleteSchema('http://www.oracle.com/emp.xsd', 4);
END;
/
BEGIN
  dbms_xmlschema.registerSchema('http://www.oracle.com/emp.xsd',
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.oracle.com/emp.xsd" version="1.0"
    xmlns:xdb="http://xmlns.oracle.com/xdb"
    elementFormDefault="qualified">
    <element name = "Employee" xdb:SQLType="EMP_T" xdb:SQLSchema="SCOTT">
      <complexType>
        <sequence>
          <element name = "EmployeeId" type = "positiveInteger" xdb:SQLName="EMPNO"
            xdb:SQLType="NUMBER" />
          <element name = "Name" type = "string" xdb:SQLName="ENAME"
            xdb:SQLType="VARCHAR2" />
          <element name = "Job" type = "string" xdb:SQLName="JOB"
            xdb:SQLType="VARCHAR2" />
          <element name = "Manager" type = "positiveInteger" xdb:SQLName="MGR"
            xdb:SQLType="NUMBER" />
          <element name = "HireDate" type = "date" xdb:SQLName="HIREDATE"
            xdb:SQLType="DATE" />
          <element name = "Salary" type = "positiveInteger" xdb:SQLName="SAL"
            xdb:SQLType="NUMBER" />
          <element name = "Commission" type = "positiveInteger" xdb:SQLName="COMM"
```

```

                                xdb:SQLType="NUMBER" />
    <element name = "Dept" xdb:SQLName="DEPT" xdb:SQLType="DEPT_T"
xdb:SQLSchema="SCOTT">
    <complexType>
    <sequence>
    <element name = "DeptNo" type = "positiveInteger"
xdb:SQLName="DEPTNO"
                                xdb:SQLType="NUMBER" />
    <element name = "DeptName" type = "string" xdb:SQLName="DNAME"
xdb:SQLType="VARCHAR2"/>
    <element name = "Location" type = "string" xdb:SQLName="LOC"
xdb:SQLType="VARCHAR2"/>
    </sequence>
    </complexType>
    </element>
    </sequence>
    </complexType>
    </element>
</schema>', TRUE, FALSE, FALSE);
END;
/

```

The preceding statement registers the XML schema with the target location:

```
"http://www.oracle.com/emp.xsd"
```

Step 4a. Create XMLType View Using the One-Step Process

With the one-step process you must create an XMLType view on the relational tables as follows:

```

CREATE OR REPLACE VIEW emp_xml OF XMLTYPE
    XMLSCHEMA "http://www.oracle.com/emp.xsd" ELEMENT "Employee"
    WITH OBJECT ID (ExtractValue(sys_nc_rowinfo$, '/Employee/EmployeeId')) AS
    SELECT emp_t(e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm,
        dept_t(d.deptno, d.dname, d.loc))
    FROM emp e, dept d
    WHERE e.deptno = d.deptno;

```

This example uses the `extractValue()` SQL function here in the OBJECT ID clause, since `extractValue()` can automatically figure out the appropriate SQL datatype mapping (in this case a SQL Number) using the XML schema information.

Step 4b. Create XMLType View Using the Two-Step Process by First Creating an Object View

In the two step process, you first create an object-relational view, then create an XMLType view on the object-relational view, as follows:

```
CREATE OR REPLACE VIEW emp_v OF emp_t WITH OBJECT ID (empno) AS
    SELECT emp_t(e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm,
               dept_t(d.deptno, d.dname, d.loc))
    FROM emp e, dept d
    WHERE e.deptno = d.deptno;

-- Create the employee XMLType view over the emp_v object view
CREATE OR REPLACE VIEW emp_xml OF XMLTYPE
    XMLSCHEMA "http://www.oracle.com/emp.xsd" ELEMENT "Employee"
    WITH OBJECT ID DEFAULT
    AS SELECT VALUE(p) FROM emp_v p;
```

Step 4c. Create XMLType View Using the One-Step Process Without Types

You can also create the XMLType views using the SQL XML generation functions without the need for object types. You can also use XMLTransform() or other SQL functions which generate XML. The resultant XML must be conformant to the XML schema specified for the view.

With the one-step process you must create an XMLType view on the relational tables without having to create and register any object type as follows:

```
CREATE OR REPLACE VIEW emp_xml OF XMLTYPE
    XMLSCHEMA "http://www.oracle.com/emp.xsd" ELEMENT "Employee"
    WITH OBJECT ID (extract(sys_nc_rowinfo$,
'/Employee/EmployeeId/text()').getnumberval()) AS
    SELECT XMLElement("Employee",
    XMLAttributes( 'http://www.oracle.com/emp.xsd' AS "xmlns" ,
                  'http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
                  'http://www.oracle.com/emp.xsd'
                  http://www.oracle.com/emp.xsd' AS "xsi:schemaLocation"),
    XMLForest(e.empno AS "EmployeeId", e.ename AS "Name",
              e.job AS "Job" , e.mgr AS "Manager",
              e.hiredate AS "HireDate", e.sal AS "Salary",
              e.comm AS "Commission",
              XMLForest(d.deptno AS "DeptNo",
                        d.dname AS "DeptName",
                        d.loc AS "Location") AS "Dept"))
    FROM emp e, dept d
    WHERE e.deptno = d.deptno;
```

The `XMLElement()` function creates the Employee XML element and the inner `XMLForest()` creates the kids of the employee element. The `XMLAttributes` clause inside the `XMLElement()` constructs the required XML namespace and schema location attributes so that the XML generated conforms to the view's XML schema. The innermost `XMLForest()` function creates the department XML element that is nested inside the Employee element.

The XML generation function simply generate a non-XML schema-based XML instance. However, in the case of XMLType views, as long as the names of the elements and attributes match those in the XML schema, Oracle converts this XML implicitly into a well-formed and valid XML schema-based document.

Using Multiple Namespaces

If you have complicated XML schemas involving multiple namespaces, then you would need to use the partially escaped mapping provided in the SQL functions and create elements with the appropriate namespaces and prefixes.

Example 11–3 Using Multiple Namespaces in XMLType Views

```
-- For example the SQL query:
SELECT  XMLElement("ipo:Employee",
                XMLAttributes('http://www.oracle.com/emp.xsd' AS "xmlns:ipo",
                              'http://www.oracle.com/emp.xsd
                              http://www.oracle.com/emp.xsd' AS "xmlns:xsi"),
                XMLForest(e.empno AS "ipo:EmployeeId", e.ename AS "ipo:Name",
                          e.job AS "ipo:Job", e.mgr AS "ipo:Manager",
                          e.hiredate AS "ipo:HireDate", e.sal AS "ipo:Salary",
                          e.comm AS "ipo:Commission",
                          XMLForest(d.deptno AS "ipo:DeptNo",
                                    d.dname AS "ipo:DeptName",
                                    d.loc AS "ipo:Location") AS "ipo:Dept"))
FROM emp e, dept d
WHERE e.deptno = d.deptno;

-- creates the XML instance with the correct namespace, prefixes and target
schema
-- location, and can be used as the query in the view definition:
-- <ipo:Employee xmlns="http://www.oracle.com/emp.xsd"
--   xmlns:xsi="http://www.oracle.com/emp.xsd
--     http://www.oracle.com/emp.xsd">
--   <ipo:EmployeeId>2100</ipo:EmployeeId>
--   <ipo:Name>John</ipo:Name>
```

```
--      <ipo:Manager>Mary</ipo:Manager>
--      <ipo:Hiredate>12-Jan-01</ipo:Hiredate>
--      <ipo:Salary>123003</ipo:Salary>
--      <ipo:Dept>
--          <ipo:Deptno>2000</ipo:Deptno>
--          <ipo:DeptName>Sports</ipo:DeptName>
--          <ipo:Location>San Francisco</ipo:Location>
--      </ipo:Dept>
-- </ipo:Employee>
```

If the XML schema had no target namespace then you can use the `xsi:noNamespaceSchemaLocation` attribute to denote that. For example, consider the following XML schema that is registered at location: "emp-noname.xsd":

```
BEGIN
  dbms_xmlschema.deleteSchema('emp-noname.xsd', 4);
END;
/

BEGIN
  dbms_xmlschema.registerSchema('emp-noname.xsd',
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:xdb="http://xmlns.oracle.com/xdb">
    <xs:element name = "Employee" xdb:defaultTable="EMP37_TAB">
      <xs:complexType>
        <xs:sequence>
          <xs:element name = "EmployeeId" type = "xs:positiveInteger"/>
          <xs:element name = "FirstName" type = "xs:string"/>
          <xs:element name = "LastName" type = "xs:string"/>
          <xs:element name = "Salary" type = "xs:positiveInteger"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>');
END;
/
```

The following CREATE OR REPLACE VIEW statement creates a view that conforms to this XML schema:

```
CREATE OR REPLACE VIEW emp_xml OF XMLTYPE
  XMLSCHEMA "emp-noname.xsd" ELEMENT "Employee"
  WITH OBJECT ID (extract(sys_nc_rowinfo$,
    '/Employee/EmployeeId/text()').getnumberval()) AS
```

```

SELECT XMLElement("Employee",
  XMLAttributes('http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
    'emp-noname.xsd' AS "xsi:noNamespaceSchemaLocation"),
  XMLForest(e.empno AS "EmployeeId", e.ename AS "Name",
    e.job AS "Job", e.mgr AS "Manager",
    e.hiredate AS "HireDate", e.sal AS "Salary",
    e.comm AS "Commission",
    XMLForest(d.deptno AS "DeptNo",
      d.dname AS "DeptName",
      d.loc AS "Location") AS "Dept"))
FROM emp e, dept d
WHERE e.deptno = d.deptno;

```

The `XMLAttributes` clause creates an XML element which contains the `noNamespace` schema location attribute.

Example 11–4 XMLType View: View 2, Wrapping Relational Department Data with Nested Employee Data as XML

For the second example view, to wrap the relational department data with nested employee information as XML, follow these steps:

Step 1. Create Object Types

```

DROP TYPE emp_t FORCE;
DROP TYPE dept_t FORCE;
CREATE OR REPLACE TYPE emp_t AS OBJECT
(
  EMPNO          NUMBER(4),
  ENAME          VARCHAR2(10),
  JOB            VARCHAR2(9),
  MGR            NUMBER(4),
  HIREDATE       DATE,
  SAL            NUMBER(7,2),
  COMM           NUMBER(7,2)
);
/
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),
  LOC            VARCHAR2(13),
  EMPS           EMPLIST_T

```

```

);
/

```

Step 2. Register XML Schema, dept.xsd

You can either use a pre-existing XML schema or you can generate an XML schema from the object type using the `DBMS_XMLSchema.generateSchema(s)` functions:

```

BEGIN
dbms_xmlschema.deleteSchema('http://www.oracle.com/dept.xsd', 4);
END;
/
BEGIN
dbms_xmlschema.registerSchema('http://www.oracle.com/dept.xsd',
'<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/dept.xsd" version="1.0"
xmlns:xdb="http://xmlns.oracle.com/xdb"
elementFormDefault="qualified">
<element name = "Department" xdb:SQLType="DEPT_T" xdb:SQLSchema="SCOTT">
<complexType>
<sequence>
<element name = "DeptNo" type = "positiveInteger" xdb:SQLName="DEPTNO"
xdb:SQLType="NUMBER"/>
<element name = "DeptName" type = "string" xdb:SQLName="DNAME"
xdb:SQLType="VARCHAR2"/>
<element name = "Location" type = "string" xdb:SQLName="LOC"
xdb:SQLType="VARCHAR2"/>
<element name = "Employee" maxOccurs = "unbounded" xdb:SQLName = "EMPS"
xdb:SQLType="EMPLIST_T"
xdb:SQLSchema="SCOTT">
<complexType>
<sequence>
<element name = "EmployeeId" type = "positiveInteger"
xdb:SQLName="EMPNO"
xdb:SQLType="NUMBER"/>
<element name = "Name" type = "string" xdb:SQLName="ENAME"
xdb:SQLType="VARCHAR2"/>
<element name = "Job" type = "string" xdb:SQLName="JOB"
xdb:SQLType="VARCHAR2"/>
<element name = "Manager" type = "positiveInteger"
xdb:SQLName="MGR"
xdb:SQLType="NUMBER"/>
<element name = "HireDate" type = "date" xdb:SQLName="HIREDATE"
xdb:SQLType="DATE"/>
<element name = "Salary" type = "positiveInteger" xdb:SQLName="SAL"

```

```

                                xdb:SQLType="NUMBER"/>
                                <element name = "Commission" type = "positiveInteger"
                                xdb:SQLName="COMM"
xdb:SQLType="NUMBER"/>
                                </sequence>
                                </complexType>
                                </element>
                                </sequence>
                                </complexType>
                                </element>
</schema>', TRUE, FALSE, FALSE);
END;
/

```

Step 3a. Create XMLType Views on Relational Tables

Create the dept_xml XMLType view from the department object type as follows:

```

CREATE OR REPLACE VIEW dept_xml OF XMLTYPE
XMLSCHEMA "http://www.oracle.com/dept.xsd" ELEMENT "Department"
WITH OBJECT ID (EXTRACTVALUE(sys_nc_rowinfo$, '/Department/DeptNo')) AS
SELECT dept_t(d.deptno, d.dname, d.loc,
cast(multiset(
SELECT emp_t(e.empno, e.ename, e.job, e.mgr, e.hiredate,
e.sal,e.comm) FROM emp e
WHERE e.deptno = d.deptno)
AS emplist_t))
FROM dept d;

```

Step 3b. Create XMLType Views on Relational Tables using SQL functions

Create the dept_xml XMLType view from the relational tables without object types:

```

CREATE OR REPLACE VIEW dept_xml OF XMLTYPE
XMLSCHEMA "http://www.oracle.com/dept.xsd" ELEMENT "Department"
WITH OBJECT ID (EXTRACT(sys_nc_rowinfo$,
'/Department/DeptNo').getNumberVal()) 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(deptno "DeptNo", d.dname "DeptName", d.loc "Location"),
(SELECT XMLAGG(XMLElement("Employee",

```

```
XMLForest(e.empno "EmployeeId", e.ename "Name",
          e.job "Job", e.mgr "Manager",
          e.hiredate "Hiredate"),
          e.sal "Salary",e.comm "Commission"))
FROM emp e
WHERE e.deptno = d.deptno))
FROM dept d;

-- Step 3c. Create XMLType Views on XMLType Tables
-- An XMLType view can be created on an XMLType table, perhaps to transform the
XML
-- or restrict the rows returned by some predicates, as follows:
DROP TABLE dept_xml_tab;
CREATE TABLE dept_xml_tab OF XMLTYPE
XMLSCHEMA "http://www.oracle.com/dept.xsd" ELEMENT "Department"
nested table xmldata."EMPS" store as dept_xml_tab_tab1;

CREATE OR REPLACE VIEW dept_xml_view OF XMLTYPE
XMLSCHEMA "http://www.oracle.com/dept.xsd" ELEMENT "Department"
AS
SELECT VALUE(p) FROM dept_xml_tab p
WHERE Extractvalue(value(p), '/DeptNo') = 10;
```

Note: The XML schema and element information must be specified at the view level because the SELECT list could arbitrarily construct XML of a different XML schema from the underlying table.

Creating XMLType Views by Transforming XMLType Tables

You can also create XMLType views by transforming XMLType tables.

Example 11–5 *Creating an XMLType View by Transforming an XMLType Table*

For example, consider the creation of XMLType table `po_tab`. Refer to [Example 6–1, "Transforming an XMLType Instance Using XMLTransform\(\) and DBUriType to Get the XSL Stylesheet"](#) on page 6-6 for an `xmltransform()` example:

```
DROP TABLE po_tab;
CREATE TABLE po_tab OF xmltype xmlschema "ipo.xsd" element
"PurchaseOrder";

-- You can then create a view of the table as follows:
```

```

CREATE OR REPLACE VIEW HR_PO_tab OF xmltype xmlschema "hrpo.xsd" element
  "PurchaseOrder"
  WITH OBJECT ID DEFAULT
  AS SELECT
    xmltransform(value(p),xdburitype('/home/SCOTT/xsl/po2.xsl')).getxml()
  FROM po_tab p;

```

Referencing XMLType View Objects Using REF()

You can reference an XMLType view object using the REF () syntax:

```
SELECT REF(p) FROM dept_xml p;
```

XMLType view reference REF() is based on one of the following object IDs:

- On a system-generated OID — for views on XMLType tables or object views
- On a 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 can be used inside SQL queries. These REFs behave in the same way as REFs to object views.

DML (Data Manipulation Language) on XMLType Views

An XMLType view may not be inherently updatable. This means that you have to write INSTEAD-OF -TRIGGERS to handle all data manipulation (DML). You can identify cases where the view is implicitly updatable, by analyzing the underlying view query.

Example 11–6 Identifying When a View is Implicitly Updatable

For example, if the XMLType view query is based on an object view or an object constructor that is itself inherently updatable:

```

DROP TYPE dept_t force;
CREATE OR REPLACE TYPE dept_t AS OBJECT
  (
    DEPTNO          NUMBER(2) ,
    DNAME           VARCHAR2(14) ,
    LOC             VARCHAR2(13)
  );
/

BEGIN

```

```

dbms_xmlschema.deleteSchema('http://www.oracle.com/dept.xsd', 4);
END;
/
BEGIN
dbms_xmlschema.registerSchema('http://www.oracle.com/dept.xsd',
'<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.oracle.com/dept.xsd" version="1.0"
  xmlns:xdb="http://xmlns.oracle.com/xdb"
  elementFormDefault="qualified">
  <element name = "Department" xdb:SQLType="DEPT_T" xdb:SQLSchema="SCOTT">
    <complexType>
      <sequence>
        <element name = "DeptNo" type = "positiveInteger" xdb:SQLName="DEPTNO"
          xdb:SQLType="NUMBER"/>
        <element name = "DeptName" type = "string" xdb:SQLName="DNAME"
          xdb:SQLType="VARCHAR2"/>
        <element name = "Location" type = "string" xdb:SQLName="LOC"
          xdb:SQLType="VARCHAR2"/>
      </sequence>
    </complexType>
  </element>
</schema>', TRUE, FALSE, FALSE);
END;
/

CREATE OR REPLACE VIEW dept_xml of xmltype
xmlschema "http://www.oracle.com/dept.xsd" element "Department"
with object id (sys_nc_rowinfo$.extract('/Department/DeptNo').getnumberval()) as
select dept_t(d.deptno, d.dname, d.loc) from dept d;

INSERT INTO dept_xml VALUES (XMLType.createXML(
'<Department xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.oracle.com/dept.xsd">
  <DeptNo>50</DeptNo>
  <DeptName>EDP</DeptName>
  <Location>NEW YORK</Location>
</Department>');

UPDATE dept_xml d
SET d.sys_nc_rowinfo$ = updateXML(d.sys_nc_rowinfo$,
'/Department/DeptNo/text()', 60)
WHERE existsNode(d.sys_nc_rowinfo$, '/Department[DeptNo=50]') = 1;

```

Query Rewrite on XMLType Views

For Query Rewrites, XMLType views are the same as regular XMLType table columns. Hence, `extract()` or `existsNode()` operations on view columns, get rewritten into underlying relational accesses for better performance.

Query Rewrite on XML Schema-Based Views

For example consider the following:

Example 11–7 Query Rewrite on XMLType Views: Query-Rewrite on Schema-Based Views

```
XCREATE OR REPLACE VIEW dept_ov OF dept_t
  WITH OBJECT ID (deptno) AS
  SELECT d.deptno, d.dname, d.loc, cast(multiset(
    SELECT emp_t(e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm)
    FROM emp e
    WHERE e.deptno = d.deptno)
    AS emplist_t)
  FROM dept d;

CREATE OR REPLACE VIEW dept_xml OF XMLTYPE
  WITH OBJECT ID (EXTRACT(sys_nc_rowinfo$, '/ROW/DEPTNO').getNumberVal()) AS
  SELECT sys_xmlgen(value(p)) FROM dept_ov p;
```

A query to select department numbers that have at least one employee making a salary more than \$200000:

```
SELECT EXTRACTVALUE(value(x), '/ROW/DEPTNO')
  FROM dept_xml x
  WHERE EXISTSNode(value(x), '/ROW/EMPS/EMP_T[SAL > 200]') = 1;
```

becomes:

```
SELECT d.deptno
  FROM dept d
  WHERE EXISTS (SELECT NULL FROM emp e WHERE e.deptno = d.deptno
    AND e.sal > 200);
```

Query Rewrite on Non-Schema-Based Views

Consider the following example:

Example 11–8 Query Rewrite on Non-Schema-Based Views

Non-schema-based XMLType views can be created on existing relational and object-relational tables and views. This provides users an XML view of the underlying data.

Existing relational data can be transformed into XMLType views by creating appropriate types, and doing a SYS_XMLGEN at the top-level. For example, the data in the emp table can be exposed as follows :

```
CREATE TYPE Emp_t AS OBJECT (
  EMPNO NUMBER(4),
  ENAME VARCHAR2(10),
  JOB   VARCHAR2(9),
  MGR   NUMBER(4),
  HIREDATE DATE,
  SAL   NUMBER(7,2),
  COMM  NUMBER(7,2));

CREATE VIEW employee_xml OF XMLTYPE
WITH OBJECT ID
  (SYS_NC_ROWINFO$.EXTRACT('/ROW/EMPNO/text()').getnumberval()) AS
  SELECT SYS_XMLGEN(
    emp_t(e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm))
  FROM emp e;
```

A major advantage of non-schema-based views is that existing object views can be easily transformed into XMLType views without any additional DDLs. For example, consider a database which contains the object view employee_ov with the following definition :

```
CREATE VIEW employee_ov OF EMP_T
WITH OBJECT ID (empno) AS
SELECT emp_t(e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm)
  FROM emp e;
```

```
-- Creating a non-schema-based XMLType views can be achieved by simply
-- calling SYS_XMLGEN over the top-level object column. No additional
-- types need to be created.
```

```
CREATE OR REPLACE VIEW employee_ov_xml OF XMLTYPE
WITH OBJECT ID
  (SYS_NC_ROWINFO$.EXTRACT('/ROW/EMPNO/text()').getnumberval()) AS
  SELECT SYS_XMLGEN(value(x)) from employee_ov x;
```

```
-- Certain kinds of queries on SYS_XMLGEN views are rewritten to
```

access the object attributes directly. Simple XPath traversals with `existsNode()`, `extractValue()`, and `extract()` are candidates for rewrite. See [Chapter 5, "Structured Mapping of XMLType"](#), ["Query Rewrite with XML Schema-Based Structured Storage"](#) on page 5-52, for details on query rewrite.

Note: Query rewrite only happens with `SYS_XMLGEN`. Queries over views based on other functions are not rewritten.

For example, a query such as the following :

```
SELECT EXTRACT(VALUE(x), '/ROW/EMPNO') FROM employee_ov_xml x
WHERE EXTRACTVALUE(value(x), '/ROW/ENAME') = 'SMITH';
```

is rewritten to :

```
SELECT SYS_XMLGEN(empno)
FROM emp e
WHERE e.ename = 'SMITH';
```

Ad-Hoc Generation of XML Schema-Based XML

In the preceding examples, the `CREATE VIEW` statement specified the XML schema URL and element name, whereas the underlying view query simply constructed a non-XML schema-based `XMLType`. However, there are several scenarios where you may want to avoid the view creation step, but still need to construct XML schema-based XML.

To achieve this, you can use the following XML generation functions to optionally accept an XML schema URL and element name:

- `createXML()`
- `SYS_XMLGEN()`
- `SYS_XMLAGG()`

See Also: [Chapter 10, "Generating XML Data from the Database"](#).

If the XML schema information is specified, the resulting XML is created to be XML schema-based:

```
SELECT XMLTYPE.createXML(dept_t(d.deptno, d.dname, d.loc,
CAST(MULTISET(SELECT emp_t(e.empno, e.ename, e.job, e.mgr,
e.hiredate, e.sal, e.comm)
```

```
FROM emp e WHERE e.deptno = d.deptno) AS emplist_t),
      'http://www.oracle.com/dept.xsd', 'Department')
FROM dept d;
```

Validating User-Specified Information

You can fill in the optional Oracle XML DB attributes *before* registering the XML schema. In this case, Oracle validates the extra information to ensure that the specified values for the Oracle XML DB attributes are compatible with the rest of the XML schema declarations. This form of XML schema registration typically happens when wrapping existing data using `XMLType` views.

See: [Chapter 5, "Structured Mapping of XMLType"](#) for more details on this process

You can use the `DBMS_XMLSchema generateSchema()` and `generateSchemas()` functions to generate the default XML mapping for specified object types. The generated XML schema document has the `SQLType`, `SQLSchema`, and so on, attributes filled in. When these XML schema documents are then registered, the following validation forms can occur:

- *SQLType for attributes or elements based on simpleType.* This is compatible with the corresponding `XMLType`. For example, an XML string datatype can only be mapped to `VARCHAR2s` or Large Objects (LOBs).
- *SQLType specified for elements based on complexType.* This is either a LOB or an object type whose structure is compatible with the declaration of the `complexType`, that is, the object type has the right number of attributes with the right datatypes.

Creating and Accessing Data Through URLs

This chapter describes how to generate and store URLs inside the database and to retrieve the data pointed to by the URLs. It also introduces the concept of DBUris which are URLs to relational data stored inside the database. It explains how to create and store references to data stored in Oracle XML DB Repository hierarchy.

This chapter contains these sections:

- [How Oracle9i Database Works with URLs and URIs](#)
- [URI Concepts](#)
- [UriTypes Store Uri-References](#)
- [HttpUriType Functions](#)
- [DBUri, Intra-Database References](#)
- [XDBUriType](#)
- [Using UriType Objects](#)
- [Creating Instances of UriType Objects with the UriFactory Package](#)
- [Why Define New Subtypes of UriType?](#)
- [SYS_DBURIGEN\(\) SQL Function](#)
- [Turning a URL into a Database Query with DBUri Servlet](#)

How Oracle9i Database Works with URLs and URIs

In developing Internet applications, and particularly Internet-based XML applications, you often need to refer to data somewhere on a network using URLs or URIs.

- A URL, or Uniform Resource Locator, refers to a complete document or a particular spot within a document.
- A URI, or Uniform Resource Identifier, is a more general form of URL. A URI can be identical to a URL, or it can use extra notation in place of the anchor to identify an enclosed section of a document (rather than a single location).

Note: Throughout this chapter, we refer to URIs because that is the more general term, but the details apply to URLs as well. Some of the type names use `Uri` instead of `URI`. Because most of this information is based on SQL and PL/SQL, the names are usually not case-sensitive; only when referring to a real filename on a Web site or a Java API name does case matter.

Oracle9i can represent various kinds of paths within the database. Each corresponds to a different object type, all derived from a general type called `UriType`:

- `HttpUriType` represents a URL that begins with `http://`. It lets you create objects that represent links to Web pages, and retrieve those Web pages by calling object methods.
- `DBUriType` represents a URI that points to a set of rows, a single row, or a single column within the database. It lets you create objects that represent links to table data, and retrieve the data by calling object methods.
- `XDBUriType` represents a URI that points to an XML document stored in the ORACLE XML DB Repository inside the database. We refer to these documents or other data as resources. It lets you create objects that represent links to resources, and retrieve all or part of any resource by calling object methods.

Accessing and Processing Data Through HTTP

Any resources stored inside ORACLE XML DB Repository can also be retrieved by using the HTTP Server in Oracle XML DB. Oracle9i also includes a servlet that makes table data available through HTTP URLs. The data can be returned as plain text, HTML, or XML.

Any Web-enabled client or application can use the data without SQL programming or any specialized database API. You can retrieve the data by linking to it in a Web page or by requesting it through the HTTP-aware APIs of Java, PL/SQL, or Perl. You can display or process the data through any kind of application, including a regular Web browser or an XML-aware application such as a spreadsheet. The servlet supports generating XML and non-XML content and also transforming the results using XSLT stylesheets.

Creating Columns and Storing Data Using UriType

You can create database columns using `UriType` or its child types, or you can store just the text of each URI or URL and create the object types as needed. When storing a mixture of subtypes in the database, you can define a `UriType` column that can store various subtypes within the same column.

Because these capabilities use object-oriented programming features such as object types and methods, you can derive your own types that inherit from the Oracle-supplied ones. Deriving new types lets you use specialized techniques for retrieving the data or transforming or filtering it before returning it to the program.

UriFactory Package

When storing just the URI text in the database, you can use the `UriFactory` package to turn each URI into an object of the appropriate subtype. `UriFactory` package creates an instance of the appropriate type by checking what kind of URI is represented by a given string. For example, any URI that begins with `http://` is considered an HTTP URL. When the `UriFactory` package is passed such a URI string, it returns an instance of a `HttpUriType` object.

See Also: ["Registering New UriType Subtypes with the UriFactory Package"](#) on page 12-26

Other Sources of Information About URIs and URLs

Before you explore the features in this chapter, you should be familiar with the notation for various kinds of URIs.

See:

- <http://www.w3.org/2002/ws/Activity.html> an explanation of HTTP URL notation
- <http://www.w3.org/TR/xpath> for an explanation of the XML XPath notation
- <http://www.w3.org/TR/xptr/> for an explanation of the XML XPointer notation
- <http://xml.coverpages.org/xmlMediaMIME.html> for a discussion of MIME types

URI Concepts

This section introduces you to URI concepts.

What Is a URI?

A URI, or Uniform Resource Identifier, is a generalized kind of URL. Like a URL, it can reference any document, and can reference a specific part of a document. It is more general than a URL because it has a powerful mechanism for specifying the relevant part of the document. A URI consists of two parts:

- **URL**, that identifies the document using the same notation as a regular URL.
- **Fragment**, that identifies a fragment within the document. The notation for the fragment depends on the document type. For HTML documents, it has the form *#anchor_name*. For XML documents, it uses XPath notation.

The fragment appears after the # in the following examples.

Note: Only `XDBUriType` and `HttpUriType` support the URI fragment in this release. `DBUriType` does *not* support the URI fragment.

How to Create a URL Path from an XML Document View

[Figure 12-1](#) shows a view of the XML data stored in a relational table, `EMP`, in the database, and the columns of data mapped to elements in the XML document. This mapping is referred to as an **XML visualization**. The resulting URL path can be derived from the XML document view.

Typical URIs look like the following:

- **For HTML:** `http://www.url.com/document1#Anchor`
where `Anchor` is a named anchor inside the document.
- **For XML:** `http://www.xml.com/xml_doc#//po/cust/custname`
where:
 - The portion before the # identifies the location of the document.
 - The portion after the # identifies a fragment within the document. This portion is defined by the W3C XPointer recommendation.

UriType Objects Can Use Different Protocols to Retrieve Data

Oracle9i introduces new datatypes in the database to store and retrieve objects that represent URIs. See "[UriTypes Store Uri-References](#)" in the following section. Each datatype uses a different protocol, such as HTTP, to retrieve data.

Oracle9i also introduces new forms of URIs that represent references to rows and columns of database tables.

Advantages of Using DBUri and XDBUri

The following are advantages of using `DBUri` and `XDBUri`:

- Reference stylesheets within database-generated Web pages. Oracle-supplied package `DBMS_METADATA` uses `DBUri`s to reference XSL stylesheets. `XDBUri` can also be used to reference XSL stylesheets stored in ORACLE XML DB Repository.
- Reference HTML, images and other data stored in the database . The URLs can be used to point to data stored in tables or in the Repository hierarchical folders.
- Improved Performance by bypassing the Web server. If you already have a URL in your XML document, you can replace it with a reference to the database by either:
 - Using a servlet
 - Using a `DBUri/XDBUri` to bring back the results

Using `DBUri/XDBUri` has performance benefits because you interact directly with the database rather than through a Web server.

- **Accessing XML Documents in the Database Without SQL.** You do not need to know SQL to access an XML document stored in the database. With `DBUri` you can access an XML document from the database without using SQL.

Since the files or resources in ORACLE XML DB Repository are stored in tables, you can access them either through the `XDBUri` or by using the table metaphor through the `DBUri`.

UriTypes Store Uri-References

URIs or Universal Resource Identifiers identify resources such as Web pages anywhere on the Web. Oracle9i provides the following `UriTypes` for storing and accessing external and internal Uri-references:

- `DBUriType`. Stores references to relational data inside the database.
- `HttpUriType`. Implements the HTTP protocol for accessing remote pages. Stores URLs to external Web pages or files. Accesses these files using Hyper Text Transfer Protocol (HTTP) protocol.
- `XDBUriType`. Stores references to resources in Oracle XML DB Repository.

These datatypes are object types with member functions that can be used to access objects or pages pointed to by the objects. By using `UriType`, you can:

- Create table columns that point to data inside or outside the database.
- Query the database columns using functions provided by `UriType`.

These are related by an inheritance hierarchy. `UriType` is an abstract type and the `DBUriType`, `HttpUriType`, and `XDBUriType` are subtypes of `UriType`. You can reference data stored in CLOBs or other columns and expose them as URLs to the external world. Oracle9i provides a standard servlet than can be installed and run under the Oracle Servlet engine that interprets these URLs.

Advantages of Using UriTypes

Oracle already provides the PL/SQL package `UTL_HTTP` and the Java class `java.net.URL` to fetch URL references. The advantages of defining this new `UriType` datatype in SQL are:

- *Improved Mapping of XML Documents to Columns.* Uri-ref support is needed when exploding XML documents into object-relational columns, so that the Uri-ref specified in documents can map to a URL column in the database.

- *Unified access to data stored inside and outside the server.* Since you can use the UriRefs to store pointers to HTTP/DB urls, you get a unified access to the data wherever it is stored. This lets you create queries and indexes without having to worry about where the data resides.

See Also: ["Using UriType Objects"](#) on page 12-22.

UriType Functions

The `UriType` abstract type supports a variety of functions that can be used over any subtype. [Table 12-1](#) lists the `UriType` member functions.

Table 12-1 *UriType Member Functions*

UriType Member Functions	Description
<code>getClob()</code>	Returns the value pointed to by the URL as a character LOB value. The character encoding will be that of the database character set.
<code>getUrl()</code>	Returns the URL stored in the <code>UriType</code> . Do not use the attribute "url" directly. Use this function instead. This can be overridden by subtypes to give you the correct URL. For example, <code>HttpUriType</code> stores only the URL and not the "http://" prefix. Hence <code>getUrl()</code> actually prepends the prefix and returns the value.
<code>getExternalUrl()</code>	Similar to the former (<code>getUrl</code>), except that it calls the escaping mechanism to escape the characters in the URL as to conform to the URL specification. For example spaces are converted to the escaped value %20. See How Oracle9i Database Works with URLs and URIs on page 12-2.
<code>getContentType()</code>	Returns the MIME information for the URL. For <code>UriType</code> , this is an abstract function.
<code>getXML()</code>	Returns the <code>XMLType</code> object corresponding to the given URI. This is provided so that an application that needs to perform operations other than <code>getClob/getBlob</code> can use the <code>XMLType</code> methods to do those operations. This throws an exception if the URI does not point to a valid XML document.

Table 12–1 UriType Member Functions (Cont.)

UriType Member Functions	Description
getBlob()	Returns the BLOB value pointed to by the URL. No character conversions are performed and the character encoding is the same as the one pointed to by the URL. This can also be used to fetch binary data.
createUri(uri IN VARCHAR2)	This constructs the UriType. It is not actually in UriType, rather it is used for creating URI subtypes.

HttpUriType Functions

Use `HttpUriType` to store references to data that can be accessed through the HTTP protocol. `HttpUriType` uses the `UTL_HTTP` package to fetch the data and hence the session settings for the package can also be used to influence the HTTP fetch using this mechanism. [Table 12–2](#) lists the `HttpUriType` member functions.

Table 12–2 HttpUriType Member Functions

HttpUriType Method	Description
getClob	Returns the value pointed to by the URL as a character LOB value. The character encoding is the same as the database character set.
getUrl	Returns stored URL.
getExternalUrl	Similar to <code>getUrl</code> , except that it calls the escaping mechanism to escape the characters in the URL as to conform to the URL specification. For example, spaces are converted to the escaped value <code>%20</code> .
getBlob	Gets the binary content as a BLOB. If the target data is non-binary then the BLOB will contain the XML or text representation of the data in the database character set.
getXML	Returns the <code>XMLType</code> object corresponding to this URI. Will throw an error if the target data is not XML. See also " getXML() Function " on page 12-9.
getContentType()	Returns the MIME information for the URL. See also " getContentType() Function " on page 12-9.
createUri()	<code>httpUriType</code> constructor. Constructs in <code>httpUriType</code> .
httpUriType()	<code>httpUriType</code> constructor. Constructs in <code>httpUriType</code> .

getContentType() Function

`getContentType()` function returns the MIME information for the URL. The `HttpUriType` de-references the URL and gets the MIME header information. You can use this information to decide whether to retrieve the URL as BLOB or CLOB based on the MIME type. You would treat a Web page with a MIME type of `x/jpeg` as a BLOB, and one with a MIME type of `text/plain` or `text/html` as a CLOB.

Example 12–1 Using `getContentType()` and `HttpUriType` to Return HTTP Headers

Getting the content type does not fetch all the data. The only data transferred is the HTTP headers (for `HTTPURiType`) or the metadata of the column (for `DBURiType`). For example:

```
declare
  httpuri HttpUriType;
  x clob;
  y blob;
begin
  httpuri := HttpUriType('http://www.oracle.com/object1');
  if httpuri.getContentType() = 'application-x/bin' then
    y := httpuri.getblob();
  else
    x := httpuri.getclob();
  end if;
end;
```

getXML() Function

`getXML()` function returns `XMLType` information for the result. If the document is not valid XML (or XHTML) an error is thrown.

DBUri, Intra-Database References

`DBUri`, a database relative to URI, is a special case of the `Uri-ref` mechanism, where `ref` is guaranteed to work inside the context of a database and session. This `ref` is not a *global ref* like the HTTP URL; instead it is *local ref* (URL) within the database.

You can also access objects pointed to by this URL globally, by appending this `DBUri` to an HTTP URL path that identifies the servlet that can handle `DBUri`. This is discussed in "[Turning a URL into a Database Query with `DBUri` Servlet](#)" on page 12-34.

Formulating the DBUri

The URL syntax is obtained by specifying XPath-like syntax over a virtual XML visualization of the database. See [Figure 12–1, "DBUri: Visual or SQL View, XML View, and Associated XPath"](#):

- The *visual model* is a hierarchical view of what a current connected user would see in terms of SQL schemas, tables, rows, and columns.
- The *XML view* contains a root element that maps to the database. The root XML element contains child elements, which are the schemas on which the user has some privileges on any object. The schema elements contain tables and views that the user can see.

Example 12–2 *The Virtual XML Document that Scott Sees*

For example, the user *scott* can see the following virtual XML document.

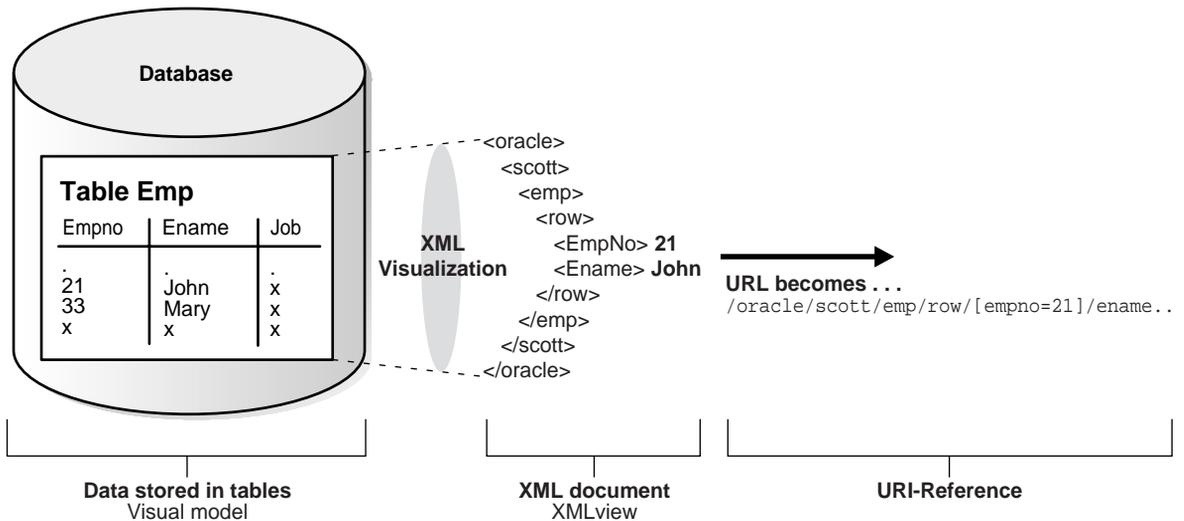
```
<?xml version='1.0'?>
<oradb SID="ORCL">
  <PUBLIC>
    <ALL_TABLES>
      ..
    </ALL_TABLES>
  <EMP>
    <!-- Emp table -->
  </EMP>
</PUBLIC>
<SCOTT>
  <ALL_TABLES>
    ....
  </ALL_TABLES>
  <EMP>
    <ROW>
      <EMPNO>1001</EMPNO>
      <ENAME>John</ENAME>
      <EMP_SALARY>20000</EMP_SALARY>
    </ROW>
    <ROW>
      <EMPNO>2001</EMPNO>
    </ROW>
  </EMP>
  <DEPT>
    <ROW>
      <DEPTNO>200</DEPTNO>
      <DNAME>Sports</DNAME>
```

```

</ROW>
</DEPT>
</SCOTT>
<JONES>
  <CUSTOMER_OBJ_TAB>
    <ROW>
      <NAME>xxx</NAME>
      <ADDRESS>
        <STATE>CA</STATE>
        <ZIP>94065</ZIP>
      </ADDRESS>
    </ROW>
  </CUSTOMER_OBJ_TAB>
</JONES>
</oradb>

```

Figure 12–1 DBUri: Visual or SQL View, XML View, and Associated XPath



This XML document is constructed at the time you do the query and based on the privileges that you have at that moment.

You can make the following observations from the previous example:

- User scott can see the scott database schema and jones database schema. These are schemas on which the user has some table or views that he can read.

- Table `emp` shows up as `EMP` with row element tags. This is the default mapping for all tables. The same for `dept` and the `customer_obj_tab` table under the `jones` schema.
- In this release, null elements are absent
- There is also a `PUBLIC` element under which tables and views are accessible without schema qualification. For example, a `SELECT` query such as:

```
SELECT * FROM emp;
```

when queried by user `scott`, matches the table `emp` under the `scott` schema and, if not found, tries to match a public synonym named `emp`. In the same way, the `PUBLIC` element contains:

- All the tables and views visible to users through their database schema
- All the tables visible through the `PUBLIC` synonym

Notation for DBUriType Fragments

With the Oracle9i database being visualized as an XML tree, you can perform XPath traversals to any part of the virtual document. This translates to any row-column intersection of the database tables or views. By specifying an XPath over the visualization model, you can create references to any piece of data in the database.

`DBUri` is specified in a simplified XPath format. Currently, Oracle does not support the full XPath or XPointer recommendation for `DBURType`. The following sections discuss the structure of the `DBUri`.

As stated in the previous paragraphs, you can now create `DBUris` to any piece of data. You can use the following instances in a column as reference:

- Scalar
- Object
- Collection
- An attribute of an object type within a column. For example:
.../ROW[empno=7263]/COL_OBJ/OBJ_ATTR

These are the smallest addressable units. For example, you can use:

```
/oradb/SCOTT/EMP
```

or

```
/oradb/SCOTT/EMP/ROW[empno=7263]
```

Note: Oracle does not currently support references within a scalar, XMLType or LOB data column.

DBUri Syntax Guidelines

There are restrictions on the kind of XPath queries that can be used to specify a reference. In general, the fragment part must:

- Include the user database schema name or specify PUBLIC to resolve the table name without a specific schema.
- Include a table or view name.
- Include the ROW tag for identifying the ROW element.
- Identify the column or object attribute that you wish to extract.
- Include predicates at any level in the path other than the schema and table elements.
- Indicate predicates not on the selection path in the ROW element.

Example 12–3 Specifying Predicate pono=100 With the ROW Node

For example, if you wanted to specify the predicate pono = 100, but the selection path is:

```
/oradb/scott/purchase_obj_tab/ROW/line_item_list
```

you must include the pono predicate along with the ROW node as:

```
/oradb/scott/purchase_obj_tab/ROW[pono=100]/line_item_list
```

- A DBUri *must* identify exactly a single data value, either an object type or a collection. If the data value is an entire row, you indicate that by including a ROW node. The DBUri can also point to an entire table.

Using Predicate (XPath) Expressions in DBUris

The predicate expressions can use the following XPath expressions:

- Boolean operators AND, OR, and NOT
- Relational operators - <, >, <=, !=, >=, =, mod, div, * (multiply)

Note:

- No XPath axes other than the child axes are supported. The wild card (*), descendant (//), and other operations are not valid.
 - Only the `text()` XPath function is supported. `text()` is valid only on a scalar node, not at the row or table level.
-
-

The predicates can be defined at any element other than the schema and table elements. If you have object columns, you can search on the attribute values as well.

Example 12–4 Searching for Attribute Values on Object Columns Using DBUri

For example, the following DBUri refers to an ADDRESS column containing state, city, street, and zip code attributes:

```
/oradb/SCOTT/EMP/ROW[ADDRESS/STATE='CA' OR  
ADDRESS/STATE='OR']/ADDRESS[CITY='Portland' OR /ZIPCODE=94404]/CITY
```

This DBUri identifies the `city` attribute whose state is either California or Oregon, or whose city name is Portland, or whose zipcode is 94404.

See Also: <http://www.w3.org/TR/xpath> for an explanation of the XML XPath notation

Some Common DBUri Scenarios

The DBUri can identify various objects, such as a table, a particular row, a particular column in a row, or a particular attribute of an object column. The following subsections describe how to identify different object types.

Identifying the Whole Table

This returns an XML document that retrieves the whole table. The enclosing tag is the name of the table. The row values are enclosed inside a `ROW` element, as follows, using the following syntax:

```
/oradb/schemaname/tablename
```

Example 12–5 Using DBUri to Identify a Whole Table as an XML Document

For example:

```
/oradb/SCOTT/EMP
```

returns an XML document with a format like the following:

```
<?xml version="1.0"?>
<EMP>
  <ROW>
    <EMPNO>7369</EMPNO>
    <ENAME>Smith</ENAME>
    ... <!-- other columns -->
  </ROW>
  <!-- other rows -->
</EMP>
```

Identifying a Particular Row of the Table

This identifies a particular ROW element in the table. The result is an XML document that contains the ROW element with its columns as child elements. Use the following syntax:

```
/oradb/schemaname/tablename/ROW[predicate_expression]
```

Example 12–6 Using DBUri to Identify a Particular Row in the Table

For example:

```
/oradb/SCOTT/EMP/ROW[EMPNO=7369]
```

returns the XML document with a format like the following:

```
<?xml version="1.0"?>
<ROW>
  <EMPNO>7369</EMPNO>
  <ENAME>SMITH</ENAME>
  <JOB>CLERK</JOB>
  <!-- other columns -->
</ROW>
```

Note: In the previous example, the predicate expression must identify a unique row.

Identifying a Target Column

In this case, a target column or an attribute of a column is identified and retrieved as XML.

Note: You cannot traverse into nested table or VARRAY columns.

Use the following syntax:

```
/oradb/schemaname/tablename/ROW[predicate_expression]/columnname  
/oradb/schemaname/tablename/ROW[predicate_expression]/columnname/attribute1../attributen
```

Example 12-7 Using DBUri to Identify a Specific Column

```
/oradb/SCOTT/EMP/ROW[EMPNO=7369 and DEPTNO=20]/ENAME
```

retrieves the `ename` column in the `emp` table, where `empno` is 7369, and department number is 20, as follows:

```
<?xml version="1.0"?>  
<ENAME>SMITH</ENAME>
```

Example 12-8 Using DBUri to Identify an Attribute Inside a Column

```
/oradb/SCOTT/EMP/ROW[EMPNO=7369]/ADDRESS/STATE
```

retrieves the `state` attribute inside an `address` object column for the employee whose `empno` is 7369, as follows:

```
<?xml version="1.0"?>  
<STATE>CA</STATE>
```

Retrieving the Text Value of a Column

In many cases, it can be useful to retrieve only the text values of a column and not the enclosing tags. For example, if XSL stylesheets are stored in a CLOB column, you can retrieve the document text without having any enclosing column name tags. You can use the `text()` function for this. It specifies that you only want the text value of the node. Use the following syntax:

```
/oradb/schemaname/tablename/ROW[predicate_expression]/columnname/text()
```

Example 12-9 Using DBUri to Retrieve Only the Text Value of the Node

For example:

```
/oradb/SCOTT/EMP/ROW[EMPNO=7369]/ENAME/text()
```

retrieves the text value of the employee name, without the XML tags, for an employee with `empno = 7369`. This returns a text document, not an XML document, with value `SMITH`.

Note: The XPath alone does not constitute a valid URI. Oracle calls it a `DBUri` since it behaves like a URI within the database, but it can be translated into a globally valid `Uri-ref`.

Note: The path is case-sensitive. To specify `scott.emp`, you use `SCOTT/EMP`, because the actual table and column names are stored capitalized in the Oracle data dictionary. If you need to use lowercase path values, you can create a lowercase table or column name by enclosing the name in double quotation marks.

How DBUris Differ from Object References

A `DBUri` can access columns and attributes and is loosely typed Object references can only access row objects. `DBUri` is a superset of this reference mechanism.

DBUri Applies to a Database and Session

A `DBUri` is scoped to a database and session. You must already be connected to the database in a particular session context. The schema and permissions needed to access the data are resolved in that context.

Note: The same URI string may give different results based on the session context used, particularly if the `PUBLIC` path is used.

For example, `/PUBLIC/FOO_TAB` can resolve to `SCOTT.FOO_TAB` when connected as `scott`, and resolve as `JONES.FOO_TAB` when connected as `JONES`.

Where Can DBUri Be Used?

`Uri-ref` can be used in a number of scenarios, including those described in the following sections:

Storing URLs to Related Documents

In the case of a travel story Web site where you store travel stories in a table, you might create links to related stories. By representing these links in a `DBUriType` column, you can create intra-database links that let you retrieve related stories through queries.

Storing Stylesheets in the Database

Applications can use XSL stylesheets to convert XML into other formats. The stylesheets are represented as XML documents, stored as CLOBs. The application can use `DBUriType` objects:

- To access the XSL stylesheets stored in the database for use during parsing.
- To make references, such as `import` or `include`, to related XSL stylesheets. You can encode these references within the XSL stylesheet itself.

Note:

- A `DBUri` is not a general purpose XPointer mechanism to XML data.
 - It is not a replacement for database object references. The syntax and semantics of references differ from those of `Uri-refs`.
 - It does not enforce or create any new security models or restrictions. Instead, it relies on the underlying security architecture to enforce privileges.
-
-

DBUriType Functions

Table 12–3 lists the `DBUriType` methods and functions.

Table 12–3 *DBUriType Methods and Functions*

Method/Function	Description
<code>getClob()</code>	Returns the value pointed to by the URL as a character LOB value. The character encoding is the same as the database character set.
<code>getUrl()</code>	Returns the URL that is stored in the <code>DBUriType</code> .
<code>getExternalUrl()</code>	Similar to <code>getUrl()</code> , except that it calls the escaping mechanism to escape the characters in the URL as to conform to the URL specification. For example, spaces are converted to the escaped value <code>%20</code> .

Method/Function	Description
<code>getBlob()</code>	Gets the binary content as a BLOB. If the target data is non-binary, then the BLOB will contain the XML or text representation of the data in the database character set.
<code>getXML()</code>	Returns the <code>XMLType</code> object corresponding to this URI.
<code>getContentType()</code>	Returns the MIME information for the URL.
<code>createUri()</code>	Constructs a <code>DBUriType</code> instance.
<code>dbUriType()</code>	Constructs a <code>DBUriType</code> instance.

Some of the functions that have a different or special behavior in the `DBUriType` are described in the following subsections.

getContentType() Function

This function returns the MIME information for the URL. The content type for a `DBUriType` object can be:

- If the `DBUri` points to a scalar value, where the MIME type is `text/plain`.
- In all other cases, the MIME type is `text/xml`.

For example, consider the table `dbtab` under `SCOTT`:

```
CREATE TABLE DBTAB( a varchar2(20), b blob);
```

A `DBUriType` of `' /SCOTT/DBTAB/ROW/A '` has a content type of `text/xml`, since it points to the whole column and the result is XML.

A `DBUriType` of `' /SCOTT/DBTAB/ROW/B '` also has a content type of `text/xml`.

A `DBUriType` of `' /SCOTT/DBTAB/ROW/A/text() '` has a content type of `text/plain`.

A `DBUriType` of `' /SCOTT/DBTAB/ROW/B/text() '` has a content type of `text/plain`.

getClob() and getBlob() Functions

In the case of `DBUri`, scalar binary data is handled specially. In the case of a `getClob()` call on a `DBUri` `' /SCOTT/DBTAB/ROW/B/text() '` where `B` is a BLOB column, the data is converted to HEX and sent out.

In the case of a `getBlob()` call, the data is returned in binary form. However, if an XML document is requested, as in `' / SCOTT / DBTAB / ROW / B '`, then the XML document will contain the binary in HEX form.

XDBUriType

`XDBUriType` is a new subtype of `UriType`. It provides a way to expose documents in the ORACLE XML DB Repository as URIs that can be embedded in any `UriType` column in a table.

The URL part of the URI is the hierarchical name of the XML document it refers to. The optional fragment part uses the XPath syntax, and is separated from the URL part by `'#'`.

The following are examples of ORACLE XML DB URIs:

```
/home/scott/doc1.xml  
/home/scott/doc1.xml#/purchaseOrder/lineItem
```

where:

- `' /home / scott '` is a folder in Oracle XML DB Repository
- `doc1.xml` is an XML document in this folder
- The XPath expression `/purchaseOrder/lineItem` refers to the line item in this purchase order document.

[Table 12-4](#) lists the `XDBUriType` methods. These methods do not take any arguments.

Table 12-4 XDBUriType Methods

Method	Description
<code>getClob()</code>	Returns the value pointed to by the URL as a Character Large Object (CLOB) value. The character encoding is the same as the database character set.
<code>getBlob()</code>	Returns the value pointed to by the URL as a Binary Large Object (BLOB) value.
<code>getUrl()</code>	Returns the URL that is stored in the <code>XDBUriType</code> .
<code>getExternalUrl()</code>	Similar to <code>getUrl()</code> , except that it calls the escaping mechanism to escape the characters in the URL as to conform to the URL specification. For example, spaces are converted to the escaped value <code>%20</code> .

Table 12–4 XDBUriType Methods (Cont.)

Method	Description
getXML()	Returns the XMLType object corresponding to the contents of the resource that this URI points to. This is provided so that an application that needs to perform operations other than getClob/getBlob can use the XMLType methods to do those operations..
getContentType()	Returns the MIME information for the resource stored in the ORACLE XML DB Repository.
XDBUriType()	Constructor. Returns an XDBUriType for the given URI.

How to Create an Instance of XDBUriType

XDBUriType is automatically registered with UriFactory so that an XDBUriType instance can be generated by providing the URI to the getURI method.

Currently, XDBUriType is the default UriType generated by the UriFactory.getUri method, when the URI does not have any of the recognized prefixes, such as "http://", "/DBURI", or "/ORADB".

All DBUriType URIs should have a prefix of either /DBURI or /ORADB, case insensitive.

Example 12–10 Returning XDBUriType Instance

For example, the following statement returns an XDBUriType instance that refers to /home/scott/doc1.xml:

```
SELECT sys.UriFactory.getUri('/home/scott/doc1.xml') FROM dual;
```

Example 12–11 Creating XDBUriType, Inserting Values Into a Purchase Order Table and Selecting All the PurchaseOrders

The following is an example of how XDBUriType is used:

```
CREATE TABLE uri_tab
(
  poUrl SYS.UriType, -- Note that we have created an abstract type column
                    --so that any type of URI can be used
  poName VARCHAR2(1000)
);

-- insert an absolute url into poUrl
-- the factory will create an XDBUriType since there's no prefix
```

```
INSERT INTO uri_tab VALUES
  (UriFactory.getUri('/public/orders/po1.xml'), 'SomePurchaseOrder');

-- Now get all the purchase orders
SELECT e.poUrl.getClob(), poName FROM uri_tab e;

-- Using PL/SQL, you can access table uri_tab as follows:
declare
  a UriType;
begin
  -- Get the absolute URL for purchase order named like 'Some%'
  SELECT poUrl into a from uri_tab WHERE poName like 'Some%';
  printDataOut(a.getClob());
end;
/
```

Example 12-12 Retrieving Purchase Orders at a URL Using UriType, getXML() and extractValue()

Since `getXML()` returns an `XMLType`, it can be used in the `EXTRACT` family of operators. For example:

```
SELECT e.poUrl.getClob() FROM uri_tab e
  WHERE extractValue(e.poUrl.getXML(), '/User') = 'SCOTT';
```

This statement retrieves all Purchase Orders for user SCOTT.

Creating Oracle Text Indexes on UriType Columns

`UriType` columns can be indexed natively in Oracle9i database using Oracle Text. No special datastore is needed.

See: [Chapter 7, "Searching XML Data with Oracle Text", "XMLType Indexing"](#) on page 7-34

Using UriType Objects

This section describes how to store pointers to documents and retrieve these documents across the network, either from the database or a Web site.

Storing Pointers to Documents with UriType

As explained earlier, `UriType` is an abstract type containing a `VARCHAR2` attribute that specifies the URI. The object type has functions for traversing the reference and extracting the data.

You can create columns using `UriType` to store these pointers in the database. Typically, you declare the column using the `UriType`, and the objects that you store use one or more of the derived types such as `HttpUriType`.

[Table 12–4](#) lists some useful `UriType` methods.

Note: You can plug in any new protocol using the inheritance mechanism. Oracle provides `HttpUriType` and `DBUriType` types for handling HTTP protocol and for deciphering `DBUri` references. For example, you can implement a subtype of `UriType` to handle the `gopher` protocol.

Example 12–13 *Creating URL References to a List of Purchase Orders*

You can create a list of all purchase orders with URL references to them as follows:

```
CREATE TABLE uri_tab
(
  poUrl SYS.UriType, -- Note that we have created abstract type columns
  -- if you know what kind of uri's you are going to store, you can
  -- create the appropriate types.
  poName VARCHAR2(200)
);

-- insert an absolute url into SYS.UriType..!
-- the UriFactory creates the correct instance (in this case a HttpUriType
INSERT INTO uri_tab VALUES
  (sys.UriFactory.getUri('http://www.oracle.com/cust/po'), 'AbsPo');

-- insert a URL by directly calling the SYS.HttpUriType constructor.
-- Note this is strongly discouraged. Note the absence of the
-- http:// prefix when creating SYS.HttpUriType instance through the default
-- constructor.
INSERT INTO uri_tab VALUES (sys.HttpUriType('proxy.us.oracle.com'), 'RelPo');

-- Now extract all the purchase orders
SELECT e.poUrl.getClob(), poName FROM uri_tab e;
```

```
-- In PL/SQL
declare
  a SYS.UriType;
begin

  -- absolute URL
  SELECT poUrl into a from uri_Tab WHERE poName like 'AbsPo%';

  SELECT poUrl into a from uri_Tab WHERE poName like 'RelPo%';
  -- here u need to supply a prefix before u can get at the data..!
  printDataOut(a.getClob());
end;
/
```

See: ["Creating Instances of UriType Objects with the UriFactory Package"](#) on page 12-25 for a description of how to use UriFactory

Using the Substitution Mechanism

You can create columns of the `UriType` directly and insert `HttpUriTypes`, `XDBUriTypes`, and `DBUriTypes` into that column. You can also query the column without knowing where the referenced document lies. For example, from the previous example, you inserted `DBUri` references into the `uri_tab` table as follows:

```
INSERT INTO uri_tab VALUES
  (UriFactory.getUri(
    '/SCOTT/PURCHASE_ORDER_TAB/ROW[PONO=1000]','ScottPo');
```

This insert assumes that there is a purchase order table in the `SCOTT` schema. Now, the URL column in the table contains values that are pointing through HTTP to documents globally as well as pointing to virtual documents inside the database.

A `SELECT` on the column using the `getClob()` method would retrieve the results as a CLOB irrespective of where the document resides. This would retrieve values from the global HTTP address stored in the first row as well as the local `DBUri` reference.:

```
SELECT e.poURL.getClob() FROM uri_tab e;
```

Using HttpUriType and DBUriType

`HttpUriType` and `DBUriType` are subtypes of `UriType` and implement the functions for HTTP and `DBUri` references respectively.

Note: `HttpUriType` cannot store relative HTTP references in this release.

Example 12–14 DBUriType: Creating DBUri References

The following example creates a table with a column of type `DBUriType` and assigns a value to it.

```
CREATE TABLE DBUriTab(DBUri DBUriType, dbDocName VARCHAR2(2000));

-- insert values into it..!
INSERT INTO DBUriTab VALUES
  (sys.DBUriType.createUri('/ORADB/SCOTT/EMP/ROW[EMPNO=7369]'), 'empl');

INSERT INTO DBUriTab VALUES
  (sys.DBUriType('/SCOTT/EMP/ROW[EMPNO=7369]/', null);

-- access the references
SELECT e.DBUri.getCLOB() from DBUriTab e;
```

Creating Instances of UriType Objects with the UriFactory Package

The functions in the `UriFactory` package generate instances of the appropriate `UriType` subtype (`HttpUriType`, `DBUriType`, and `XDBUriType`). This way, you can avoid hardcoding the implementation in the program and handle whatever kinds of URI strings are used as input. See [Table 12–5](#).

The `getUri` method takes a string representing any of the supported kinds of URI and returns the appropriate subtype instance. For example:

- If the prefix starts with `http://`, `getUri` creates and returns an instance of a `SYS.HttpUriType` object.
- If the string starts with either `/oradb/` or `/dburi/`, `getUri` creates and returns an instance of a `SYS.DBUriType` object.
- If the string does not start with one of the prefixes noted in the preceding bullets, `getUri` creates and returns an instance of a `SYS.XDBUriType` object.

Note: The way UriFactory generates DBUriType instances has changed since Oracle9i release 1 (9.0.1):

In Oracle9i release 1 (9.0.1), any URL which did not start with one of the registered or standard prefixes such as `http://...` was mapped to a DBUriType by UriFactory.

In this release, you need to have a `/oradb` or `/dburi` prefix in order for UriFactory to generate a DBUriType. Otherwise it generates an XDBUriType.

Registering New UriType Subtypes with the UriFactory Package

The UriFactory package lets you register new UriType subtypes:

- Derive these types using the `CREATE TYPE` statement in SQL.
- Override the default methods to perform specialized processing when retrieving data, or to transform the XML data before displaying it.
- Pick a new prefix to identify URIs that use this specialized processing.
- Register the prefix using `UriFactory.registerURLHandler`, so that the UriFactory package can create an instance of your new subtype when it receives a URI starting with the new prefix you defined.

For example, you can invent a new protocol `ecom://` and define a subtype of UriType to handle that protocol. Perhaps the subtype implements some special logic for `getCLOB`, or does some changes to the XML tags or data within `getXML`. When you register the `ecom://` prefix with UriFactory, any calls to `UriFactory.getUri` generate the new subtype instance for URIs that begin with the `ecom://` prefix.

Table 12–5 UriFactory: Functions and Procedures

UriFactory Function	Description
escapeUri() MEMBER FUNCTION escapeUri() RETURN varchar2	Escapes the URL string by replacing the non-URL characters as specified in the Uri-ref specification by their equivalent escape sequence.
unescapeUri() FUNCTION unescapeUri() RETURN varchar2	Unescapes a given URL.
registerUriHandler() PROCEDURE registerUriHandler(prefix IN varchar2, schemaName in varchar2, typename in varchar2, ignoreCase in boolean := true, stripprefix in boolean := true)	Registers a particular type name for handling a particular URL. The type also implements the following static member function: STATIC FUNCTION createUri(url IN varchar2) RETURN <typename>; This function is called by getUrl() to generate an instance of the type. The stripprefix indicates that the prefix must be stripped off before calling the appropriate constructor for the type.
unRegisterUriHandler() PROCEDURE unregisterUriHandler(prefix in varchar2)	Unregisters a URL handler.

Example 12–15 UriFactory: Registering the ecom Protocol

Assume you are storing different kinds of URIs in a single table:

```
CREATE TABLE url_tab (urlcol varchar2(80));

-- Insert an HTTP URL
INSERT INTO url_tab VALUES ('http://www.oracle.com/');

-- Insert a database URI
INSERT INTO url_tab VALUES ('/oradb/SCOTT/EMPLOYEE/ROW[ENAME="Jack"]');

-- Create a new type to handle a new protocol called ecom://
CREATE 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 getUrl 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:// prefix.
begin
  -- register a new handler for ecom:// prefixes. The handler
  -- type name is ECOMUriTYPE, schema is SCOTT
  -- 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 the createUri function
  -- so that the string 'ecom:/' is not stored inside the
  -- ECOMUriTYPE object. (It is added back automatically when
  -- you call ECOMUriTYPE.getURL.)
  urifactory.registerURLHandler
  (
    prefix => 'ecom:/',
    schemaname => 'SCOTT',
    typename => 'ECOMURITYPE',
    ignoreprefixcase => true,
    stripprefix => true
  );
end;
/

-- Now the example inserts this new type of URI into the table.
insert into url_tab values ('ECOM://company1/company2=22/comp');

-- Use the factory to generate an instance of the appropriate
-- subtype for each URI in the table.
select urifactory.getUri(urlcol) from url_tab;

-- would now generate
HttpUriType('www.oracle.com'); -- a Http uri type instance

DBUriType('/oradb/SCOTT/EMPLOYEE/ROW[ENAME="Jack"],null); -- a DBUriType

EComUriType('company1/company2=22/comp'); -- an EComUriType instance
```

Why Define New Subtypes of UriType?

Deriving a new class for each protocol has these advantages:

- If you choose a subtype for representing a column, it provides an implicit constraint on the column to contain only instances of that protocol type. This

might be useful for implementing specialized indexes on that column for specific protocols. For example, for the `DBUri` you can implement some specialized indexes that can directly go and fetch the data from the disk blocks rather than executing SQL queries.

- Additionally, you can have different constraints on the columns based on the type involved. For instance, for the HTTP case, you could potentially define proxy and firewall constraints on the column so that any access through the HTTP would use the proxy server.

SYS_DBURIGEN() SQL Function

You can create an instance of `DBUriType` type by specifying the path expression to the constructor or the `UriFactory` methods. However, you also need methods to generate these objects dynamically, based on strings stored in table columns. You do this with the SQL function `SYS_DBURIGEN()`.

Example 12–16 *SYS_DBURIGEN(): Generating a URI of type DBUriType that points to a Column*

The following example uses `SYS_DBURIGEN()` to generate a URI of datatype `DBUriType` pointing to the email column of the row in the sample table `hr.employees` where the `employee_id = 206`:

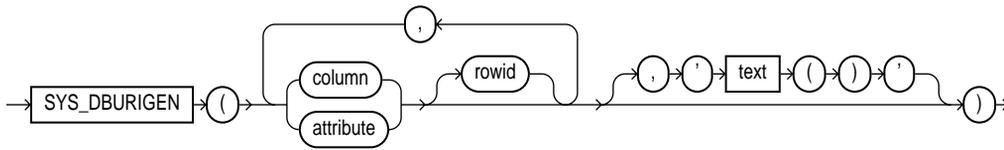
```
SELECT SYS_DBURIGEN(employee_id, email)
       FROM employees
       WHERE employee_id = 206;
```

```
SYS_DBURIGEN(EMPLOYEE_ID,EMAIL)(URL, SPARE)
```

```
-----
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID = "206"]/EMAIL', NULL)
```

`SYS_DBURIGEN()` takes as its argument one or more columns or attributes, and optionally a rowid, and generates a URI of datatype `DBUriType` to a particular column or row object. You can use the URI to retrieve an XML document from the database. The function takes an additional parameter to indicate if the text value of the node is needed. See [Figure 12–2](#).

Figure 12–2 SYS_DBURIGEN Syntax



All columns or attributes referenced must reside in the same table. They must reference a unique value. If you specify multiple columns, the initial columns identify the row in the database, and the last column identifies the column within the row.

By default, the URI points to a formatted XML document. To point only to the text of the document, specify the optional `text()` keyword.

See Also: *Oracle9i SQL Reference* for `SYS_DBURIGEN` syntax

If you do not specify an XML schema, Oracle interprets the table or view name as a public synonym.

Rules for Passing Columns or Object Attributes to SYS_DBURIGEN()

The column or attribute passed to the `SYS_DBURIGEN()` function must obey the following rules:

- **Unique mapping:** The column or object attribute must be uniquely mappable back to the table or view from which it comes. The only virtual columns allowed are the `VALUE` and `REF` operators. The column may come from a `TABLE()` subquery or an inline view, as long as the inline view does not rename the columns.
- **Key columns:** Either the `rowid` or a set of key columns must be specified. The list of key columns does not need to be declared as a unique or primary key, as long as the columns uniquely identify a particular row in the result.
- **Same table:** All columns referenced in the `SYS_DBURIGEN()` function must come from the same table or view.
- **PUBLIC element:** If the table or view pointed by the `rowid` or key columns does not have a database schema specified, then the `PUBLIC` keyword is used instead of the schema. When the `DBURI` is accessed, the table name resolves to the same table, synonym, or view that was visible by that name when the `DBURI` was created.

- **TEXT function:** `DBUri`, by default, retrieves an XML document containing the result. To retrieve only the text value, use the `text()` keyword as the final argument to the function.

For example:

```
select SYS_DBURIGEN(empno,ename,'text()') from scott.emp,
       WHERE empno=7369;
```

or you can just generates a URL of the form:

```
/SCOTT/EMP/ROW[EMPNO=7369]/ENAME/text()
```

- **Single-column argument:** If there is a single-column argument, the column is used both as the key column to identify the row and as the referenced column.

Example 12–17 Passing Columns With Single Arguments to SYS_DBURIGEN()

For example:

```
select SYS_DBURIGEN(empno) from emp
       WHERE empno=7369;
```

uses the `empno` both as the key column and the referenced column, generating a URL of the form:

```
/SCOTT/EMP/ROW[EMPNO=7369]/EMPNO,
```

for the row with `empno=7369`

SYS_DBURIGEN Examples

Example 12–18 Inserting Database References Using SYS_DBURIGEN()

```
CREATE TABLE doc_list_tab(docno number primary key, doc_ref SYS.DBUriType);
```

```
-- inserts /SCOTT/EMP/ROW[rowid='xxx']/EMPNO
INSERT INTO doc_list_tab values(1001,
    (select SYS_DBURIGEN(rowid,empno) from emp where empno = 100));
```

```
-- insert a Uri-ref to point to the ename column of emp!
INSERT INTO doc_list_tab values(1002,
    (select SYS_DBURIGEN(empno, ename) from emp where empno = 7369));
```

```
-- result of the DBURIGEN looks like, /SCOTT/EMP/ROW[EMPNO=7369]/ENAME
```

Returning Partial Results

When selecting the results of a large column, you might want to retrieve only a portion of the result and create a URL to the column instead. For example, consider the case of a travel story Web site. If all the 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 result page. Instead, you show the first 100 characters or gist of the story and then return a URL to the full story. This can be done as follows:

Example 12–19 *Returning a Portion of the Results By Creating a View and Using SYS_DBURIGEN()*

Assume that the travel story table is defined as follows:

```
CREATE TABLE travel_story
(
  story_name varchar2(100),
  story clob
);

-- insert some value..!
INSERT INTO travel_story values ('Egypt','This is my story of how I spent my
time in Egypt, with the pyramids in full view from my hotel room');
```

Now, you create a function that returns only the first 20 characters from the story:

```
create 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;
/
```

Now, you create a view that selects out only the first 100 characters from the story and then returns a DBUri reference to the story column:

```
CREATE VIEW travel_view as select story_name, charfunc(story) short_story,
  SYS_DBURIGEN(story_name,story,'text()') story_link
FROM travel_story;
```

Now, a SELECT from the view returns the following:

```
SELECT * FROM travel_view;

STORY_NAME          SHORT_STORY          STORY_LINK
```

```
-----
Egypt           This is my story of h
```

```
SYS.DBUriType('/PUBLIC/TRAVEL_STORY/ROW[STORY_NAME='Egypt']/STORY/text()')
```

RETURNING Uri-Refs

You can use `SYS_DBURIGEN()` in the `RETURNING` clause of DML statements to retrieve the URL of an object as it is inserted.

Example 12-20 Using SYS_DBURIGEN in the RETURNING Clause to Retrieve the URL of an Object

For example, consider the table `CLOB_TAB`:

```
CREATE TABLE clob_tab ( docid number, doc clob);
```

When you insert a document, you might want to store the URL of that document in another table, `URI_TAB`.

```
CREATE TABLE uri_tab (docs sys.DBUriType);
```

You can specify the storage of the URL of that document as part of the insert into `CLOB_TAB`, using the `RETURNING` clause and the `EXECUTE IMMEDIATE` syntax to execute the `SYS_DBURIGEN` function inside PL/SQL as follows:

```
declare
  ret sys.dburitype;
begin
  -- exucute the insert 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;
/
```

The URL created has the form:

```
/SCOTT/CLOB_TAB/ROW[DOCID="xxx"]/DOC/text()
```

Note: The `text()` keyword is appended to the end indicating that you want the URL to return just the CLOB value and not an XML document enclosing the CLOB text.

Turning a URL into a Database Query with DBUri Servlet

You can make table data accessible from your browser or any Web client, using the URI notation within a URL to specify the data to retrieve:

- Through `DBUri` servlet linked in with the database server.
- By writing your own servlet that runs on a servlet engine. The servlet can read the URI string from the path of the invoking URL, create a `DBUriType` object using that URI, call the `UriType` methods to retrieve the data, and return the values in the form of a Web page or an XML document.

Note: The Oracle servlet engine OSE is being desupported. Consequently the `oracle.xml.dburi.OraDBUriServlet` supported in Oracle9i Release 1 (9.0.1), is also being desupported. Use the `DBUri C-servlet` instead which uses the Oracle XML DB servlet system. See also [Chapter 20, "Writing Oracle XML DB Applications in Java"](#).

DBUri Servlet Mechanism

For the preceding methods, a servlet runs for accessing this information through HTTP. This servlet takes in a path expression following the servlet name as the `DBUri` reference and outputs the document pointed to by the `DBUri` to the output stream.

The generated document can be a Web page, an XML document, plain text, and so on. You can specify the MIME type so that the browser or other application knows what kind of content to expect:

- By default, the servlet can produce MIME types of `text/xml` and `text/plain`. If the URI ends in a `text()` function, then the `text/plain` MIME type is used, else an XML document is generated with the MIME type of `text/xml`.
- You can override the MIME type and set it to `binary/x-jpeg` or some other value using the `contentType` argument to the servlet.

Example 12–21 URL for Overriding the MIME Type by Generating the contentType Argument, to Retrieve the empno Column of Table Employee

For example, to retrieve the `empno` column of the `employee` table, you can write a URL such as one of the following:

```
-- Generates a contentType of text/plain
http://machine.oracle.com:8080/oradb/SCOTT/EMP/ROW[EMPNO=7369]/ENAME/text()
-- Generates a contentType of text/xml
http://machine.oracle.com:8080/oradb/SCOTT/EMP/ROW[EMPNO=7369/ENAME
```

where the machine `machine.oracle.com` is running the Oracle9i database, with a Web service at port 8080 listening to requests. `oradb` is the virtual path that maps to the servlet.

DBUri Servlet: Optional Arguments

Table 12–6 describes the three optional arguments you can pass to DBUri servlet to customize the output.

Table 12–6 DBUri Servlet: Optional Arguments

Argument	Description
rowsettag	Changes the default root tag name for the XML document. For example: <code>http://machine.oracle.com:8080/oradb/SCOTT/EMP?rowsettag=Employee</code>
contentType	Specifies the MIME type of the returned document. For example: <code>http://machine.oracle.com:8080/oradb/SCOTT/EMP?contentType=text/plain</code>
transform	This argument passes a URL to UriFactory, which in turn retrieves the XSL stylesheet at that location. This stylesheet is then applied to the XML document being returned by the servlet. For example: <code>http://machine.oracle.com:8080/oradb/SCOTT/EMP?transform=/oradb/SCOTT/XSLS/DOC/text()&contentType=text/html</code>

Note: When using XPath notation in the URL for this servlet, you may have to escape certain characters such as square brackets. You can use the `getExternalUrl()` functions in the `UriType` types to get an escaped version of the URL.

Note: In HTTP access, special characters such as], [, &, | have to be escaped using the %xx format, where xx is the hexadecimal number of the ASCII code for that character. Use the `getExternalUrl()` function in the `UriType` family to get an escaped version of the URL.

Installing DBUri Servlet

`DbUriServlet` is built into the database, and the installation is handled by the ORACLE XML DB configuration file. To customize the installation of the servlet, you need to edit it. You can edit the config file, `xdbconfig.xml` under the ORACLE XML DB user, through WebDAV, FTP, from Oracle Enterprise Manager, or in the database. To update the file using FTP or WebDAV, simply download the document, edit it as necessary, and save it back in the database. There are several things that can be customized using the configuration file.

See Also:

- [Chapter 20, "Writing Oracle XML DB Applications in Java"](#)
- [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)
- [Appendix A, "Installing and Configuring Oracle XML DB"](#)

Notice that the servlet is installed at `/oradb/*` specified in the `servlet-pattern` tag. The `*` is necessary to indicate that any path following `oradb` is to be mapped to the same servlet. The `oradb` is published as the virtual path. Here, you can change the path that will be used to access the servlet.

Example 12–22 *Installing DBUri Servlet Under /dburi/**

For example, to have the servlet installed under `/dburi/*`, you can run the following PL/SQL:

```
declare
  doc XMLType;
  doc2 XMLType;
begin
  doc := dbms_xdb.cfg_get();
  select updateXML(doc,
' /xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/servlet-mappings/servlet-mapping[servlet-name="DBUriServlet"]/servlet-pattern/text()'

```

```
, '/dburi/*') into doc2 from dual;
dbms_xdb.cfg_update(doc2);
commit;
end;
/
```

Security parameters, the servlet display-name, and the description can also be customized in the `xdbconfig.xml` configuration file. See [Appendix A, "Installing and Configuring Oracle XML DB"](#) and [Chapter 20, "Writing Oracle XML DB Applications in Java"](#). The servlet can be removed by deleting the servlet-pattern for this servlet. This can also be done using `updateXML()` to update the servlet-mapping element to null.

DBUri Security

Servlet security is handled by Oracle9i database using roles. When users log in to the servlet, they use their database username and password. The servlet will check to make sure the user logging in belongs to one of the roles specified in the configuration file. The roles allowed to access the servlet are specified in the `security-role-ref` tag. By default, the servlet is available to the special role **authenticatedUser**. Any user who logs into the servlet with any valid database username and password belongs to this role.

This parameter can be changed to restrict access to any role(s) in the database. To change from the default `authenticated-user` role to a role that you have created, say `servlet-users`, run:

```
declare
  doc XMLType;
  doc2 XMLType;
  doc3 XMLType;
begin
  doc := dbms_xdb.cfg_get();
  select updateXML(doc,
'/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/servlet-list/servlet[servlet-name="DBUriServlet"]/security-role-ref/role-name/text()'
, 'servlet-users') into doc2 from dual;
  select updateXML(doc2,
'/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/servlet-list/servlet[servlet-name="DBUriServlet"]/security-role-ref/role-link/text()'
, 'servlet-users') into doc3 from dual;
  dbms_xdb.cfg_update(doc3);
  commit;
end;
```

/

Configuring the UriFactory Package to Handle DBUris

The `UriFactory`, as explained in "[Creating Instances of UriType Objects with the UriFactory Package](#)" on page 12-25, takes a URL and generates the appropriate subtypes of the `UriType` to handle the corresponding protocol. For HTTP URLs, `UriFactory` creates instances of the `HttpUriType`. But when you have an HTTP URL that represents a URI path, it is more efficient to store and process it as a `DBUriType` instance in the database. The `DBUriType` processing involves fewer layers of communication and potentially fewer character conversions.

After you install `OraDBUriServlet`, so that any URL such as `http://machine-name/servlets/oradb/` gets handled by that servlet, you can configure the `UriFactory` to use that prefix and create instances of the `DBUriType` instead of `HttpUriType`:

```
begin
  -- register a new handler for the dburi prefix..
  urifactory.registerHandler('http://machine-name/servlets/oradb'
    , 'SYS', 'DBUriTYPE', true, true);
end;
```

/

After you execute this block in your session, any `UriFactory.getUri()` call in that session automatically creates an instance of the `DBUriType` for those HTTP URLs that have the prefix.

See Also: *Oracle9i XML API Reference - XDK and XDB* for details of all functions in `DBUriFactory` package.

Part V

Oracle XML DB Repository: Foldering, Security, and Protocols

Part V of this manual describes Oracle XML DB Repository. It includes how to version your data, implement and manage security, and how to use the associated Oracle XML DB APIs to access and manipulate Repository data.

Part V contains the following chapters:

- [Chapter 13, "Oracle XML DB Foldering"](#)
- [Chapter 14, "Oracle XML DB Versioning"](#)
- [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)
- [Chapter 16, "Oracle XML DB Resource API for PL/SQL \(DBMS_XDB\)"](#)
- [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#)
- [Chapter 18, "Oracle XML DB Resource Security"](#)
- [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#)
- [Chapter 20, "Writing Oracle XML DB Applications in Java"](#)

Oracle XML DB Foldering

This chapter describes how to access data in Oracle XML DB Repository using standard protocols such as FTP, HTTP/WebDAV and other Oracle XML DB Resource APIs. It also introduces you to using `RESOURCE_VIEW` and `PATH_VIEW` as the SQL mechanism for accessing and manipulating Repository data. It includes a table for comparing Repository operations through the various Resource APIs.

This chapter contains the following sections:

- [Introducing Oracle XML DB Foldering](#)
- [Oracle XML DB Repository](#)
- [Oracle XML DB Resources](#)
- [Accessing Oracle XML DB Repository Resources](#)
- [Navigational or Path Access](#)
- [Query-Based Access](#)
- [Accessing Repository Data Using Servlets](#)
- [Accessing Data Stored in Oracle XML DB Repository Resources](#)
- [Managing and Controlling Access to Resources](#)
- [Extending Resource Metadata Properties](#)

Introducing Oracle XML DB Foldering

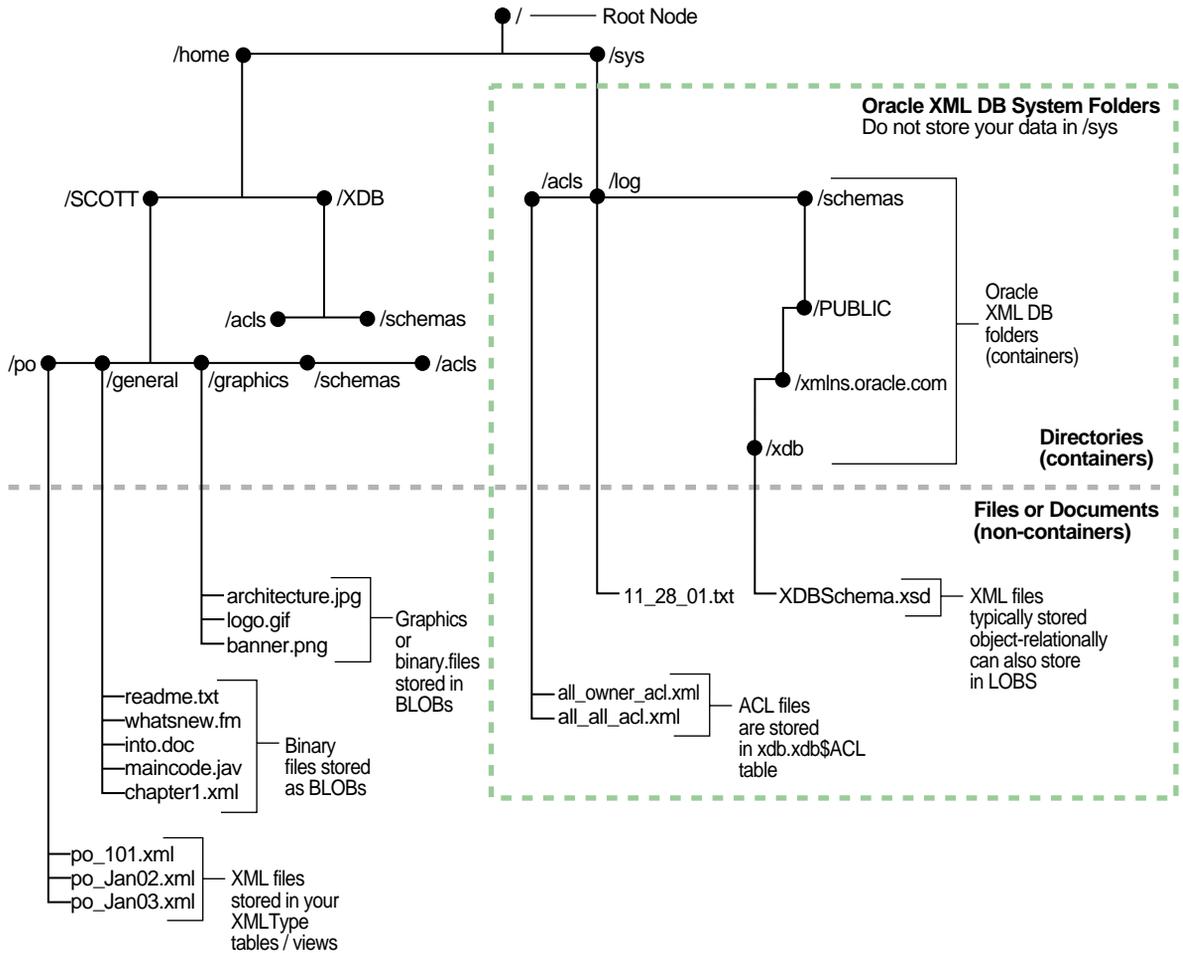
Using the foldering feature in Oracle XML DB you can store content in the database in hierarchical structures, as opposed to traditional relational database structures.

[Figure 13–1](#) 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.

Foldering allows applications to access hierarchically indexed content in the database using the FTP, HTTP, and WebDAV protocol standards as if the database content is stored in a file system.

This chapter provides an overview of how to access data in Oracle XML DB Repository folders using the standard protocols. There are other APIs available in this release, which allow you to access the Repository object hierarchy using Java, SQL, and PL/SQL.

Figure 13–1 A Typical Folder Tree Showing Hierarchical Structures in Oracle XML Repository



Caution: The directory `/sys` is used by Oracle XML DB to maintain system-defined XML schemas, ACLs, and so on. In general:

- Do not store any data under the `/sys` directory.
- Do not modify any content in the `/sys` directory.

See Also:

- [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)
- [Chapter 16, "Oracle XML DB Resource API for PL/SQL \(DBMS_XDB\)"](#)
- [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#)
- [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#)

Oracle XML DB Repository

Oracle XML DB Repository (Repository) is the set of database objects, across all XML and database schemas, that are mapped to path names. It is a connected, directed, acyclic *graph* of resources with a single root node (/). Each resource in the graph has one or more associated path names.

The Repository can be thought of as a file system of objects rather than files.

Note: The Repository supports multiple links to a given resource.

Repository Terminology

The following lists describes terms used in Oracle XML DB Repository:

- **Resource:** A resource is any object or node in the hierarchy. Resources are identified by URLs. See "[Oracle XML DB Resources](#)" on page 6.
- **Folder:** A folder is a node (or directory) in the hierarchy that can contain a collection of resources. A folder is also a resource.
- **Pathname:** A hierarchical name composed of a root element (the first /), element separators /, and various subelements (or path elements). A path element may be composed of any character in the database character set except \ and /, which have special meanings in Oracle XML DB. The forward slash is the default name separator in a path name, and the backward slash is used to escape characters. The Oracle XML DB configuration file `xdbconfig.xml` also contains a list of user-defined characters that may not appear within a path name (`<invalid-pathname-chars>`).
- **Resource or Link name:** The name of a resource within its parent folder. Resource names must be unique (case-sensitive in this release) within a folder. Resource names are always in the UTF8 character set (NVARCHAR).

- **Contents:** The body of a resource, what you get when you treat the resource like a file and ask for its contents. Contents is always an `XMLType`.
- **XDBBinary:** An XML element defined by the Oracle XML DB schema that contains binary data. `XDBBinary` elements are stored in the Repository when unstructured binary data is uploaded into Oracle XML DB.
- **Access Control List (ACL):** Restricts access to a resource or resources. Oracle XML DB uses ACLs to restrict access to any Oracle XML DB resource (any `XMLType` object that is mapped into the Oracle XML DB file system hierarchy).

See Also: [Chapter 18, "Oracle XML DB Resource Security"](#)

Many terms used by Oracle XML DB have common synonyms used in other contexts, as shown in [Table 13-1](#).

Table 13-1 *Synonyms for Oracle XML DB Foldering Terms*

Synonym	Oracle XML DB Foldering 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
Context	Folder	JNDI

Oracle XML DB Resources

Oracle XML DB resources conform to the `xdbresource.xml` schema, which is defined by Oracle XML DB. 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.

Contents Element in Resource Index

A resource index has a special element called `Contents` which contains the contents of the resource.

any Element

The XML schema for a resource also defines an `any` element, with `maxOccurs` unbounded, which allowed to contain any element outside the Oracle XML DB XML namespace. This way, arbitrary instance-defined properties can be associated with the resource.

Where Exactly Is Repository Data Stored?

Oracle XML DB stores Repository data in a set of tables and indexes in the Oracle XML DB database schema. These tables are accessible. If you register an XML schema and request the tables be generated by Oracle XML DB, the tables are created in your database schema. This means that you are able to see or modify them. However, other users will not be able to see your tables unless you explicitly grant them permission to do so.

Generated Table Names

The names of the generated tables are assigned by Oracle XML DB and can be obtained by finding the `xdb:defaultTable=XXX` attribute in your XML schema document (or the default XML schema document). When you register an XML schema, you can also provide your own table name, and override the default created by Oracle XML DB.

See Also: ["Guidelines for Using Registered XML Schemas"](#) on page 5-14 in [Chapter 5, "Structured Mapping of XMLType"](#)

Defining Structured Storage for Resources

Applications that need to define structured storage for resources can do so by either:

- Subclassing the Oracle XML DB resource type. Subclassing Oracle XML DB resources requires privileges on the table `XDB$RESOURCE`.
- Storing data conforming to a visible, registered XML schema.

See Also: [Chapter 5, "Structured Mapping of XMLType"](#).

Pathname Resolution

The data relating a folder to its children is managed by the Oracle XML DB hierarchical index. This provides a fast mechanism for evaluating path names, similar to the directory mechanisms used by operating-system file systems.

Resources that are folders have the `Container` attribute 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, he receives an `access denied` error. Folder listings and other queries will not return a row when the `read-properties` privilege is denied on its resource.

Note: Error handling in path name resolution differentiates between invalid resource names and resources that are not folders for compatibility with file systems. Since Oracle XML DB resources are accessible from outside the Repository (using SQL), denying read access on a folder that contains a resource will *not* prevent read access to that resource.

Deleting Resources

Deletion of a link deletes the resource pointed to by the link if and only if that was the last link to the resource and the resource is not versioned. Links in Oracle XML DB Repository are analogous to Unix “hard links”.

See Also: ["Deleting Repository Resources: Examples"](#) on page 15-10

Accessing Oracle XML DB Repository Resources

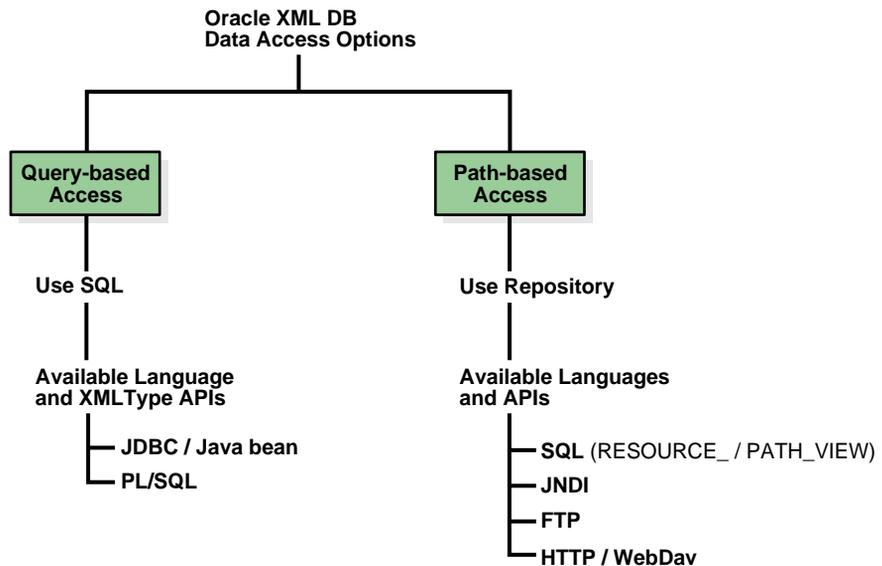
Oracle XML DB provides two techniques for accessing resources:

- *"Navigational or Path Access"* on page 13-9. Navigational/path access to content in Oracle XML DB is achieved using a hierarchical index of objects or resources. Each resource has one or more unique path names that reflect its location in the hierarchy. You can use navigational access to reference any object in the database without regard to its location in the tablespace.
- *"Query-Based Access"* on page 13-12. SQL access to the Repository is done using a set of views that expose resource properties and path names and map hierarchical access operators onto the Oracle XML DB schema.

Figure 13-2 illustrates Oracle XML DB data access options. A high level discussion of which data access option to select is described in Chapter 2, "Getting Started with Oracle XML DB", "Oracle XML DB Application Design: b. Access Models" on page 13-4.

See Also: Table 13-3, "Accessing Oracle XML DB Repository: API Options"

Figure 13-2 Oracle XML DB Repository Data Access Options



A Uniform Resource Locator (URL) is used to access an Oracle XML DB resource. A URL includes the hostname, protocol information, path name, and resource name of the object.

Navigational or Path Access

Oracle XML DB folders support the same protocol standards used by many operating systems. This allows an Oracle XML DB folder to function just like a native folder or directory in supported operating-system environments. For example: you can:

- Use Windows Explorer to open and access Oracle XML DB folders and resources the same way you access other directories or resources in the Windows NT file system, as shown in [Figure 13-3](#).
- Access Repository data using HTTP/WebDAV from an Internet Explorer browser, such as when viewing Web Folders, as shown in [Figure 13-4](#).

Figure 13-3 Oracle XML DB Folders in Windows Explorer

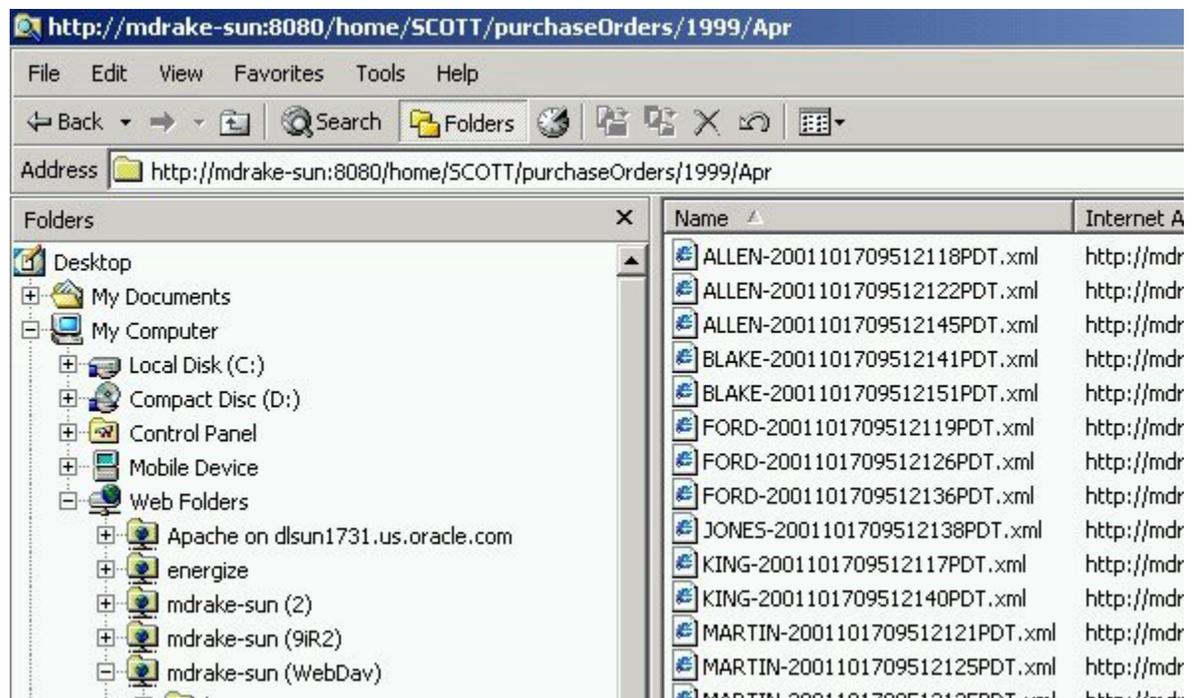
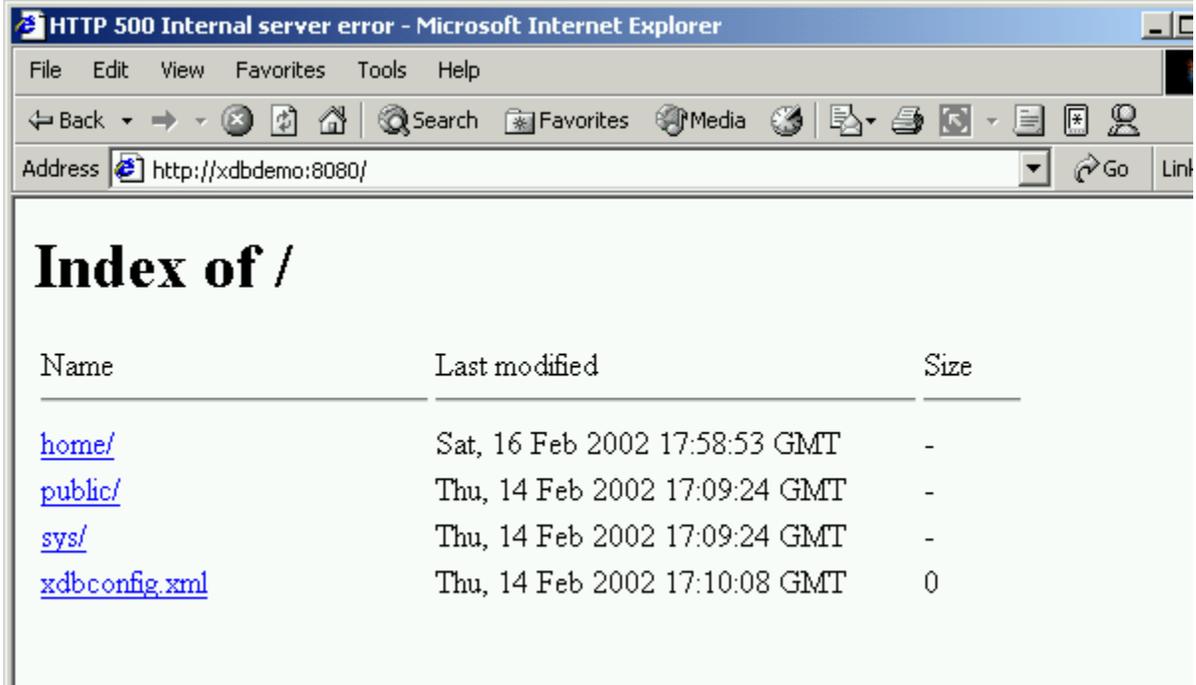


Figure 13–4 Accessing Repository Data Using HTTP/WebDAV and Navigational Access From IE Browser: Viewing Web Folders



Accessing Oracle XML DB Resources Using Internet Protocols

Oracle Net Services provides one way of accessing database resources. Oracle XML DB support for Internet protocols provides another way of accessing database resources.

Where You Can Use Oracle XML DB Protocol Access

Oracle Net Services is optimized for record-oriented data. Internet protocols are designed for stream-oriented data, such as binary files or XML text documents.

Oracle XML DB protocol access is a valuable alternative to Net Services in the following scenarios:

- Direct database access from file-oriented applications using the database like a file system

- Heterogeneous application server environments that want 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 want data as XML text
- Web applications using client-side XSL to format datagrams not needing much application processing
- Web applications using Java servlets running inside the database
- Web access to XML-oriented stored procedures

Protocol Access Calling Sequence

Protocol access uses the following steps in Oracle XML DB:

1. A connection object is established, and the protocol may decide to read part of the request.
2. The protocol decides if the user is already authenticated and wants to reuse an existing session or if the connection must be reauthenticated (generally the case).
3. An existing session is pulled from the session pool, or a new one is created.
4. If authentication has not been provided and the request is HTTP *Get* or *Head*, the session is run as the `ANONYMOUS` user. If the session has already been authenticated as the `ANONYMOUS` user, there is no cost to reuse the existing session. If authentication has been provided, the database reauthentication routines are used to authenticate the connection.
5. The request is parsed.
6. If the requested path name maps to a servlet (for HTTP only), the servlet is invoked using the Java VM. The servlet code writes out the response to a response stream or asks `XMLType` instances to do so.

Retrieving Oracle XML DB Resources

When the protocol indicates that a resource is to be retrieved, the path name to the resource is resolved. Resources being fetched are always streamed out as XML, with the exception of resources containing the `XDBBinary` element, an element defined to be the XML binary data type, which have their contents streamed out in RAW form.

Storing Oracle XML DB Resources

When the protocol indicates that a resource must be stored, Oracle XML DB checks the document's file name extension for .xml, .xsl, .xsd, and so on. If the document is XML, a pre-parse step is done, where enough of the resource is read to determine the XML `schemaLocation` and `namespace` of the root element in the document. This location is used to look for a registered schema with that `schemaLocation` URL. If a registered schema is located with a definition for the root element of the current document, the default table specified for that element is used to store that resource's contents.

Using Internet Protocols and XMLType: XMLType Direct Stream Write

Oracle XML DB supports Internet protocols at the `XMLType` level by using the `writeToStream()` Java method on `XMLType`. This method is natively implemented and writes `XMLType` data directly to the protocol request stream. This avoids the overhead of converting database data through Java datatypes, creating Java objects, and Java VM execution costs, resulting in significantly higher performance. This is especially the case if the Java code deals with XML element trees only close to the root, without traversing too many of the leaf elements, hence minimizing the number of Java objects created.

See Also: [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#)

Query-Based Access

Oracle XML DB provides two Repository views to enable SQL access to Repository data:

- `PATH_VIEW`
- `RESOURCE_VIEW`

[Table 13–2](#) summarizes the differences between `PATH_VIEW` and `RESOURCE_VIEW`.

Table 13–2 Differences Between `PATH_VIEW` and `RESOURCE_VIEW`

<code>PATH_VIEW</code>	<code>RESOURCE_VIEW</code>
Contains link properties	No link properties
Contains resource properties and path name	Contains resource properties and path name
Has one row for each unique path in the Repository	Has one row for each resource in the Repository

Note: Each resource can have multiple paths.

The single path in the `RESOURCE_VIEW` is arbitrarily chosen from among the many possible paths that refer to a resource. Oracle XML DB provides operators like `UNDER_PATH` that enable applications to search for resources contained (recursively) within a particular folder, get the resource depth, and so on. Each row in the views is of `XMLType`.

DML on the Oracle XML DB Repository views can be used to insert, rename, delete, and update resource properties and contents. Programmatic APIs must be used for other operations, such as creating links to existing resources.

See Also:

- [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#) for details on SQL Repository access
- [Chapter 18, "Oracle XML DB Resource Security"](#)
- *Oracle9i XML API Reference - XDK and Oracle XML DB*

Accessing Repository Data Using Servlets

Oracle XML DB implements the Java servlet API, version 2.2, with the following exceptions:

- All servlets must be distributable. They must expect to run in different VMs.
- WAR and `web.xml` files are not supported. Oracle XML DB supports a subset of the XML configurations in this file. An XSL 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.

See Also: [Chapter 20, "Writing Oracle XML DB Applications in Java"](#)

Accessing Data Stored in Oracle XML DB Repository Resources

The three main ways you can access data stored in Oracle XML DB Repository resources are through:

- Oracle XML DB Resource APIs for Java/JNDI
- A combination of Oracle XML DB Resource Views API and Oracle XML DB Resource API for PL/SQL
- Internet protocols (HTTP/WebDAV and FTP) and Oracle XML DB Protocol Server

[Table 13–3](#) lists common Oracle XML DB Repository operations and describes how these operations can be accomplished using each of the three methods. The table shows functionality common to three methods. Note that not all the methods are equally suited to a particular set of tasks.

See Also:

- [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)
- [Chapter 16, "Oracle XML DB Resource API for PL/SQL \(DBMS_XDB\)"](#)
- [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#)
- [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#)
- *Oracle9i XML API Reference - XDK and Oracle XML DB*

Table 13–3 Accessing Oracle XML DB Repository: API Options

Data Access Operation	Path-Based Access: Resource API for Java/JNDI	Query-Based Access: RESOURCE_VIEW API Path-Based Access: Resource API for PL/SQL	Path-Based Access: Protocols
Creating a resource	<code>r = new Resource();</code> <code>jndictx.bind(String oldpath, r)</code>	INSERT INTO PATH_VIEW VALUES (path, res, linkprop) See also DBMS_XDB.CreateResource.	HTTP PUT; FTP PUT
Updating contents of a resource using path name	<code>r = jndictx.lookup(String path);</code> <code>r.setContents(InputStream s);</code> <code>r.save()</code>	UPDATE RESOURCE_VIEW SET resource = updateXML(res, '/Resource/Contents', lob) WHERE EQUALS_PATH(res, :path) > 0	HTTP PUT; FTP PUT
Updating properties of a resource by path name	<code>r = jndictx.lookup(String path)</code> <code>r.setXXX(Object val); ... r.save()</code>	UPDATE RESOURCE_VIEW SET resource = updateXML(res, '/Resource/propname', newval, '/Resource/propname2', newval2, ...) WHERE EQUALS_PATH(res, :path) > 0	HTTP PROPPATCH; FTP N/A
Updating the ACL of a resource	<code>r = jndictx.lookup(String path);</code> <code>r.setACL(XMLType acl); ... r.save()</code>	UPDATE RESOURCE_VIEW SET resource = updateXML(res, '/Resource/ACL', XMLType) WHERE EQUALS_PATH(res, :path) > 0	N/A
Unlinking a resource, deleting it if it is the last link	<code>r=jndictx.unbind(String path);</code>	DELETE FROM RESOURCE_VIEW WHERE EQUALS_PATH(res, :path) > 0	HTTP DELETE; FTP DELETE
Forcibly removing all links to a resource	N/A	DELETE FROM PATH_VIEW WHERE extractValue(res,'display_name') = 'My resource';	N/A FTP: quote rm-rf<resource>
Moving a resource or folder	<code>r = jndictx.rename(String oldpath, String newpath);</code>	UPDATE PATH_VIEW SET path = newpath WHERE EQUALS_PATH(res, :path) > 0	WebDAV MOVE; FTP RENAME
Copying a resource or folder	<code>r = jndictx.lookup(String oldpath);</code> <code>r.clone(); r = jndictx.bind(r);</code>	INSERT INTO PATH_VIEW SELECT:::newpath, resource, link FROM PATH_VIEW WHERE EQUALS_PATH(res,:oldpath)> 0	WebDav COPY; FTP N/A
Creating a link to an existing resource	<code>r = jndictx.lookup(String oldpath);</code> <code>jndictx.bind(r);</code>	Call dbms_xdb.Link(srcpath IN VARCHAR2, linkfolder IN VARCHAR2, linkname IN VARCHAR2);	N/A
Getting binary/text representation of resource contents by path name	<code>r=jndictx.lookup(String path);</code> <code>r.getContents();</code>	SELECT GET_CONTENTS(resource) FROM RESOURCE_VIEW p WHERE EQUALS_PATH(res, :path) > 0	HTTP GET; FTP GET

Table 13–3 Accessing Oracle XML DB Repository: API Options (Cont.)

Data Access Operation	Path-Based Access: Resource API for Java/JNDI	Query-Based Access: RESOURCE_VIEW API	Path-Based Access: Resource API for PL/SQL	Path-Based Access: Protocols
Getting XMLType representation of resource contents by path name	N/A	SELECT extract(res, '/Resource/Contents/*') FROM RESOURCE_VIEW p WHERE EQUALS_PATH(Res, :path) > 0		N/A
Getting resource properties by path name	r = jndictx.lookup(String path); r.getXXX(); ...	SELECT extractValue(res, '/Resource/XXX') FROM RESOURCE_VIEW WHERE EQUALS_PATH(res, :path) > 0		WebDAV PROPFIND (depth = 0); FTP N/A
Listing a directory	jndictx.listBindings(String path);	SELECT * FROM PATH_VIEW WHERE UNDER_PATH(res, :path, 1) > 0		WebDAV PROPFIND (depth=1); FTP LS
Creating a folder	jndictx.bind(String path);	ICall dbms_xdb.createFolder(VARCHAR2)		WebDAV MKCOL; FTP MKDIR
Unlinking a folder	jndictx.unbind(String path);	DELETE FROM PATH_VIEW WHERE EQUALS_PATH(res, :path) > 0		HTTP DELETE; FTP RMDIR
Forcibly deleting a folder and all links to it	jndictx.destroySubcontext()	Call dbms_xdb.deleteFolder(VARCHAR2)		N/A FTP: quote rm.rf <folder>
Getting a resource with a row lock	xdb.jndictx.lookup(String path, boolean rowLocked, null);	SELECT ... FROM RESOURCE_VIEW FOR UPDATE ...;		N/A
Putting a WebDAV lock on the resource	r = jndictx.lookup(String path); r.lock(int lock_type);	DBMS_XDB.lock(lock_type INTEGER);		WebDAV LOCK; FTP: QUOTE LOCK
Removing a WebDAV lock	r = jndictx.lookup(String path); r.unlock(int lock_type);	DBMS_XDB.unlock(path VARCHAR2);		WebDAV UNLOCK; QUOTE UNLOCK
Committing changes	java.sql.Connection.commit();	COMMIT;		Automatically commits at the end of each request
Rollback changes	java.sql.Connection.rollback();	ROLLBACK;		N/A

Managing and Controlling Access to Resources

You can set access control privileges on Oracle XML DB folders and resources.

See Also:

- [Chapter 18, "Oracle XML DB Resource Security"](#) for more detail on using access control on Oracle XML DB folders
- [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)
- *Oracle9i XML API Reference - XDK and Oracle XML DB* the chapters on DBMS_XDB

Extending Resource Metadata Properties

Oracle XML DB resources are described by XML schema, `XDBResource.xsd`. This XML schema is described in [Appendix G, "Example Setup scripts. Oracle XML DB-Supplied XML Schemas"](#). `XDBResource.xsd` declares a fixed set of metadata properties such as the `Owner`, `CreationDate`, and so on. You can specify values for these metadata attributes while creating or updating resources.

You have two options for storing proprietary (custom) tags as extra metadata with resources, that is metadata properties that are not defined by the Resource XML schema:

- **Option 1: Storing extra metadata as a CLOB (`ResExtra` element).** The default schema has a top-level `any` element (declared with `maxOccurs="unbounded"`), thus allowing any valid XML data as part of the resource document, and gets stored in the `RESEXTRA` CLOB column:

Note: The XML representing user level metadata must be within a namespace other than that `XDBResource` namespace.

```
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
  <Owner>SCOTT</Owner>
  ... <!-- other system defined metadata -->
  <!-- User Metadata (appearing within different namespace) -->
  <ResExtra>
    <myCustomAttrs xmlns="http://www.example.com/customattr">
      <attr1>value1</attr1>
      <attr2>value2</attr2>
```

```

        </myCustomAttrs>
    </ResExtra>
    <!-- contents of the resource>
    <Contents>
        ...
    </Contents>
</Resource>

```

Though this approach works well for ad-hoc extensions to the resource metadata, the queryability and updatability of the user metadata is impacted because the user metadata is stored in a CLOB.

- Option 2: Extending the Resource XML Schema.** You can extend the resource XML schema using the techniques specified by the XML schema specifications, that is, you can register a new XML schema that extends the `ResourceType` complexType. This triggers the creation of object subtypes under the `XDB$RESOURCE_T` object type, thereby adding new columns to the `XDB$RESOURCE` table:

```

<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xdbres="http://xmlns.oracle.com/xdb/XDBResource.xsd"
  <complexType name="myCustomResourceType">
    <complexContent>
      <extension base="xdbres:ResourceType">
        <element name="custom-attr1" type="string">
          ...
        </extension>
      </complexContent>
    </complexType>
  </schema>

```

Oracle XML DB Versioning

This chapter describes how to create and manage versions of Oracle XML DB resources. It contains the following sections:

- [Introducing Oracle XML DB Versioning](#)
- [Creating a Version-Controlled Resource \(VCR\)](#)
- [Access Control and Security of VCR](#)
- [Frequently Asked Questions: Oracle XML DB Versioning](#)

Introducing Oracle XML DB Versioning

Oracle XML DB versioning provides a way to create and manage different versions of a resource in Oracle XML DB. In previous releases of Oracle9i database, after a resource (table, column,...) is updated, its previous contents and properties are lost. Oracle XML DB versioning prevents this loss by storing a version of the resource in the database to keep the old resource contents and properties when an update is issued.

Oracle XML DB provides a PLSQL package, `DBMS_XDB_VERSION` to put a resource under version-control and retrieve different versions of the resource.

Oracle XML DB Versioning Features

Oracle XML DB versioning helps keep track of all changes on version-controlled Oracle XML DB resources (VCR). The following sections discuss these features in detail. Oracle XML DB versioning features include the following:

- *Version control on a resource.* You have the option to turn on or off version control on an Oracle XML DB resource. See "[Creating a Version-Controlled Resource \(VCR\)](#)".
- *Updating process of a version-controlled resource.* When Oracle XML DB updates a version-controlled resource, it also creates a new version of the resource, and this version will not be deleted from the database when the version-controlled resource is deleted by you. See "[Updating a Version-Controlled Resource \(VCR\)](#)".
- *Loading a version-controlled resource is similar to loading any other regular resource* in Oracle XML DB using the path name. See "[Creating a Version-Controlled Resource \(VCR\)](#)".
- *Loading a version of the resource.* To load a version of a resource, you must first find the resource object id of the version and then load the version using that id. The resource object id can be found from the resource version history or from the version-controlled resource itself. See "[Oracle XML DB Resource ID and Pathname](#)".

Note: In this release, Oracle XML DB versioning supports version control for Oracle XML DB resources. It does not support version control for user-defined tables or data in Oracle9i database.

See Also: *Oracle9i XML API Reference - XDK and XDB*

Oracle XML DB Versioning Terms Used in This Chapter

Table 14–1 lists the Oracle XML DB versioning terms used in this chapter.

Table 14–1 Oracle XML DB Versioning Terms

Oracle XML DB Versioning Term	Description
Version control	When a record or history of all changes to an Oracle XML DB resource is stored and managed, the resource is said to be put under version control.
Versionable resource	Versionable resource is an Oracle XML DB resource that can be put under version control.
Version-controlled resource (VCR).	Version-controlled resource is an Oracle XML DB resource that is put under version control. Here, a VCR is a reference to a version Oracle XML DB resource. It is not physically stored in the database.
Version resource.	Version resource is a version of the Oracle XML DB resource that is put under version control. Version resource is a read-only Oracle XML DB resource. It cannot be updated or deleted. However, the version resource will be removed from the system when the version history is deleted from the system.
Checked-out resource.	It is an Oracle XML DB resource created when version-controlled resource is checked out.
Checkout, checkin, and uncheckout.	These are operations for updating Oracle XML DB resources. Version-controlled resources must be checked out before they are changed. Use the checkin operation to make the change permanent. Use uncheckout to void the change.

Oracle XML DB Resource ID and Pathname

Oracle XML DB resource ID is a unique system-generated ID for an Oracle XML DB resource. Here Resource ID helps identify resources that do not have path names. For example, version resource is a system-generated resources and does not have a path name. The function `GetResourceByResId()` can be used to retrieve resources given the resource object ID.

Example 14–1 DBMS_XDB_VERSION. GetResourceByResId(): First version ID is Returned When Resource 'home/index.html' is makeversioned

```
declare
resid DBMS_XDB_VERSION.RESID_TYPE;
res XMLType;
begin
```

```
resid := DBMS_XDB_VERSION.MakeVersioned('/home/SCOTT/versample.html');
-- Obtain the resource
res := DBMS_XDB_VERSION.GetResourceByResId(resid);
```

Creating a Version-Controlled Resource (VCR)

Oracle XML DB does not automatically keep a history of updates since not all Oracle XML DB resources need this. You must send a request to Oracle XML DB to put an Oracle XML DB resource under version control. In this release, all Oracle XML DB resources are versionable resources except for the following:

- Folders (directories or collections)
- ACL
- Resources having XML schema-based contents

When a VCR is created the first version resource of the VCR is created, and the VCR is a reference to the newly-created version.

See "[Version Resource or VCR Version](#)" on page 14-4.

Example 14–2 DBMS_XDB_VERSION.makeVersioned(): Creating a Version-Controlled Resource (VCR)

Resource 'home/SCOTT/versample.html' is turned into a version-controlled resource.

```
declare
resid DBMS_XDB_VERSION.RESID_TYPE;
begin
resid := DBMS_XDB_VERSION.MakeVersioned('/home/SCOTT/versample.html');
end;
```

`MakeVersioned()` returns the resource ID of the very first version of the version-controlled resource. This version is represented by a resource ID, which is discussed in "[Resource ID of a New Version](#)" on page 14-5.

`MakeVersioned()` is not an auto-commit SQL operation. You have to commit the operation.

Version Resource or VCR Version

Oracle XML DB does not provide path name for version resource. However, it provides a version resource ID.

Version resources are read-only resources.

The version ID is returned by a couple of methods in package `DBMS_XDB_VERSION`, which are described in the following sections.

Resource ID of a New Version

When a VCR is checked in, a new version resource is created, and its resource ID is returned to the you. Because the VCR is just a reference to this new version resource, the resource ID of the version can also be found by calling method `DBMS_XDB.getResourceID()` whose input is the VCR path name.

Example 14–3 Retrieving the Resource ID of the New Version After Check In

The following example shows how to get the resource id of the new version after checking in `/home/index.html`:

```
-- Declare a variable for resource id
declare
resid DBMS_XDB_VERSION.RESID_TYPE;
res XMLType;
begin
-- Get the id as user checks in.
resid := DBMS_XDB_VERSION.checkin('/home/SCOTT/versample.html');

-- Obtain the resource
res := DBMS_XDB_VERSION.GetResourceByResId(resid);
end;
/
```

Example 14–4 Oracle XML DB: Creating and Updating a Version-Controlled Resource (VCR)

```
-- Variable definitions.
declare
resid1 DBMS_XDB_VERSION.RESID_TYPE;
resid2 DBMS_XDB_VERSION.RESID_TYPE;
begin
-- Put a resource under version control.
resid1 := DBMS_XDB_VERSION.MakeVersioned('/home/SCOTT/versample.html');

-- Checkout to update contents of the VCR
DBMS_XDB_VERSION.Checkout('/home/SCOTT/versample.html');

-- Use resource_view to update versample.html
```

```
update resource_view
set res = sys.xmltype.createxml(
'<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.oracle.com/xdb/XDBResource.xsd
http://xmlns.oracle.com/xdb/XDBResource.xsd">
<Author>Jane Doe</Author>
<DisplayName>versample</DisplayName>
<Comment>Has this got updated or not ?? </Comment>
<Language>en</Language>
<CharacterSet>ASCII</CharacterSet>
<ContentType>text/plain</ContentType>
</Resource>')
where any_path = '/home/SCOTT/versample.html';

-- Checkin the change
resid2 := DBMS_XDB_VERSION.Checkin('/home/SCOTT/versample.html');
end;
/

-- At this point, you can download the first version with the resource object
ID,
-- resid1 and download the second version with resid2.

-- Checkout to delete the VCR
DBMS_XDB_VERSION.Checkout('/home/SCOTT/versample.html');

-- Delete the VCR
delete from resource_view where any_path = '/home/SCOTT/versample.html';

-- Once the delete above is done, any reference
-- to the resource (that is, checkin, checkout, and so on, results in
-- ORA-31001: Invalid resource handle or path name "/home/SCOTT/versample.html"
```

Accessing a Version-Controlled Resource (VCR)

VCR also has a path name as any regular resource. Accessing a VCR is the same as accessing any other resources in Oracle XML DB.

Updating a Version-Controlled Resource (VCR)

The operations on regular Oracle XML DB resources do not require the VCR to be checked-out. Updating a VCR requires more steps than for a regular Oracle XML DB resource:

Before updating the contents and properties of a VCR, check out the resource. The resource must be checked in to make the update permanent. All of these operations are not auto-commit SQL operations. You must explicitly commit the SQL transaction. Here are the steps to update a VCR:

1. *Checkout a resource.* To checkout a resource, the path name of the resource must be passed to Oracle XML DB.
2. *Update the resource.* You can update either the contents or the properties of the resource. These features are already supported by Oracle XML DB. A new version of a resource is not created until the resource is checked in, so an update or deletion is not permanent until after a checkin request for the resource is done.
3. *Checkin or uncheckout a resource.* When a VCR is checked in, a new version is created, and the VCR has the same contents and properties as the new version. When a VCR is unchecked out, the VCR is unchanged. That is, it will have the same contents and properties with the old version. To checkin or uncheckout a resource, the path name of the resource must be passed to Oracle XML DB. If the path name has been updated since checkout, the new path name must be used. It is illegal to use the old path name.

Checkout

In Oracle9i Release 2 (9.2), the VCR checkout operation is executed by calling `DBMS_XDB_VERSION.CheckOut()`. If you want to commit an update of a resource, it is a good idea to commit after checkout. If you do not commit right after checking out, you may have to rollback your transaction at a later point, and the update is lost.

Example 14-5 VCR Checkout

For example:

```
-- Resource '/home/SCOTT/versample.html' is checked out.  
DBMS_XDB_VERSION.CheckOut('/home/SCOTT/versample.html');
```

Checkin

In Oracle9i Release 2 (9.2), the VCR checkin operation is executed by calling `DBMS_XDB_VERSION.CheckIn()`. Checkin takes the path name of a resource. This path name does not have to be the same as the path name that was passed to checkout, but the checkin and checkout path names must be of the same resource.

Example 14–6 VCR Checkin

For example:

```
-- Resource '/home/SCOTT/versample.html' is checked in.
declare
resid DBMS_XDB_VERSION.RESID_TYPE;
begin
    resid := DBMS_XDB_VERSION.CheckIn('/home/SCOTT/versample.html');
end;
/
```

Uncheckout

In Oracle9i Release 2 (9.2), uncheckout is executed by calling `DBMS_XDB_VERSION.UncheckOut()`. This path name does not have to be the same as the path name that was passed to `checkout`, but the checkin and checkout path names must be of the same resource.

Example 14–7 VCR Uncheckout

For example:

```
-- Resource '/home/SCOTT/versample.html' is unchecked out.
declare
resid DBMS_XDB_VERSION.RESID_TYPE;
begin
    resid := DBMS_XDB_VERSION.UncheckOut('/home/SCOTT/versample.html');
end;
/
```

Update Contents and Properties

After checking out a VCR, all Oracle XML DB user interfaces for updating contents and properties of a regular resource can be applied to a VCR.

See Also: [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#) for details on updating an Oracle XML DB resource.

Access Control and Security of VCR

Access control on VCR and version resource is the same as for a regular resource. Whenever you request access to these resources, ACL is checked.

See Also: [Chapter 18, "Oracle XML DB Resource Security"](#)

Version Resource

When a regular resource is `makeversioned`, the first version resource is created, and the ACL of this first version is the same as the ACL of the original resource. When a checked-out resource is checked in, a new version is created, and the ACL of this new version is exactly the same as the ACL of the checked-out resource. After version resource is created, its ACL cannot be changed and is used the same way as the ACL of a regular resource.

VCR's ACL is the Same as the First Version

When a VCR is created by `makeversioned`, the ACL of the VCR is the same as the ACL of the first version of the resource. When a resource is checked in, a new version is created, and the VCR will have the same contents and properties including ACL property with this new version.

[Table 14-2](#) describes the subprograms in `DBMS_XDB_VERSION`.

Table 14–2 DBMS_XDB_VERSION Functions and Procedures

DBMS_XDB_VERSION Function/Procedure	Description
FUNCTION MakeVersioned MakeVersioned(pathname VARCHAR2) RETURN dbms_xdb.resid_type;	Turns a regular resource whose path name is given into a version controlled resource. If two or more path names are bound with the same resource, a copy of the resource will be created, and the given <code>path name</code> will be bound with the newly-created copy. This new resource is then put under version control. All other path names continue to refer to the original resource. <code>pathname</code> - the path name of the resource to be put under version control. <code>return</code> - This function returns the resource ID of the first version (root) of the VCR. This is not an auto-commit SQL operation. It is legal to call <code>MakeVersioned</code> for VCR, and neither exception nor warning is raised. It is illegal to <code>makeversioned</code> for folder, version resource, and ACL. An exception is raised if the resource doesn't exist.
PROCEDURE Checkout Checkout(pathname VARCHAR2);	Checks out a VCR before updating or deleting it. <code>pathname</code> - the path name of the VCR to be checked out. This is not an auto-commit SQL operation. Two users of the same workspace cannot checkout the same VCR at the same time. If this happens, one user must rollback. As a result, it is a good idea for you to commit the checkout operation before updating a resource. That way, you do not loose the update when rolling back the transaction. An exception is raised when: <ul style="list-style-type: none"> ■ the given resource is not a VCR, ■ the VCR is already checked out ■ the resource doesn't exist
FUNCTION Checkin Checkin(pathname VARCHAR2) RETURN dbms_xdb.resid_type;	Checks in a checked-out VCR. <code>pathname</code> - the path name of the checked-out resource. <code>return</code> - the resource id of the newly-created version. This is not an auto-commit SQL operation. <code>Checkin</code> does not have to take the same path name that was passed to <code>checkout</code> operation. However, the <code>checkin path name</code> and the <code>checkout path name</code> must be of the same resource for the operations to function correctly. If the resource has been renamed, the new name must be used to <code>checkin</code> because the old name is either invalid or bound with a different resource at the time being. Exception is raised if the path name does not exist. If the path name has been changed, the new path name must be used to <code>checkin</code> the resource.

Table 14–2 DBMS_XDB_VERSION Functions and Procedures (Cont.)

DBMS_XDB_VERSION Function/Procedure	Description
FUNCTION Uncheckout Uncheckout(pathname VARCHAR2) RETURN dbms_xdb.resid_type;	Checks in a checked-out resource. pathname - the path name of the checked-out resource. return - the resource id of the version before the resource is checked out. This is not an auto-commit SQL operation. UncheckOut does not have to take the same path name that was passed to checkout operation. However, the uncheckout path name and the checkout path name must be of the same resource for the operations to function correctly. If the resource has been renamed, the new name must be used to uncheckout because the old name is either invalid or bound with a different resource at the time being. An exception is raised if the path name does not exist. If the path name has been changed, the new path name must be used to checkin the resource.
FUNCTION GetRoot GetFirst(vh_id dbms_ xdb.resid_type) RETURN dbms_xdb.resid_type;	Given the version history, gets the root of all versions. vh_id - the resid of the version history. return - first version resource id. An exception is raised if the vh_id is illegal.

Table 14–2 DBMS_XDB_VERSION Functions and Procedures (Cont.)

DBMS_XDB_VERSION Function/Procedure	Description
FUNCTION GetPredecessors GetPredecessors(pathname VARCHAR2) RETURN resid_list_type; GetPredsByResId(resid dbms_xdb.resid_type) RETURN resid_list_type;	Given a version resource or a VCR, gets the predecessors of the resource by path name, the path name of the resource. return - list of predecessors. Getting predecessors by resid is more efficient than by path name. An exception is raised if the resid or path name is illegal. Given a version resource or a VCR, gets the predecessors of the resource by resid (resource id.)
FUNCTION GetSuccessors GetSuccessors(pathname VARCHAR2) RETURN resid_list_type; GetSuccsByResId(resid dbms_xdb.resid_type) RETURN resid_list_type;	Given a version resource or a VCR, gets the successors of the resource by pathname, the path name of the resource. return - list of predecessors. Getting successors by resid is more efficient than by path name. An exception is raised if the resid or path name is illegal. Given a version resource or a VCR, get the successors of the resource by resid (resource id.)
FUNCTION GetResourceByResId GetResourceByResId(resid dbms_xdb.resid_type) RETURN XMLType;	Given a resource object ID, gets the resource as an XMLType. resid - the resource object ID return - the resource as an XMLType

Frequently Asked Questions: Oracle XML DB Versioning

Can I Switch a VCR to a Non-VCR?

Answer: No.

How Do I Access the Old Copy of a VCR After Updating It?

Answer: The old copy is the version resource of the last checked-in:

- If you have the version ID or path name, you can load it using that ID.
- If you don't have its ID, you can call getPredecessors() to get the ID.

Can We Use Version Control for Data Other Than Oracle XML DB Data?

Answer: Only Oracle XML DB resources can be put under version control in this release.

RESOURCE_VIEW and PATH_VIEW

This chapter describes the SQL-based mechanisms, `RESOURCE_VIEW` and `PATH_VIEW`, used to access Oracle XML DB Repository data. It discusses the SQL operators `UNDER_PATH` and `EQUALS_PATH` used to query resources based on their path names and the SQL operators `PATH` and `DEPTH` that return the resource path names and depth.

It contains the following sections:

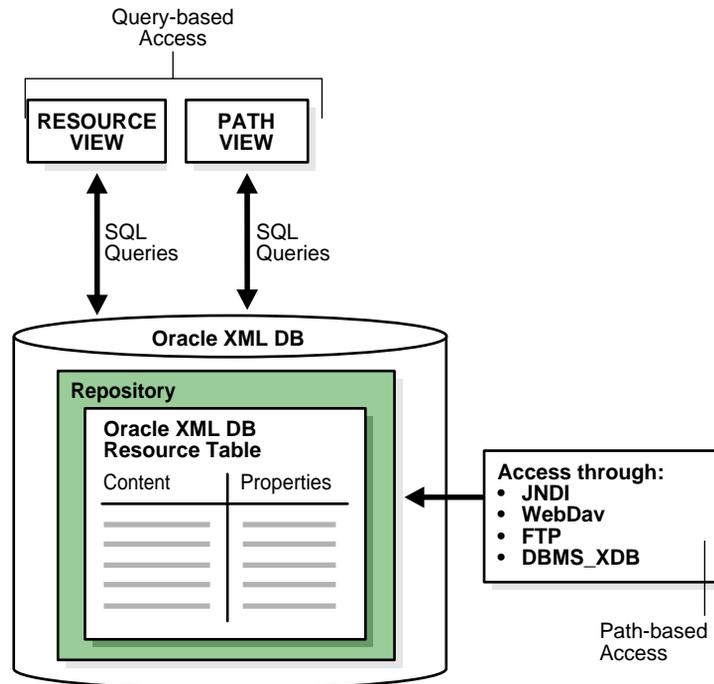
- [Oracle XML DB RESOURCE_VIEW and PATH_VIEW](#)
- [Resource_View, Path_View API](#)
- [UNDER_PATH](#)
- [EQUALS_PATH](#)
- [PATH](#)
- [DEPTH](#)
- [Using the Resource View and Path View API](#)
- [Working with Multiple Oracle XML DB Resources Simultaneously](#)

Oracle XML DB RESOURCE_VIEW and PATH_VIEW

Figure 15-1 shows how Oracle XML DB RESOURCE_VIEW and PATH_VIEW provide a mechanism for using SQL to access data stored in Oracle XML DB Repository. Data stored in Oracle XML DB Repository through protocols like FTP, WebDAV, or programming API such as JNDI, can be accessed in SQL using the resource and PATH_VIEWS, and vice versa.

RESOURCE_VIEW and PATH_VIEW together, along with PL/SQL package, DBMS_XDB, provide all query-based access to Oracle XML DB and DML functionality that is available through the programming API.

Figure 15-1 Accessing Repository Resources Using RESOURCE_VIEW and PATH_VIEW



RESOURCE_VIEW Definition and Structure

The RESOURCE_VIEW contains one row for each resource in the Repository. The following describes its structure:

Column	Datatype	Description
RES	XMLTYPE	A resource in Oracle XML Repository
ANY_PATH	VARCHAR2	A path that can be used to access the resource in the Repository

See Also: [Appendix G, "Example Setup scripts. Oracle XML DB-Supplied XML Schemas"](#)

PATH_VIEW Definition and Structure

The PATH_VIEW contains one row for each unique path to access a resource in the Repository. The following describes its structure:

Column	Datatype	Description
PATH	VARCHAR2	Path name of a resource
RES	XMLTYPE	The resource referred by PATH
LINK	XMLTYPE	Link property

See Also: [Appendix G, "Example Setup scripts. Oracle XML DB-Supplied XML Schemas"](#)

Figure 15-2 illustrates the structure of Resource and PATH_VIEWS.

Note: Each resource may have multiple paths called links.

The path in the RESOURCE_VIEW is an arbitrary one of the accessible paths that can be used to access that resource. Oracle XML DB provides operator UNDER_PATH that enables applications to search for resources contained (recursively) within a particular folder, get the resource depth, and so on. Each row in the views is of XMLTYPE. DML on the Oracle XML DB 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.

Figure 15–2 RESOURCE_VIEW and PATH_VIEW Structure

RESOURCE_VIEW Columns		PATH_VIEW Columns		
Resource as an XMLType	Path	Path	Resource as an XMLType	Link as XMLType
_____	_____	_____	_____	_____
_____	_____	_____	_____	_____
_____	_____	_____	_____	_____
_____	_____	_____	_____	_____

Understanding the Difference Between RESOURCE_VIEW and PATH_VIEW

The major difference between the RESOURCE_VIEW and PATH_VIEW is:

- PATH_VIEW displays all the path names to a particular resource whereas RESOURCE_VIEW displays one of the possible path names to the resource
- PATH_VIEW also displays the properties of the link

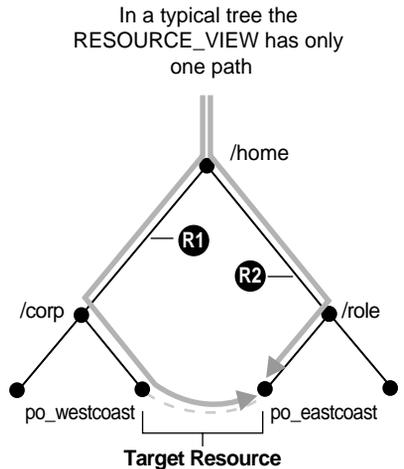
Figure 15–3 illustrates the difference between Resource and PATH_VIEW.

Since many internet applications only need one URL to access a resource, RESOURCE_VIEW is widely applicable.

PATH_VIEW contains the *link* properties as well as resource properties, whereas the RESOURCE_VIEW only contains resource properties.

The RESOURCE_VIEW benefit is generally optimization, if the database knows that only one path is needed, the index does not have to do as much work to determine all the possible paths.

Note: When using the RESOURCE_VIEW, if you are specifying a path with the UNDER_PATH or EQUALS_PATH operators, they will find the resource regardless of whether or not that path is the arbitrary one chosen to normally display with that resource using RESOURCE_VIEW.

Figure 15–3 RESOURCE_VIEW and PATH_VIEW Explained

With PATH_VIEW, to access the target resource node; You can create a link. This provides two access paths **R1** or **R2** to the target node, for faster access.

RESOURCE_VIEW Example:

```
select path(1) from resource_view where under_path(res, '/sys',1);
displays one path to the resource:
/home/corp/po_westcoast
```

PATH_VIEW Example:

```
select path from path_view;
displays all pathnames to the resource:
/home/corp/po_westcoast
/home/role/po_eastcoast
```

Operations You Can Perform Using UNDER_PATH and EQUALS_PATH

You can perform the following operations using UNDER_PATH and EQUALS_PATH:

- Given a path name:
 - Get a resource
 - List the directory given by the path name
 - Create a resource
 - Delete a resource
 - Update a resource
- Given a condition, containing UNDER_PATH operator or other SQL operators:
 - Update resources
 - Delete resources
 - Get resources

See the ["Using the Resource View and Path View API"](#) and `EQUALS_PATH`.

Resource_View, Path_View API

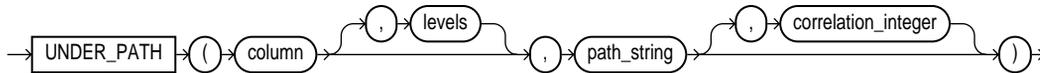
This section describes the `RESOURCE_VIEW` and `PATH_VIEW` operators:

UNDER_PATH

The `UNDER_PATH` operator uses the Oracle XML DB Repository hierarchical index to return the paths under a particular path. The hierarchical index is designed to speed access walking down a path name (the normal usage).

If the other parts of the query predicate are very selective, however, a functional implementation of `UNDER_PATH` can be chosen that walks back up the Repository. This can be more efficient, since a much smaller number of links may need to be traversed. [Figure 15-4](#) shows the `UNDER_PATH` syntax.

Figure 15-4 *UNDER_PATH Syntax*



[Table 15-1](#) describes the `UNDER_PATH` syntax.

Table 15-1 *RESOURCE_VIEW and PATH_VIEW API Syntax: UNDER_PATH*

Syntax	Description
INTEGER UNDER_PATH(resource_column, pathname);	Determines if a resource is under a specified path. Parameters: <ul style="list-style-type: none"> ▪ resource_column - The column name or column alias of the 'resource' column in the <code>path_view</code> or <code>resource_view</code>. ▪ pathname - The path name to resolve.

Table 15–1 RESOURCE_VIEW and PATH_VIEW API Syntax: UNDER_PATH (Cont.)

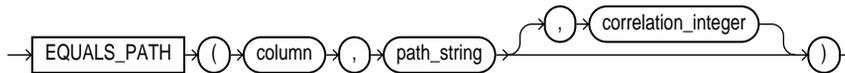
Syntax	Description
INTEGER UNDER_PATH(resource_column, depth, pathname);	<p>Determines if a resource is under a specified path, with a depth argument to restrict the number of levels to search.</p> <p>Parameters:</p> <ul style="list-style-type: none"> ■ resource_column - The column name or column alias of the 'resource' column in the path_view or resource_view. ■ depth - The maximum depth to search; a depth of less than 0 is treated as 0. ■ pathname - The path name to resolve.
INTEGER UNDER_PATH(resource_column, pathname, correlation)	<p>Determines if a resource is under a specified path, with a correlation argument for ancillary operators.</p> <p>Parameters:</p> <ul style="list-style-type: none"> ■ 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 the UNDER_PATH operator (a primary operator) with ancillary operators (PATH and DEPTH).
INTEGER UNDER_PATH(resource_column, depth, pathname, correlation)	<p>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 ancillary operators.</p> <p>Parameters:</p> <ul style="list-style-type: none"> ■ 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 depth of less than 0 is treated as 0. ■ pathname - The path name to resolve. ■ correlation - An integer that can be used to correlate the UNDER_PATH operator (a primary operator) with ancillary operators (PATH and DEPTH). <p>Note that only one of the accessible paths to the resource needs to be under the path argument for a resource to be returned.</p>

EQUALS_PATH

The `EQUALS_PATH` operator is used to find the resource with the specified path name. It is functionally equivalent to `UNDER_PATH` with a depth restriction of zero. The `EQUALS_PATH` syntax is describe here and in [Figure 15–5](#).

```
EQUALS_PATH INTEGER EQUALS_PATH( resource_column,pathname);
```

Figure 15–5 *EQUALS_PATH Syntax*



where:

- `resource_column` is the column name or column alias of the 'resource' column in the `path_view` or `resource_view`.
- `pathname` is the path name to resolve.

PATH

`PATH` is an ancillary operator that returns the relative path name of the resource under the specified `pathname` argument. Note that the `path` column in the `RESOURCE_VIEW` always contains the absolute path of the resource. The `PATH` syntax is:

```
PATH VARCHAR2 PATH( correlation);
```

where:

- `correlation` is an integer that can be used to correlate the `UNDER_PATH` operator (a primary operator) with ancillary operators (`PATH` and `DEPTH`).

DEPTH

`DEPTH` is an ancillary operator that returns the folder depth of the resource under the specified starting path.

```
DEPTH INTEGER DEPTH( correlation);
```

where:

correlation is an integer that can be used to correlate the UNDER_PATH operator (a primary operator) with ancillary operators (PATH and DEPTH).

Using the Resource View and Path View API

The following RESOURCE_VIEW and PATH_VIEW examples use operators UNDER_PATH, EQUALS_PATH, PATH, and DEPTH.

Accessing Paths and Repository Resources: Examples

The following examples illustrate how you can access paths, resources, and link properties in the Repository:

Example 15–1 Using UNDER_PATH: Given a Path Name, List the Directory Given by the Path Name from the RESOURCE_VIEW

```
select any_path from resource_view where any_path like '/sys%';
```

Example 15–2 Using UNDER_PATH: Given a Path Name, Get a Resource From the RESOURCE_VIEW

```
select any_path, extract(res, '/display_name') from resource_view
  where under_path(res, '/sys') = 1;
```

Example 15–3 Using RESOURCE_VIEW: Given a Path, Get all Relative Path Names for Resources up to Three Levels

```
select path(1) from resource_view
  where under_path (res, 3, '/sys',1)=1;
```

Example 15–4 Using UNDER_PATH: Given a Path Name, Get Path and Depth Under a Specified Path from the PATH_VIEW

```
select path(1) PATH,depth(1) depth
  from path_view
  where under_path(RES, 3,'/sys',1)=1
```

Example 15–5 Given a Path Name, Get Paths and Link Properties from PATH_VIEW

```
select path, extract(link, '/LINK/Name/text()').getstringval(),
  extract(link, '/LINK/ParentName/text()').getstringval(),
  extract(link, '/LINK/ChildName/text()').getstringval(),
  extract(res, '/Resource/DisplayName/text()').getstringval()
```

```
from path_view
where path LIKE '/sys%';
```

Example 15–6 Using UNDER_PATH: Given a Path Name, Find all the Paths up to a Certain Number of Levels, Including Links Under a Specified Path from the PATH_VIEW

```
select path(1) from path_view
where under_path(res, 3, '/sys', 1) > 0 ;
```

Example 15–7 Using EQUALS_PATH to Locate a Path

```
select any_path from resource_view
where equals_path(res, '/sys') > 0;
```

Inserting Data into a Repository Resource: Examples

The following example illustrates how you can insert data into a resource:

Example 15–8 Creating Resources: Inserting Data Into a Resource

```
insert into resource_view values(sys.xmltype.createxml( '
  <Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/xdb/XDBResource.xsd
http://xmlns.oracle.com/xdb/XDBResource.xsd">
  <Author>John Doe</Author>
  <DisplayName>example</DisplayName>
  <Comment>This resource was contrived for resource view demo</Comment>
  <Language>en</Language>
  <CharacterSet>ASCII</CharacterSet>
  <ContentType>text/plain</ContentType>
  </Resource>' ), '/home/SCOTT');
```

Deleting Repository Resources: Examples

The following examples illustrate how you can delete resources or paths:

Example 15–9 Deleting Resources

```
delete from resource_view where any_path = '/home/SCOTT/example
```

If only leaf resources are deleted, you can perform a delete using delete from resource_view where....

Deleting Non-Empty Containers Recursively

If only leaf resources are deleted, you can delete them using "delete from resource_view where...". For example, one way to delete leaf node '/public/test/doc.xml' is as follows:

```
delete from resource_view where under_path(res, '/public/test/doc.xml') = 1;
```

However, if you attempt to delete a non-empty container recursively, the following rules apply:

- Delete on a non-empty container is not allowed
- The order of the paths returned from the where clause predicates is not guaranteed

Therefore you should guarantee that a container is deleted only after its children have been deleted.

Example 15–10 Recursively Deleting Paths

For example, to recursively delete paths under '/public'), you may want to try the following:

```
delete from
(select 1 from resource_view
 where UNDER_PATH(res, '/public', 1) = 1
 order by depth(1) desc);
```

Updating Repository Resources: Examples

The following examples illustrate how to update resources and paths:

Example 15–11 Updating Resources

```
update resource_view set res = sys.xmltype.createxml( '
  <Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/xdb/XDBResource.xsd
http://xmlns.oracle.com/xdb/XDBResource.xsd">
  <Author>John Doe</Author>
  <DisplayName>example</DisplayName>
  <Comment>Has this got updated or not ? </Comment>
  <Language>en</Language>
  <CharacterSet>ASCII</CharacterSet>
  <ContentType>text/plain</ContentType>
```

```
</Resource>')  
  where any_path = '/home/SCOTT/example';
```

Example 15–12 Updating a Path in the PATH_VIEW

```
update path_view set path = '/home/XDB'  
  where path = '/home/SCOTT/example'
```

Note: If you need to get all the resources under a directory, you can use the LIKE operator, as shown in [Example 15–1](#) on page 15-9.

If you need to get the resources up to a certain number of levels, or get the relative path, then use the UNDER_PATH operator, as shown in [Example 15–2](#) on page 15-9.

The query plan for [Example 15–1](#) will be more optimal than that of [Example 15–2](#).

See Also: [Chapter 13, "Oracle XML DB Foldering"](#), [Table 13–3, "Accessing Oracle XML DB Repository: API Options"](#) on page 13-15 for additional examples that use the RESOURCE_VIEW and PATH_VIEW operators.

Working with Multiple Oracle XML DB Resources Simultaneously

Operations listed in [Table 13–3, Chapter 13, "Oracle XML DB Foldering"](#), typically apply to only one resource at a time. To perform the same operation on multiple Oracle XML DB resources, or to find one or more Oracle XML DB resources that meet a certain set of criteria, use RESOURCE_VIEW and PATH_VIEW in SQL.

For example, you can perform the following operations with these resource_view and PATH_VIEW SQL clauses:

- Updating based on attributes

```
UPDATE RESOURCE_VIEW SET ... WHERE extractValue(resource, '/display_name') =  
'My stuff'
```

- Finding recursively in a folder

```
SELECT FROM RESOURCE_VIEW WHERE UNDER_PATH(resource, '/sys') ...
```

- Copying a set of Oracle XML DB resources

```
INSERT INTO PATH_VIEW SELECT .... FROM PATH_VIEW WHERE ...
```

Oracle XML DB Resource API for PL/SQL (DBMS_XDB)

This chapter describes the Oracle XML DB Resource API for PL/SQL (DBMS_XDB) used for accessing and managing Oracle XML DB Repository resources and data using PL/SQL. It includes methods for managing the resource security and Oracle XML DB configuration.

It contains the following sections:

- [Introducing Oracle XML DB Resource API for PL/SQL](#)
- [Overview of DBMS_XDB](#)
- [DBMS_XDB: Oracle XML DB Resource Management](#)
- [DBMS_XDB: Oracle XML DB ACL-Based Security Management](#)
- [DBMS_XDB: Oracle XML DB Configuration Management](#)
- [DBMS_XDB: Rebuilding Oracle XML DB Hierarchical Indexes](#)

Introducing Oracle XML DB Resource API for PL/SQL

This chapter describes the Oracle XML DB Resource API for PL/SQL (PL/SQL package `DBMS_XDB`). This is also known as the PL/SQL foldering API.

Oracle XML DB Repository is modeled on XML and provides a database file system for any data. Oracle XML DB Repository maps path names (or URLs) onto database objects of `XMLType` and provides management facilities for these objects.

`DBMS_XDB` package provides functions and procedures for accessing and managing Oracle XML DB Repository using PL/SQL.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

Overview of `DBMS_XDB`

The `DBMS_XDB` provides the PL/SQL application developer with an API that manages:

- Oracle XML DB Resources
- Oracle XML DB ACL based Security
- Oracle XML DB Configuration
- Oracle XML DB Hierarchical Index Rebuild

`DBMS_XDB`: Oracle XML DB Resource Management

[Table 16-1](#) lists the `DBMS_XDB` Oracle XML DB resource management methods.

Table 16-1 *DBMS_XDB Resource Management Methods*

DBMS_XDB Method	Arguments, Return Values
Link	Argument: (srcpath VARCHAR2, linkfolder VARCHAR2, linkname VARCHAR2) Return value: N/A
LockResource	Argument: (path IN VARCHAR2, depthzero IN BOOLEAN, shared IN boolean) Return value: TRUE if successful.
GetLockToken	Argument: (path IN VARCHAR2, locktoken OUT VARCHAR2) Return value: N/A
UnlockResource	Argument: (path IN VARCHAR2, deltoken IN VARCHAR2) Return value: TRUE if successful.

Table 16–1 DBMS_XDB Resource Management Methods (Cont.)

DBMS_XDB Method	Arguments, Return Values
CreateResource	<p>FUNCTION CreateResource (path IN VARCHAR2, data IN VARCHAR2) RETURN BOOLEAN;</p> <p>Creates a new resource with the given string as its contents.</p> <p>FUNCTION CreateResource (path IN VARCHAR2, data IN SYS.XMLTYPE) RETURN BOOLEAN;</p> <p>Creates a new resource with the given XMLType data as its contents.</p> <p>FUNCTION CreateResource (path IN VARCHAR2, datarow IN REF SYS.XMLTYPE) RETURN BOOLEAN;</p> <p>Given a PREF to an existing XMLType row, creates a resource whose contents point to that row. That row should not already exist inside another resource.</p> <p>FUNCTION CreateResource (path IN VARCHAR2, data IN CLOB) RETURN BOOLEAN;</p> <p>Creates a resource with the given CLOB as its contents.</p> <p>FUNCTION CreateResource (path IN VARCHAR2, data IN BFILE) RETURN BOOLEAN;</p> <p>Creates a resource with the given BFILE as its contents.</p>
CreateFolder	<p>Argument: (path IN VARCHAR2)</p> <p>Return value: TRUE if successful.</p>
DeleteResource	<p>Argument: (path IN VARCHAR2)</p> <p>Return value: N/A</p>

Using DBMS_XDB to Manage Resources, Calling Sequence

Figure 16–1 describes the calling sequence when using DBMS_XDB to manage Repository resources:

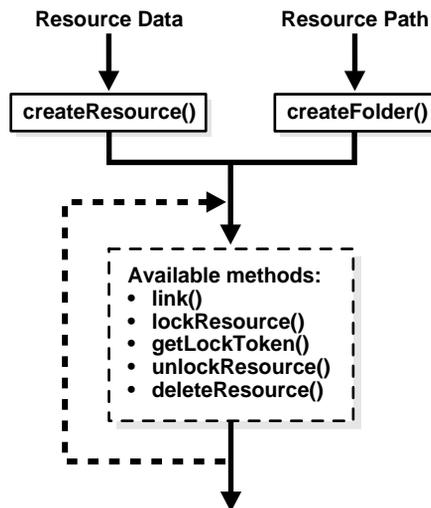
- When managing Repository resources the calling sequence diagram assumes that the resources and folders already exist. If not, you need to create the resources using `createResource()` or create folders using `createFolder()`
 - `createResource()` takes resource data and the resource path as parameters.
 - `createFolder()` takes the resource path as a parameter.
- If the resource or folder does not need further processing or managing, they are simply output.

3. If the resource or folder need further processing or managing you can apply any or all of the following methods as listed in [Table 16-1](#):

- `Link()`
- `LockResource()`
- `GetLockToken()`
- `UnlockResource()`
- `DeleteResource()`

See [Example 16-1](#) for an examples of using `DBMS_XDB` to manage Repository resources.

Figure 16-1 Using DBMS_XDB to Manage Resources: Calling Sequence



Example 16-1 Using DBMS_XDB to Manage Resources

```

DECLARE
    retb boolean;
BEGIN
    retb := dbms_xdb.createfolder('/public/mydocs');
    commit;
END;
/
  
```

```
declare
    bret boolean;
begin
    bret :=
dbms_xdb.createresource('/public/mydocs/emp_scott.xml', '<emp_name>scott</emp_
name>');
    commit;
end;
/

declare
    bret boolean;
begin
    bret :=
dbms_xdb.createresource('/public/mydocs/emp_david.xml', '<emp_name>david</emp_
name>');
    commit;
end;
/

call dbms_xdb.link('/public/mydocs/emp_scott.xml', '/public/mydocs',
'person_scott.xml');
call dbms_xdb.link('/public/mydocs/emp_david.xml', '/public/mydocs',
'person_david.xml');
commit;

call dbms_xdb.deleteresource('/public/mydocs/emp_scott.xml');
call dbms_xdb.deleteresource('/public/mydocs/person_scott.xml');
call dbms_xdb.deleteresource('/public/mydocs/emp_david.xml');
call dbms_xdb.deleteresource('/public/mydocs/person_david.xml');
call dbms_xdb.deleteresource('/public/mydocs');
commit;
```

DBMS_XDB: Oracle XML DB ACL-Based Security Management

Table 16–2 lists the DBMS_XDB Oracle XML DB ACL- based security management methods. Because the arguments and return values for the methods are self-explanatory, only a brief description of the methods is provided here.

Table 16–2 DBMS_XDB: Security Management Methods

DBMS_XDB Method	Arguments, Return Values
getAclDocument	Argument: (abspath VARCHAR2) Return value: XMLType for the ACL document
ACLCheckPrivileges	Argument: (acl_path IN VARCHAR2, owner IN VARCHAR2, privs IN XMLType) Return value: Positive integer if privileges are granted.
checkPrivileges	Argument: (res_path IN VARCHAR2, privs IN XMLType) Return value: Positive integer if privileges are granted.
getprivileges	Argument: (res_path IN VARCHAR2) Return value: XMLType instance of the <privilege> element.
changePrivileges	Argument: (res_path IN VARCHAR2, ace IN XMLType) Return value: Positive integer if ACL was successfully modified.
setAcl	Argument: (res_path IN VARCHAR2, acl_path IN VARCHAR2). This sets the ACL of the resource at res_path to the ACL located at acl_path. Return value: N/A

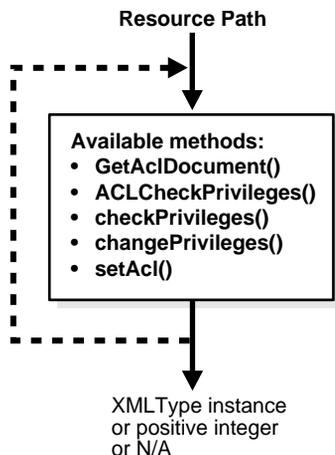
Using DBMS_XDB to Manage Security, Calling Sequence

Figure 16–2 describes the calling sequence when using DBMS_XDB to manage security.

1. Each DBMS_XDB security management method take in a path (resource_path, abspath, or acl_path).
2. You can then use any or all of the DBMS_XDB methods listed in Table 16–2 to perform security management tasks:
 - getAclDocument()
 - ACLCheckPrivileges()
 - checkPrivileges()
 - getPrivileges()
 - changePrivileges()
 - setACL()

See [Example 16–2](#) for an examples of using DBMS_XDB to manage Repository resource security.

Figure 16–2 Using DBMS_XDB to Manage Security: Calling Sequence



Example 16–2 Using DBMS_XDB to Manage ACL-Based Security

```

DECLARE
    retb boolean;
BEGIN
    retb := dbms_xdb.createfolder('/public/mydocs');
    commit;
END;
/

declare
    bret boolean;
begin
    bret :=
dbms_xdb.createresource('/public/mydocs/emp_scott.xml', '<emp_name>scott</emp_
name>');
    commit;
end;
/

call dbms_xdb.setacl('/public/mydocs/emp_scott.xml',
'/sys/acls/all_owner_acl.xml');

```

```
commit;

select dbms_xdb.getacldocument('/public/mydocs/emp_scott.xml') from
dual;

declare
  r          pls_integer;
  ace       xmltype;
  ace_data  varchar2(2000);
begin
  ace_data :=
'<ace
  xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                      http://xmlns.oracle.com/xdb/acl.xsd
                      DAV:http://xmlns.oracle.com/xdb/dav.xsd">
  <principal>SCOTT</principal>
  <grant>true</grant>
  <privilege>
    <all/>
  </privilege>
</ace>';
  ace := xmltype.createxml(ace_data);
  r := dbms_xdb.changeprivileges('/public/mydocs/emp_scott.xml', ace);
  dbms_output.put_line('retval = ' || r);
  commit;
end;
/

select dbms_xdb.getacldocument('/public/mydocs/emp_scott.xml') from
dual;

select dbms_xdb.getprivileges('/public/mydocs/emp_scott.xml') from dual;

call dbms_xdb.deleteresource('/public/mydocs/emp_scott.xml');
call dbms_xdb.deleteresource('/public/mydocs');
commit;
```

DBMS_XDB: Oracle XML DB Configuration Management

[Table 16-3](#) lists the DBMS_XDB Oracle XML DB Configuration Management Methods. Because the arguments and return values for the methods are self-explanatory, only a brief description of the methods is provided here.

Table 16–3 DBMS_XDB: Configuration Management Methods

DBMS_XDB Method	Arguments, Return Value
CFG_get	Argument: None Return value: XMLType for session configuration information
CFG_refresh	Argument: None Return value: N/A
CFG_update	Argument: (xdbconfig IN XMLType) Return value: N/A

Using DBMS_XDB for Configuration Management, Calling Sequence

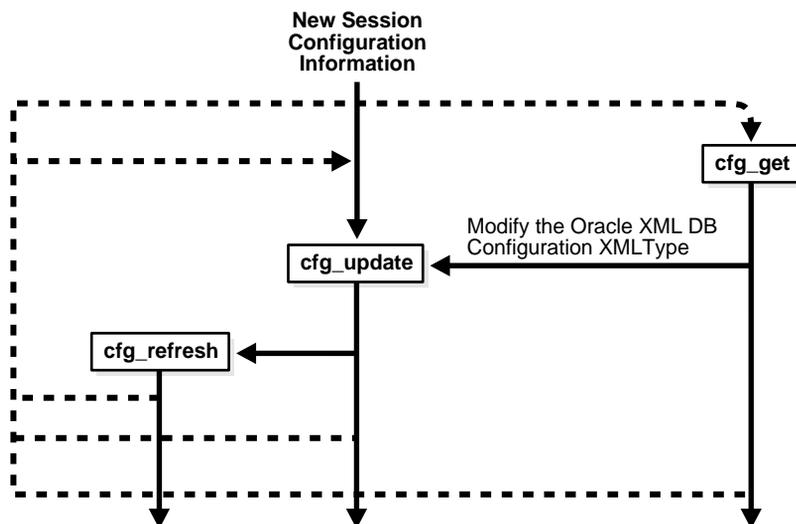
[Figure 16–3](#) shows the calling sequence when using DBMS_XDB for configuration management.

The diagram shows the following sequence:

1. To manage the Oracle XML DB configuration you must first retrieve the configuration instance using `cfg_get`.
2. You can then optionally also modify the Oracle XML DB configuration xmltype instance in order to update it, or simply output the Oracle XML DB configuration.
3. To update the Oracle XML DB configuration resource use `cfg_update`. You need to either input a new Oracle XML DB configuration xmltype instance or use a modified version of the current configuration.
4. To refresh the Oracle XML DB configuration resource use `cfg_refresh`. You do not need to input a configuration xmltype instance.

See [Example 16–3](#) for an example of using DBMS_XDB for configuration management of Repository resources.

Figure 16–3 Using DBMS_XDB for Configuration Management: Calling Sequence



Example 16–3 Using DBMS_XDB for Configuration Management of Oracle XML DB

```
connect system/manager
```

```
select dbms_xdb.cfg_get() from dual;
```

```
declare
```

```
  config  xmltype;
```

```
begin
```

```
  config := dbms_xdb.cfg_get();
```

```
  -- Modify the xdb configuration using updatexml, etc ...
```

```
  dbms_xdb.cfg_update(config);
```

```
end;
```

```
/
```

```
-- To pick up the latest XDB Configuration
```

```
-- In this example it is not needed as cfg_update(),
```

```
-- automatically does a cfg_refresh().
```

```
call dbms_xdb.cfg_refresh();
```

DBMS_XDB: Rebuilding Oracle XML DB Hierarchical Indexes

[Table 16-4](#) lists the DBMS_XDB Oracle XML DB hierarchical index rebuild methods. Because the arguments and return values for the methods are self-explanatory, only a brief description of the methods is provided here.

Table 16-4 *DBMS_XDB: Hierarchical Index Rebuild Method*

DBMS_XDB Method	Arguments, Return Values
RebuildHierarchicalIndex	Argument: None Return value: N/A

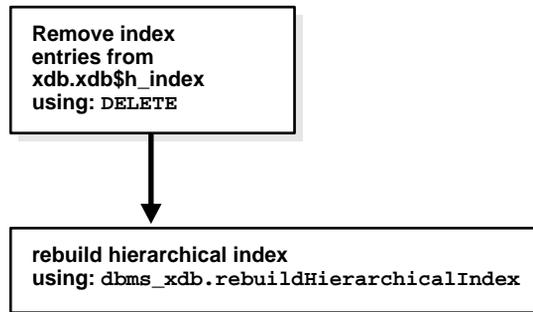
Using DBMS_XDB to Rebuild Hierarchical Indexes, Calling Sequence

[Figure 16-4](#) shows the calling sequence when using DBMS_XDB for rebuilding hierarchical indexes.

To rebuild the hierarchical indexes, first delete the entries from `xdb.xdb$h_index` and rebuild the hierarchical index by executing `DBMS_XDB.RebuildHierarchicalIndex`.

[Example 16-4](#) shows how to use DBMS_XDB to rebuild the Repository hierarchical indexes.

Figure 16-4 *Using DBMS_XDB to Rebuild Hierarchical Indexes: Calling Sequence*



Example 16-4 *Using DBMS_XDB for to Rebuild a Hierarchical index*

```

connect system/manager
delete from xdb.xdb$h_index;
  
```

```
commit;  
execute dbms_xdb.RebuildHierarchicalIndex;
```

Oracle XML DB Resource API for Java/JNDI

This chapter describes the Oracle XML DB Resource API for Java. It includes the JNDI interface for accessing and manipulating the Repository data.

This chapter contains these sections:

- [Introducing Oracle XML DB Resource API for Java/JNDI](#)
- [Using Oracle XML DB Resource API for Java/JNDI](#)
- [Calling Sequence for Oracle XML DB Resource API for Java/JNDI](#)
- [Parameters for Oracle XML DB Resource API for Java/JNDI](#)
- [Oracle XML DB Resource API for Java/JNDI Examples](#)

Introducing Oracle XML DB Resource API for Java/JNDI

What Is JNDI?

Oracle XML DB supports Java Naming and Directory Interface (JNDI) as the standard navigational access Application Program Interface (API). JNDI is a Sun programming interface. It connects Java programs to naming and directory services such as Domain Name Server (DNS), Lightweight Directory Access Protocol (LDAP), and Network Directory Service (NDS). JNDI is most commonly used as a client interface to LDAP servers and can also be used to access local operating system files.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

JNDI Support in Oracle XML DB

Oracle XML DB uses JNDI to locate resources and manage collections. JNDI returns `XMLType` objects that support both a Java Bean and Document Object Model (DOM) interface. Most typical file system operations (rename, link, delete, and so on) are provided through the JNDI interface. JNDI also provides a simple interface for accessing attribute data (as do DOM and Java Beans), which is not as powerful as the other APIs.

See Also: ["Java Bean API for XMLType"](#) on page 9-20

Once an `XMLType` object is retrieved (using either JNDI or JDBC/SQL), it can be cast to the matching Java Bean class or the `DOMNode` class. Oracle XML DB tracks any changes to the `XMLType` object that you make in memory through Java Bean methods or DOM APIs, and you can save those incremental changes back to the database with the `save()` method on `XMLType`. Alternatively, you can bind `XMLType` objects into SQL statements to modify an entire `XMLType` object at once or to alter individual pieces with `UPDATE_VAL()` in SQL.

- Oracle XML DB Resource API for Java/JNDI allows Java applications in the server or client to access and manipulate objects stored in Oracle XML DB Repository.
- JNDI is flexible enough to allow an object to be returned for a given path without imposing the limitations of current file systems on the interface. The object returned is commonly a Java bean with methods for each element and attribute of the XML element loaded.

- The bean also implements DOM interfaces so that dynamic data can be accessed easily as well. Server access to data is centralized in LDAP servers as well as the database with the same API. In JNDI, *folders* are called *Contexts*. The two terms are used interchangeably in this chapter.
- Oracle XML DB Resource API for Java/JNDI supports JNDI Service Provider Interface (SPI).

Note: In this release, Oracle XML DB's implementation of JNDI SPI only supports the `javax.naming` interface. Directory, attribute, and event operations are not yet supported.

oracle.xdb.spi

The application writes to the JNDI API. The directory drivers are written to the JNDI SPI (Service Provider Interface). Classes in package `oracle.xdb.spi` implement service provider drivers.

WebDAV Support

Package `oracle.xdb.spi` contains Oracle-specific extensions to the JNDI public standard. `oracle.xdb.spi` also provides support for WebDAV (Web Distributed Authoring and Versioning).

See Also: [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#)

Oracle XML DB Resource API for Java/JNDI Features

Oracle XML DB Resource API for Java/JNDI provides application programs with the following features:

- Resource manipulation. You can create, update, remove, rename, and move an Oracle XML DB resource, and look up an Oracle XML DB resource by name.
- Folder manipulation. You can create, update, remove, rename, move, and list the contents of a folder.
- Retrieve contents of the Oracle XML DB resource. Using Oracle extensions, you can update the properties of the Oracle XML DB resource.
- Save contents to the database.

Oracle XML DB Resource API for Java/JNDI is implemented as a wrapper over C functions using Java Native Interface (JNI). Because the Oracle Java layer is very thin, all the performance benefits of the underlying C implementation are achieved.

Use With Java API for XMLType

The JNDI API can be embedded in any Java application that can also use the Java API for XMLType to perform database operations such as:

- Opening connections to the database
- Committing or rolling back operations. Oracle XML DB Resource API for Java/JNDI does not perform commits by itself

See Also: [Chapter 9, "Java and Java Bean APIs for XMLType"](#),
["Java Bean API for XMLType"](#) on page 9-20

Using Oracle XML DB Resource API for Java/JNDI

Oracle XML DB Resource API for Java/JNDI operates as follows:

- JNDI is used, typically through the context `lookup()` method, to locate an existing Oracle XML DB resource.
- Alternatively, JDBC is used to access the Oracle XML DB resource view to retrieve and modify resources.
- JNDI can return:
 - An instance of the Oracle XML DB Resource interface
 - An instance of the Java bean (if registered) or `XMLDocument` for the contents of the XML data

Additionally, you can perform these operations:

- Use the Java API for XMLType to access and modify parts of the XMLType object.
- Use the `save()` method to write changes to the database.
- Use the JNDI `bind()` family to bind the Oracle XML DB resource to a new path name.

The Oracle XML DB Resource API for Java includes a set of subinterfaces that indicate resource type information as follows:

- `Context` (used for folders). This is for JNDI.

- Versioning. This is for WebDAV.

Oracle XML DB Resource API for Java includes interfaces for objects such as workspaces, branches, and baselines (as defined by the WebDAV versioning specification). `Context` and `Resource` are Java bean-based accessors. Since they are `XMLType` subclasses, they can implement the Java API for `XMLType`.

JNDI SPI resides in Oracle9i database in the internal SYS schema in the package. The package `xdb.jar.xdb.jar` can also be used by applications for further development. Applications developed to use Oracle XML DB Resource API for Java/JNDI must also be loaded in Oracle9i database in order to use the JNDI API.

Calling Sequence for Oracle XML DB Resource API for Java/JNDI

Get started with JNDI by creating an initial context or folder. This corresponds to the root directory of Oracle XML DB. To create an initial context, pass a hashtable with parameters (name/value pairs) to indicate how JNDI is to be configured.

Parameters for Oracle XML DB Resource API for Java/JNDI

[Table 17-1](#) lists the parameters supported by Oracle XML DB Resource API for Java/JNDI.

Table 17-1 Oracle XML DB Resource API for Java/JNDI: Parameters

Parameter Name	Description
PROVIDER_URL	The start path from which objects are to be returned.
INITIAL_CONTEXT_FACTORY	The context factory to be used to generating contexts – always “oracle.xdb.spi.XDBContextFactory”.

Table 17–1 Oracle XML DB Resource API for Java/JNDI: Parameters (Cont.)

Parameter Name	Description
XDB_RESOURCE_TYPE	<p>Determines what data is returned to the application by default when a path name is resolved:</p> <ul style="list-style-type: none"> ■ “Resource” : A resource class will be returned. ■ “XMLType” : The contents of the resource will be returned. <p>In this release, Oracle XML DB has implemented only the <code>javax.naming</code> package. Oracle XML DB has an extension to this package, an extension to the <code>lookup()</code> method. The extension (an overload) also takes a Boolean (indicating that a row lock should be grabbed) to indicate a “lookup FOR UPDATE” and a String with an XPath to define a fragment of the document to load immediately (rather than relying on the lazy manifest facility).</p>

Oracle XML DB Resource API for Java/JNDI has two main components:

- Context module. This implements the JNDI Context interface, which accesses the Oracle XML DB Repository using the `lookup`, `bind`, `rename`, `unbind`, `move`, and `list` functions.
- A set of classes/interfaces. These are returned from operations performed on the Context module. These classes/interfaces are:
 - Enumerators. These enumerate name/classname pairs or name/object pairs.
 - Other classes returned by the Context module. These can be:
 - * A resource. A leaf in Oracle XML DB Repository that implements methods to modify the properties of the Oracle XML DB resources and folders.
 - * `XMLType/Bean/org.w3c.Document` interface. These operate on XML documents stored in Oracle XML DB Repository.

Using JNDI Contexts - `XDBContextFactory()`

Oracle XML DB implementation of JNDI provides a context factory class `XDBContextFactory()`. This implements the `initialContextFactory` interface used by JNDI to create initial contexts.

Initial Context

The initial context describes the starting point in the Repository hierarchy from which all name resolutions happens, with the exception of absolute names which begin with “/”. Oracle XML DB does not differentiate between `initialContexts` and `Contexts` since it stores the initial path of the application in an environment variable in the class. Thus, `initialContextFactory` generates an `XDBContext` class that implements the `Context` interface. `initialContextFactory` also takes in as parameters the Language and the Country names for Globalization support.

`XDBContext` class return results in the form of a pointer to the data. The JNI implementation passes this pointer and other relevant data back to the Java Context API, which generates the appropriate objects from the pointer and other data. This information is then returned to the application program, which can further make calls on these objects. The implementation is very fast.

How JNDI's Context Module Returns Objects

JNDI's Context module returns various objects based on the following:

- Operation
- Path
- Environment variables

The following objects are created by the Context module's operation:

- `oracle.xml.xdb.XDBResource`: Returned by lookup when the environment variable is set to `resource`. The environment variables are described in *Oracle9i XML API Reference - XDK and Oracle XML DB*
- This class implements methods to manipulate the metadata of the objects in the hierarchy.
- `oracle.xml.xdb.XMLType`, Java Bean: Implements the methods defined by the SQL `XMLType` API. This can also be obtained from the Resource if it does not point to a folder (that is, a leaf). This can also handle non-XML data using the Binary schema.

See Also: [Chapter 9, "Java and Java Bean APIs for XMLType"](#) which explains the `XMLType` interface and Oracle XML DB's DOM support.

A Java bean is a class specific to XML documents for which an XML schema has been registered. The lookup/list methods that return objects will determine if

the document returned is an XML object for which an XML schema has been registered.

- `javax.naming.NamingEnumeration`: Provides a means for enumerating the contents of the Oracle XML DB Folder as a name/class name pair. The class name is one of the following:
 - `oracle.xdb.XDBCContext` (if object is a folder)
 - A bean
 - `oracle.xdb.XMLType`

The class name is dependent on the document stored and the environment returned. It also provides a high-level overview of the processing flows and associated inputs and outputs among the major functional components. It includes the processing flows to the external sources for the external inputs and outputs that cross into and out of the system domain.

- `javax.naming.BindingEnumeration`: Provides a way to enumerate the contents of the Oracle XML DB folder as a name/object pair. The object is one of the following:
 - `oracle.xdb.XDBCContext` (if object is a folder)
 - A bean
 - `oracle.xdb.XMLType`

The object type is dependent on the document stored and the environment returned.

JNDI Context Module's Inputs

`InitialContextFactory()` inputs a `java.util.Hashtable` class, which contains the input parameters to control the creation of the context.

[Table 17-2](#) describes these input parameters, their valid values, and their uses.

Table 17–2 Oracle XML DB Resource API for Java: JNDI Context Input Parameters

JNDI Input Parameters Name	Valid Values	Description
Context.PROVIDER_URL	path string, “/” for example	Contains the starting path from which name resolution is to begin.
Context.INITIAL_CONTEXT_FACTORY	<code>oracle.xml.db.spi.XDBContextFactory</code>	The value of the context factory to be used for generating the initial context object.
XDBContext.RETURN_OBJECT_TYPE	<code>Resource</code> , <code>contents</code>	Determines if the caller is interested in the resource (the metadata object) or the actual contents. For example, a database application which is interested only in the data might not be interested in the metadata. The default is <code>Contents</code> .
XDBContext.COUNTRY	The country name string	Determines the locale to be used for error messages. The default is database locale.
XDBContext.LANGUAGE	The language name strings	Controls the error message language, and so on. The default is database language.

JNDI Context Processing

The first class in the Oracle XML DB SPI that JNDI invokes for the application is `XDBContextFactory()`.

`NamingManager` calls `getInitialContext()` method of `XDBContextFactory()`. `XDBContextFactory()` is part of `oracle.xml.db.spi` package.

`XDBContextFactory` is defined as follows:

```
public class XDBContextFactory implements InitialContextFactory
{
    public Context getInitialContext(Hashtable env) throws NamingException;
};
```

`RETURN_OBJECT_TYPE` lets you specify the intent of the application, that is, the kind of data you are interested in.

The purpose of `InitialContextFactory()` is to generate the initial object from which the child objects can be obtained.

`XDBContext()` class implements `javax.naming.Context` interface. It is part of `oracle.xml.db.spi` package.

`Context` interface implements a method, `close()` which implements cleanup operations for the object.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

JNDI Context Module Outputs

The following is the output generated by the methods of the JNDI Context module:

- `XDBResource()`: Class generated as a result of a lookup operation if `RETURN_OBJECT_TYPE` is set to `Resource`.
- `XMLType()`: Class generated as a result of the lookup operation if `RETURN_OBJECT_TYPE` is set to `Contents`.
- `NamingEnumeration`: Returned as a result of the `list` method.
- `BindingEnumeration`: Returned as a result of the `listBindings` operation.
- `Bean`: Generated as a result of the lookup operation if the object being looked up is an XML schema-based XML document and a bean for the XML schema has been generated.

JNDI Context Module Error Messages

The following are exceptions thrown by JNDI Context APIs:

- `javax.naming.NamingException`: A naming exception has occurred.
- `javax.naming.NameAlreadyBoundException`: The name is already bound.
- `javax.naming.NameNotFoundException`: The name cannot be located in the hierarchy
- `javax.naming.NotContextException`: The object is not a context.
- `javax.naming.ContextNotEmptyException`: The context being deleted is not empty.

JNDI Objects

JNDI objects are constructed on calling various methods on the `Context` interface. The `Context` `lookup` method generates either of the following:

- `XDBResource` class

- XDBDocument class/bean class

The `listBinding()` method also generates a `BindingEnumeration()` class, which has methods to generate these classes.

- JNDI objects input

The input to this module is the `cstate` which has been obtained by the Context from the various methods for the given path.

The important methods in `XDBClassEnumeration()` class are:

- `hasMore()`: The derived classes of the `XDBClassEnumeration` class use this method from the base class directly without any modification.
- `next()`. This gives you the next object in the enumeration. The object can be either a `ClassName` pair or a `ListBinding`.

- JNDI objects output

The outputs of the JNDI objects component are:

- `XDBResource`: Result of `XDBBindingEnumeration`.
- `XDBDocument`: Result of `XDBBindingEnumeration`.
- `NameClassPair`: Result of `NamingEnumeration`.
- `Binding`: Result of `BindingEnumeration`.

- JNDI Objects Error Messages

Exceptions will be thrown by these classes:

- `NamingException`: General naming exception.
- `NoSuchNameException`: Attempt to get next when none is available.

Oracle XML DB Resource API for Java/JNDI Examples

Example 17-1 Resource API JNDI: Using JNDI to Determine Purchase Order Properties

Here is a way to find out properties of an Oracle XML DB purchase order object using JNDI alone. JNDI allows properties to be looked up using a string property name. This API is useful when the properties to be retrieved are not known at compile time, as well as to provide the same interface as JNDI service providers other than Oracle XML DB, like an LDAP directory.

This example retrieves a string property `title` and a Boolean property `isCurrent` from an XMLType resource:

```
import javax.naming.*;
import java.util.*;

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "oracle.xdb.spi.XDBContextFactory");

DirContext ctx = new InitialDirContext(env);
Attributes po = ctx.getAttributes("/orders/123.xml");
String title = (String)po.get("title");
boolean isCurrent = ((Boolean)po.get("isCurrent")).booleanValue();
```

Example 17–2 Resource API JNDI: Using DOM to Determine Purchase Order Properties

Here is a JNDI example using DOM for attribute access. The environment setup is the same as in the previous example and omitted here:

```
Context ctx = new InitialContext(env);
Document po = (Document)ctx.lookup("/orders/123.xml");
Node n;

n = po.getElementsByTagName("Title").item(1);
String title = n.getNodeValue();
n = po.getElementsByTagName("IsCurrent").item(1);
boolean isCurrent = new Boolean(n.getNodeValue()).booleanValue();
```

Note: The DOM example requires a type conversion from a string to a Boolean to get the desired data. This adversely affects performance.

Example 17–3 Resource API Java Beans: Using JNDI to Determine Purchase Order Properties

Here is a Java Bean version of the DOM example. It uses JNDI to locate the object with the same setup:

```
Context ctx = new InitialContext(env);
PurchaseOrder po = ctx.lookup("/orders/123.xml");
Boolean isCurrent = po.isCurrent();
String title = po.getTitle();
```

Note how simple the Java Bean API is. In this case, the return of `lookup()` is simply cast to the corresponding Bean class. The application must already have knowledge of the datatype at a particular filename or else it must use methods defined on `XMLType` to determine what class to cast to.

Example 17–4 Resource JDBC: Using SQL To Determine Purchase Order Properties

Here is an example using SQL instead of JNDI to find the object. This example uses `XMLType` to access the resource. JDBC is not as fast as JNDI, but it provides the power of SQL `SELECT` potentially using criteria other than path name to find the `XMLType` object.

```
PreparedStatement pst = con.prepareStatement(
"SELECT r.RESOLVE_PATH('/companies/oracle') FROM XDB$RESOURCE r");

pst.executeQuery();
XMLType po = (XMLType)pst.getObject(1);
Document podoc = (Document) po.getDOM();
```

Oracle XML DB Resource Security

This chapter describes Access Control Lists (ACL) based security mechanism for Oracle XML DB resources. It describes how to create ACLs, set and change ACLs on resources, and how ACL security interacts with other database security mechanisms.

This chapter contains the following sections:

- [Introducing Oracle XML DB Resource Security and ACLs](#)
- [Access Control List Terminology](#)
- [Oracle XML DB ACL Features](#)
- [Access Control: User and Group Access](#)
- [Oracle XML DB Supported Privileges](#)
- [ACL Evaluation Rules](#)
- [Using Oracle XML DB ACLs](#)
- [ACL and Resource Management](#)
- [Using DBMS_XDB to Check Privileges](#)

Introducing Oracle XML DB Resource Security and ACLs

Oracle XML DB maintains object-level security for any resource in Oracle XML DB Repository hierarchy.

Note: XML objects not stored in Oracle XML DB Repository do not have object-level access control.

Oracle XML DB uses an access control list (ACL) mechanism to restrict access to any Oracle XML DB resource or database object mapped to Oracle XML DB Repository.

The Oracle XML DB ACL security mechanism supports the WebDAV ACL specification. ACLs are a standard security mechanism used in Java, Windows NT, and other systems.

Oracle XML DB ACL security mechanism is designed to handle large volumes of XML data stored in Oracle9i database. Privileges can be granted or denied to the principal `dav:owner`, that represents the owner of the document, regardless of who the owner is.

See Also:

- [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)
- *Oracle9i XML API Reference - XDK and Oracle XML DB*

How the ACL-Based Security Mechanism Works

Before a user performs an operation or method on a resource, a check of privileges for the user on the resource takes place. The set of privileges checked depends on the operation or method performed. For example, to increase employee Scott's salary by 10 percent, `READ` and `WRITE` privileges are needed for the `scott/salary.xml` resource.

Access Control List Terminology

A few access control list (ACL) terms are described here:

- *Principal*. An entity that may be granted access control privileges to an Oracle XML DB resource. Oracle XML DB supports as principals:
 - Database users.

- Database roles. A database role can be understood as a group, for example, the DBA role represents the DBA group of all the users granted the DBA role.

Note: Users and roles imported from an LDAP server are also supported as a part of the database's general authentication model.

There is a special principal named `dav:owner` that corresponds to a separate property on the object being secured. Use of the `dav:owner` principal allows greater ACL sharing between users, since the owner of the document often has special rights. See Also "[Access Control: User and Group Access](#)" on page 18-6.

- *Privilege:* This is a particular right that can be granted to a principal. Oracle XML DB has a set of system-defined rights (such as READ, INSERT, or UPDATE) that can be referenced in any ACL. Privileges can be one of the following:
 - Aggregate (containing other privileges)
 - Atomic (which cannot be subdivided)

Aggregate privileges are a naming convenience to simplify usability when the number of privileges becomes large, as well as to promote interoperability between ACL clients. A set of privileges controls the ability to perform a given operation or method on an Oracle XML DB resource. For example, if the principal `Scott` wants to perform the `read` operation on a given resource, the `read` privileges must be granted to `Scott` prior to the read operation. Therefore, privileges control how users can operate on given resources.

- *ACE (access control entry):* Part of an ACL that grants or denies access to a particular principal. An ACL consists of a list of ACEs where ordering is irrelevant. There can be only one `grant ACE` and one `deny ACE` for a particular principal in a single ACL.

Note: Many `grant ACEs` (or `deny ACEs`) may apply to a particular user since a user may be granted many roles.

An Oracle XML DB ACE element has the following attributes:

- **Operation:** Either `grant` or `deny`
- **Principal:** Either a User or a group/collection

- Privileges Set: Particular set of privileges that are to be either granted or denied for a particular principal
- *Access control list (ACL):* A list of access control entry elements, with the element name *ace*, that defines access control to a resource. An ACE either grants or denies privileges for a principal.

Note: ACLs are stored as Oracle XML DB resources, so they also need to be protected by Oracle XML DB ACLs. There is only one ACL, the **bootstrap ACL**, that is self-protected; that is, it is protected by its own contents.

- *Named ACLs:* An ACL that is a resource itself, that is, it has its own path name. Named ACLs can be shared by multiple resources, improving manageability, ease of use, and performance. Named ACLs have a unique name and also have an optional type restrictor, such as:
`http://xmlns.oracle.com/xdb/XDBDemo.xsd#PurchaseOrder`
 that specifies that the ACL can only be applied to instances of that XML element and elements in a substitution group with that element.
- *Default ACL:* When a resource is inserted into the Oracle XML DB Repository, there are two ways to specify an ACL for this resource:
 - Using the default ACL (the ACL of the parent folder)
 - Specifying a particular ACL
- *Bootstrap ACL:* Every ACL is protected by the contents of another ACL except the bootstrap ACL. The bootstrap ACL, stored in:
`/sys/acls/bootstrap_acl.xml`, is the only ACL protected by its own contents. All of the default ACLs are protected by the bootstrap ACL, which grants the `xdb:readContents` privilege to all users. The bootstrap ACL grants FULL ACCESS to Oracle XML DB ADMIN and DBA groups. The XDBADMIN role is particularly useful for users that must register global XML schemas.
- *Other ACLs supplied with Oracle XML DB:*
 - `all_all_acl.xml` Grants all privileges to all users.
 - `all_owner_acl.xml` Grants all privileges to owner user.
 - `ro_all_acl.xml` Grants read privileges to all users.

- *ACL file-naming conventions:* Supplied ACLs use the following file-naming conventions: `privilege_users_acl.xml`
 where `privilege` represents the privilege granted and `user` represents the users that are granted access to the resource.

Oracle XML DB ACL Features

Oracle XML DB supports the following ACL features:

ACL Interaction with Oracle XML DB Table/View Security

Users must have the appropriate privilege on the underlying table/view where the XML object is stored, as well as permissions through the ACL for that individual instance.

Note: Some, but not all, objects in a particular table may be mapped to Oracle XML DB resources. In that case, only those objects mapped into the Oracle XML DB Repository hierarchy have ACL checking done, although they will all have table-level security.

LDAP Integration and User IDs

LDAP is integrated with Oracle XML DB to allow external users access to Oracle XML DB. External users can perform the same operations that a local database user can.

Oracle XML DB Resource API for ACLs (PL/SQL)

The PL/SQL API for ACL security allows the PL/SQL developer access to the security mechanisms, to check privileges given a particular ACL, and to list the set of privileges the current user has for a particular ACL and object.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

How Concurrency Issues Are Resolved with Oracle XML DB ACLs

Oracle XML DB ACLs are cached for very fast evaluation. When a transaction modifying an ACL is committed, the modified ACL is picked up after the time-out specified in the Oracle XML DB configuration file is up. The XPath for this configuration parameter is `/xdbconfig/sysconfig/acl-max-age`.

Access Control: User and Group Access

The `principal` can be either an individual user or a group. A group is also referred to as a **collection**. A user is granted access as a group principal if the user has been granted a database role.

Access privileges for each principal are stored in access control entries (ACEs) in the ACL.

Example 18–1 ACE Entries in an ACL for Controlling User and Group Access

The following example shows entries in an ACL:

```
<acl description="myacl"
  xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
  xmlns:dav="DAV:"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
    http://xmlns.oracle.com/xdb/acl.xsd">
  <ace>
    <principal>OWNER</principal>
    <grant>true</grant>
    <privilege>
      <all/>
    </privilege>
  </ace>
</acl>
```

ACE Elements Specify Access Privileges for Principals

The preceding ACL grants all privileges to the owner of the document. Access to an Oracle XML DB resource is granted for each principal. [Table 18–1](#) lists the access control entry (ACE) elements. Each ACE element specifies access privileges for a given principal using values set for the following elements.

Table 18–1 Access Control Entry (ACE) Elements

Element	Description
<code><principal></code>	Specifies the principal (user or group).
<code><grant></code>	A boolean value that specifies whether the principal has been granted access to the resource. A value of <code>true</code> specifies that the access is granted. A value of <code>false</code> specifies that access is denied.
<code><privilege></code>	Specifies the privileges granted to the principal.

Oracle XML DB Supported Privileges

Oracle XML DB provides a set of privileges to control access to Oracle XML DB resources. Access privileges in an ACE are stored in the privilege element. Privileges can be:

- Aggregate, composed of other privileges
- Atomic, cannot be subdivided

When an ACL is stored in Oracle XML DB, the aggregate privileges retain their identity, that is, they are not decomposed into the corresponding leaf privileges. In WebDAV terms, these are non-abstract aggregate privileges, so they can be used in ACEs.

Atomic Privileges

[Table 18–2](#) lists the atomic privileges supported by Oracle XML DB.

Table 18–2 Atomic Privileges

Privilege Name	Description	Database Counterpart
read-properties	Read the properties of a resource	SELECT
read-contents	Read the contents of a resource	SELECT
update	Update the properties and contents of a resource	UPDATE
link	For containers only. Allows resources to be bound to the container	INSERT
unlink	For containers only. Allows resources to be unbound from the container	DELETE
linkto	Allows resources to be linked	N/A
unlinkfrom	Allows resources to be unlinked	N/A
read-acl	Read the resource's ACL	SELECT
write-acl-ref	Changes the resource's ID	UPDATE
update-acl	Change the contents of the resource's ACL	UPDATE
resolve	For containers only: Allows the container to be traversed	SELECT
dav:lock	Lock a resource using WebDAV locks	UPDATE
dav:unlock	Unlock a resource locked using a WebDAV lock	UPDATE

Note: Privilege names are XML element names. Privileges with a `dav:` prefix are part of the WebDAV namespace and others are part of the Oracle XML DB ACL namespace, which is:
`http://xmlns.oracle.com/xdb/acl.xsd`

Since you can directly access the `XMLType` storage for ACLs, the XML structure is part of the client interface. Hence ACLs can be manipulated using `XMLType` APIs.

Aggregate Privileges

[Table 18-3](#) lists the aggregate privileges defined by Oracle XML DB, along with the atomic privileges of which they are composed.

Table 18-3 Aggregate Privileges

Aggregate Privilege Name	Atomic Privileges
all	All atomic privileges
dav:all	All atomic privileges except linkto
dav:read	read-properties, read-contents, resolve
dav:write	update, link, unlink, unlinkfrom
dav:read-acl	read-acl
dav:write-acl	write-acl-ref, update-acl

[Table 18-4](#) shows the privileges required for some common operations on resources in Oracle XML DB Repository. The Privileges Required column assumes that you already have `resolve` privilege on container `C` and all its parent containers, up to the root of the hierarchy.

Table 18-4 Privileges Needed for Operations on Oracle XML DB Resources

Operation	Description	Privileges Required
CREATE	Create a new resource in container <code>C</code>	update and link on <code>C</code>
DELETE	Delete resource <code>R</code> from container <code>C</code>	update and unlinkfrom on <code>R</code> , update and unlink on <code>C</code>
UPDATE	Update the contents/properties of resources <code>R</code>	update on <code>R</code>

Table 18–4 Privileges Needed for Operations on Oracle XML DB Resources (Cont.)

Operation	Description	Privileges Required
GET	An FTP/HTTP GET of resource R	read-properties, read-contents on R
SET_ACL	Set the ACL of a resource R	dav:write-acl on R
LIST	List the resources in container C	read-properties on C, read-properties on resources in C. Only those resources on which the user has read-properties privilege are listed.

ACL Evaluation Rules

To evaluate an ACL, the database collects the list of ACEs applying to the user logged into the current database session. The list of currently active roles for the given user is maintained as a part of the session and is used to match ACEs with the current users. To resolve conflicts between ACEs, the following rule is used: if a privilege is denied by any ACE, the privilege is denied for the entire ACL.

Entries in an ACL must observe the following rule:

- Each principal will have two individual ACEs at most, one for granting privileges and one for denying privileges.
- Multiple grant ACEs are not allowed for any principal.
- Multiple deny ACEs are not allowed for any principal.

Using Oracle XML DB ACLs

Every resource in the Oracle XML DB Repository hierarchy has an associated ACL. The ACL mechanism specifies a privilege-based access control for resources to principals. Whenever a resource is accessed, a security check is performed. The ACL determines which principals have which set of privileges to access the resource. An Oracle XML DB principal can be either of the following:

- An individual principal, such as a database user
- A group principal, such as a group of database users that are granted a common role

Each ACL has a list of ACEs. An ACE has the following elements:

- A boolean value indicating whether or not this ACE is granting or denying privileges

- A principal (either a user or a role) indicating to whom the ACL applies
- A list of privileges that are being granted or denied

Named ACLs also have a name attribute and an optional type restrictor, for example, `http://xmlns.oracle.com/xdm/XDBDemo.xsd#PurchaseOrder`, that specifies that the ACL may only be applied to instances of that XML element (and elements in a substitution group with that element). Note that a privilege that is neither granted nor denied to a user is assumed to be denied.

To evaluate an ACL, the database collects the list of ACEs applying to the user logged into the current database session. The list of currently active roles for the given user is maintained as a part of the session and is used to match ACEs along with the current user.

To check if a user has a certain privilege, you need to know the ID of the ACL and the owner of the object being secured. The Oracle XML DB hierarchy automatically associates an ACL ID and owner with an object that is mapped into its file system (they are stored in a table in the Oracle XML DB schema).

Updating the Default ACL on a Folder

Example 18–2 Updating the Default ACL on a Folder and the Owner of the Folder

This example creates two users, Oracle XML DB administrator, `xdbadmin`, and Oracle XML DB user, `xdbuser`. The administrator creates the user's folder under `/'`. The default ACL on this folder, inherited from the parent container, allows:

- All permissions to the owner
- Only read permission to public

The owner of the folder is changed to the user, by updating the `resource_view`. You can also make the user's folder completely private by changing the ACL to another system ACL, such as, `all_owner_acl.xml`

```
connect system/manager
```

```
Rem Create an Oracle XML DB administrator user (has XDBADMIN role)
grant connect, resource, xdbadmin to xdbadm identified by xdbadm;
```

```
Rem Create Oracle XML DB user
grant connect, resource to xdbuser identified by xdbuser;
```

```
conn xdbadm/xdbadm
```

```

Rem create the user's folder
declare
retval boolean;
begin
retval := dbms_xdb.createfolder('/xdbuser');
end;
/

Rem update the OWNER of the user folder
update resource_view
set res = updatexml(res, '/Resource/Owner/text()', 'XDBUSER')
where any_path = '/xdbuser';

commit;

connect xdbuser/xdbuser

Rem XDBUSER has full permissions to operate on her folder
declare
retval boolean;
begin
retval := dbms_xdb.createfolder('/xdbuser/workdir');
end;
/

Rem All users can read /xdbuser folder at this time.
Rem change ACL to make folder completely private
call dbms_xdb.setacl('/xdbuser', '/sys/acls/all_owner_acl.xml');

```

ACL and Resource Management

The following subsections describe ACL and resource management in the Oracle XML DB Repository.

See Also: [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)

How to Set Resource Property ACLs

Any Oracle XML DB resource has an ACL as a resource property. Therefore, in order to set the ACL resource property, any of the following methods can be used:

- Resource view update

- The ACL bean in the Java resource view API
- PL/SQL API: Use `DBMS_XDB.SETACL(res_path VARCHAR2, acl_path VARCHAR2)` to set the ACL property of the resource represented by `res_path` to the ACL represented by `acl_path`.
- Quote command in FTP: Use “`quote sacl <res_path> <acl_path>`” in order to set the ACL resource property of `res_path` to the ACLOID for `acl_path`

Default Assignment of ACLs

When a resource is inserted into the Oracle XML DB hierarchy, and the resource does not specify an ACL, it shares the ACL of its parent container.

Retrieving ACLs for a Resource

The following `DBMS_XDB` API can be used to get the ACL for a given resource:

```
DBMS_XDB.getAclDocument(res_path IN VARCHAR2)
```

It returns an `XMLType` instance of `<acl>` element representing the ACL for the resource at `res_path`.

Changing Privileges on a Given Resource

The following `DBMS_XDB` API can be used to add an ACE to a resource's ACL:

```
DBMS_XDB.changePrivileges(res_path IN VARCHAR2, ace IN XMLType)
```

Restrictions for Operations on ACLs

All named ACLs are XML schema-based resources in the Oracle XML DB Repository hierarchy. Every method used for other resources in Oracle XML DB Repository hierarchy can also be used for ACLs. For example, FTP commands, JNDI, PL/SQL DOM, and `XMLType` methods can operate on ACLs. However, because ACLs are part of the access control security scheme and the Oracle XML DB Repository hierarchy, the following restrictions are enforced:

- ACL insertion: Can be at most one grant ACE and one deny ACE for a particular principal in an ACL
- ACL deletion: If a resource is currently using the ACL, the ACL cannot be deleted.

- ACL update (modify): If an ACL resource is updated with non-ACL content, the same rules as for ACL deletion will apply.

Using DBMS_XDB to Check Privileges

You can enforce Oracle XML DB access control using the following DBMS_XDB functions:

- `CheckPrivileges`, `getAclDocument`, and `getPrivileges` for Oracle XML DB resources.
- `AclCheckPrivileges` for database objects. This function loads the ACL from the cache, and performs the access request evaluation as described in the next section.

Row-Level Security for Access Control Security

ACL security in Oracle XML DB acts in conjunction with database security for XML objects. The user must have the appropriate rights on the underlying table/view where the XML object is stored as well as permissions in the ACL for that individual instance. When an object from a particular table is first stored in the Oracle XML DB hierarchy (and mapped to a resource), a row-level security (RLS) policy is added to that table that checks ACL-based permission only for those rows in the table that are mapped to a resource. RLS is enforced for `XMLType` tables/views that are part of the Oracle XML DB hierarchy.

Using FTP, HTTP, and WebDAV Protocols

This chapter describes how to access Oracle XML DB Repository using FTP, HTTP/WebDAV protocols. It contains the following sections:

- [Introducing Oracle XML DB Protocol Server](#)
- [Using FTP and Oracle XML DB Protocol Server](#)
- [Using HTTP and Oracle XML DB Protocol Server](#)
- [Using WebDAV and Oracle XML DB](#)

Introducing Oracle XML DB Protocol Server

As described in [Chapter 2, "Getting Started with Oracle XML DB"](#) and [Chapter 13, "Oracle XML DB Foldering"](#), Oracle XML DB Repository provides a hierarchical data repository in the database modeled on XML. Oracle XML DB Repository maps path names (or URLs) onto database objects of XMLType and provides management facilities for these objects.

Oracle XML DB also provides the Oracle XML DB Protocol Server. This supports standard Internet protocols, FTP, WebDAV, and HTTP, for accessing its hierarchical repository/ file system. Since XML documents reference each other using URLs, typically HTTP URLs, Oracle XML DB Repository and its protocol support are important Oracle XML DB components. These protocols can provide direct access to Oracle XML DB to many users without having to install additional software.

See Also: ["Accessing Oracle XML DB Resources Using Internet Protocols"](#) on page 13-10

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.

HTTP Performance is Improved

Session Pooling improves performance of HTTP by avoiding the cost of re-creating session states, especially when using HTTP 1.0, which creates new connections for each request. For example, a couple of small files can be retrieved by an existing HTTP/1.1 connection in the time necessary to create a database session. You can tune the number of sessions in the pool by setting session-pool-size in Oracle XML DB's `xdbconfig.xml` file, or disable it by setting pool size to zero.

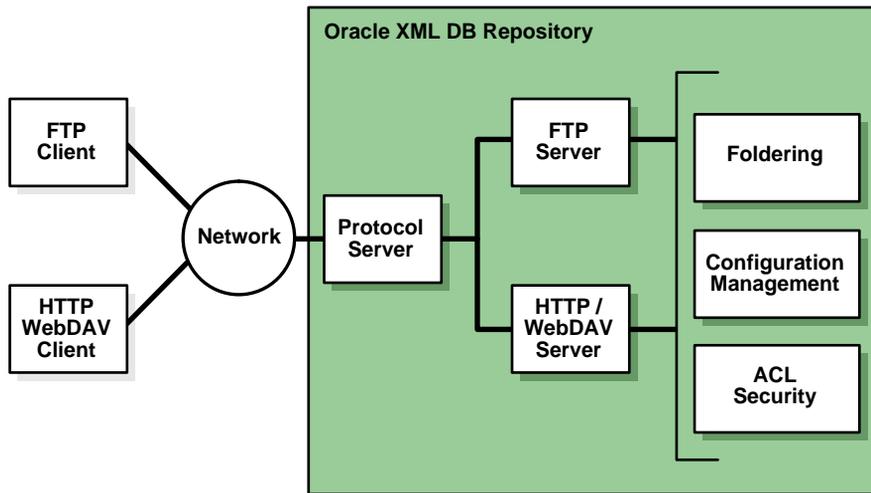
Java Servlets

Session pooling can affect users writing Java servlets, since other users can come along and 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. Per user state must be stored in the database or in a look-up table rather than assuming a session will only exist for a single user.

See Also: [Chapter 20, "Writing Oracle XML DB Applications in Java"](#)

Figure 19-1 illustrates the Oracle XML DB Protocol Server components and how they are used to access files in Oracle XML DB XML Repository and other data. Only the relevant components of the Repository are shown.

Figure 19-1 Oracle XML DB Architecture: Protocol Server



Oracle XML DB Protocol Server Configuration Management

Oracle XML DB Protocol Server uses configuration parameters stored in `/xdbconfig.xml` to initialize its startup state and manage session level configuration. The following section describes the protocol-specific configuration parameters that you can configure in the Oracle XML DB configuration file.

See Also: [Appendix A, "Installing and Configuring Oracle XML DB"](#).

Configuring Protocol Server Parameters

Table 19–1 shows the parameters common to all protocols. All parameter names in this table, except those starting with `/xdbconfig`, are relative to the following XPath in the Oracle XML DB configuration schema:

```
/xdbconfig/sysconfig/protocolconfig/common
```

- **FTP-specific parameters.** **Table 19–2** shows the FTP-specific parameters. These are relative to the following XPath in the Oracle XML DB configuration schema:

```
/xdbconfig/sysconfig/protocolconfig/ftpconfig
```

- **HTTP/WebDAV specific parameters except servlet-related parameters.** **Table 19–3** shows the HTTP/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`, in Oracle XML DB Repository.

See Also: [Appendix A, "Installing and Configuring Oracle XML DB"](#)

Table 19–1 Common Protocol Configuration Parameters

Parameter	Description
extension-mappings/mime-mappings	Specifies the mapping of file extensions to mime types. When a resource is stored in the 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 the 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 the Oracle XML DB Repository, and its encoding is not specified, this list of mappings is used to set its encoding.

Table 19–1 Common Protocol Configuration Parameters (Cont.)

Parameter	Description
extension-mappings/charset-mappings	Specifies the mapping of file extensions to character sets. When a resource is stored in the Oracle XML DB Repository, and its character set is not specified, this list of mappings is used to set its character set.
session-pool-size	Maximum number of sessions that are kept in the protocol server's session pool
/xdbconfig/sysconfig/call-timeout	If a connection is idle for this time (in hundredths of a second), 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) will be terminated by the protocol server if the connection has been idle for that time. This parameter is used only if the specific protocol's session timeout is not present in the configuration
/xdbconfig/sysconfig/default-lock-timeout	Time 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 19–2 Configuration Parameters Specific to FTP

Parameter	Description
ftp-port	Port on which FTP server listens. By default this is 2100
ftp-protocol	Protocol over which the FTP server runs. By default this is tcp
session-timeout	Time (in hundredths of a second) after which an FTP session (and consequently the corresponding connection) will be terminated by the protocol server if the connection has been idle for that time.

Table 19–3 Configuration Parameters Specific to HTTP/WebDAV (Except Servlet Parameters)

Parameter	Description
http-port	Port on which HTTP/WebDAV server listens

Table 19–3 Configuration Parameters Specific to HTTP/WebDAV (Except Servlet Parameters) (Cont.)

Parameter	Description
http-protocol	Protocol over which the HTTP/WebDAV server runs. By default this is tcp
session-timeout	Time (in hundredths of a second) after which an HTTP session (and consequently the corresponding connection) will be terminated by the protocol server if the connection has been idle for that time.
server-name	The value of the Server header in an HTTP response
max-header-size	Maximum size (in bytes) of an HTTP header
max-request-body	Maximum size (in bytes) of an HTTP request body
webappconfig/welcome-file-list	List of filenames that are considered “welcome files”. When an HTTP GET request for a container is received, the server first checks if there’s a resource in the container with any of these names. If so, the contents of that file are sent, instead of a list of resources in the container.

Interaction with Oracle XML DB Filesystem Resources

The protocol specifications, RFC 959 (FTP), RFC 2616 (HTTP), and RFC 2518 (WebDAV) implicitly assume an abstract, hierarchical file system on the server side. This is mapped to the Oracle XML DB hierarchical Repository. Oracle XML DB Repository provides features such as:

- Name resolution
- ACL-based security
- The ability to store and retrieve any content. Oracle XML DB Repository can store both binary data input through FTP and XML schema-based documents.

See Also:

- <http://rfc.sunsite.dk/rfc/rfc959.html>
- <http://256.com/gray/docs/rfc2616/>
- <http://www.faqs.org/rfcs/rfc2518.html>

Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents

Oracle XML DB Protocol Server enhances the protocols by always checking if XML documents being inserted are based on XML schemas registered in the Repository.

- If the incoming XML document specifies an XML schema, the Oracle XML DB storage to use is decided by that XML schema. This functionality comes in handy when you need to store XML documents object-relationally in the database, using simple protocols like FTP or WebDAV instead of having to write SQL statements.
- If the incoming XML document is not XML schema-based, it is stored as a binary document.

Using FTP and Oracle XML DB Protocol Server

The following sections describe FTP features supported by Oracle XML DB.

Oracle XML DB Protocol Server: FTP Features

File Transfer Protocol (FTP) is one of the oldest and most popular protocols on the net. FTP is specified in RFC959 and provides access to heterogeneous file systems in a uniform manner. FTP works by providing well defined commands for communication between the client and the server. The transfer of commands and the return status happens on a single connection. However, a new connection is opened between the client and the server for data transfer. In HTTP, the transfer of commands and data happens on a single connection.

FTP is implemented by both dedicated clients at the operating system level, file system explorer clients, and browsers. FTP is typically session-oriented, in that a user session is created through an explicit logon, a number of files / directories are downloaded and browsed, and then the connection is closed.

See Also: RFC 959: FTP Protocol Specification

Non-Supported FTP Features

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 command 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.

Using FTP on Standard or Non-Standard Ports

It can be configured through the Oracle XML DB configuration file `/xdbconfig.xml`, to listen on an arbitrary port. FTP ships listening on a non-standard, non-protected port. To use FTP on the standard port (21), your DBA has to `chown` the TNS listener to `setuid ROOT` rather than `setuid ORACLE`.

FTP Server Session Management

Protocol Server also provides session management for this protocol. After a short wait for a new command, FTP returns to the protocol layer and the shared server is freed up to serve other connections. The duration of this short wait is configurable by changing the `call-timeout` parameter in the Oracle XML DB configuration file. For high traffic sites, the `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.

Using HTTP and Oracle XML DB Protocol Server

Oracle XML DB implements HyperText Transfer Protocol (HTTP), HTTP 1.1 as defined in RFC2616 specification. In this release, Oracle XML DB Protocol Server also supports the HTTP protocol extension, RFC 2109 “HTTP State Management”, that is “cookies”.

Oracle XML DB Protocol Server: HTTP Features

The Oracle XML DB HTTP component in the Oracle XML DB Protocol Server implements the RFC2616 specification with the exceptions of the following optional features:

- `gzip` and `compress` transfer encodings
- `byte-range` headers

- The TRACE method (used for proxy error debugging)
- Cache-Control directives (requires you to specify expiration dates for content, and are not generally used)
- TE, Trailer, Vary & Warning headers
- Weak entity tags
- Web common log format
- Multi-homed Web server

See Also: RFC 2616: HTTP 1.1 Protocol Specification

Non-Supported HTTP Features

Oracle XML DB does not implement the new Set-Cookie2 header specified in RFC 2965, as most of the Internet community is not yet using it. Digest Authentication (RFC 2617) is not supported. In this release, Oracle XML DB supports Basic Authentication, where a client sends the user name and password in clear text in the “Authorization” header.

Using HTTP on Standard or Non-Standard Ports

HTTP ships listening on a non-standard, non-protected port (8080). To use HTTP on the standard port (80), your DBA must `chown` the TNS listener to `setuid ROOT` rather than `setuid ORACLE`, and configure the port number in the Oracle XML DB configuration file `/xdbconfig.xml`.

HTTP Server and Java Servlets

Oracle XML DB supports Java servlets. To use a servlet, it must be registered with a unique name in the Oracle XML DB configuration file, along with parameters to customize its behavior. It should be compiled, and loaded into the database. Finally, the servlet name must be associated with a pattern, which can be an extension such as `*.jsp` or a path name such as `/a/b/c` or `/sys/*`, as described in Java servlet API version 2.2.

While processing an HTTP request, the path name for the request is matched with the registered patterns. If there is a match, the Protocol Server invokes the corresponding servlet with the appropriate initialization parameters. For Java servlets, the existing Java Virtual Machine (JVM) infrastructure is used. This starts the JVM if need be, which in turn runs a Java method to initialize the servlet, create response, and request objects, pass these on to the servlet, and run it.

See Also: [Chapter 20, "Writing Oracle XML DB Applications in Java"](#)

Using WebDAV and Oracle XML DB

Web Distributed Authoring and Versioning (WebDAV) is a standard protocol used to provide users with a file system interface to Oracle XML Repository over the Internet. The most popular way of accessing a WebDAV server folder is through “WebFolders” on Microsoft Windows 2000 or Microsoft NT.

WebDAV is an extension to HTTP 1.1 protocol. It allows clients to perform remote web content authoring through a coherent set of methods, headers, request body formats and response body formats. WebDAV provides operations to store and retrieve resources, create and list contents of resource collections, lock resources for concurrent access in a coordinated manner, and to set and retrieve resource properties.

Oracle XML DB WebDav Features

Oracle XML DB supports the following WebDAV features:

- Foldering, specified by RFC2518
- Access Control

WebDAV is a set of extensions to the HTTP protocol that allow you to edit or manage your files on remote Web servers. WebDav can also be used, for example, to:

- Share documents over the Internet
- Edit content over the Internet

See Also: [RFC 2518: WebDAV Protocol Specification](#)

Oracle XML DB's Non-Supported WebDAV Features

Oracle XML DB supports the contents of RFC2518, with the following exceptions:

- Lock-NULL resources create actual zero-length resources in the file system, and cannot be converted to folders.
- Implementing the WebDAV ACL protocol and binding protocol
- Depth-infinity locks

- Only Basic Authentication is supported

Using Oracle XML DB and WebDAV: Creating a WebFolder in Windows 2000

To create a WebFolder in Windows 2000, follow these steps:

1. From your desktop, select My Network Places.
2. Double click “Add Network Place”.
3. Type the location of the folder, for example:
`http://[Oracle server name]:<HTTP port number>`

See [Figure 19-2](#).

4. Click on Next.
5. Enter any name to identify this WebFolder
6. Click on Finish.

You can now access Oracle XML DB Repository just like you access any Windows folder.

Figure 19–2 *Creating a WebFolder in Windows 2000*



Writing Oracle XML DB Applications in Java

This chapter describes how to write Oracle XML DB applications in Java. It includes design guidelines for writing Java applications including servlets, and how to configure the Oracle XML DB servlets.

It contains these sections:

- [Introducing Oracle XML DB Java Applications](#)
- [Design Guidelines: Java Inside or Outside the Database?](#)
- [Writing Oracle XML DB HTTP Servlets in Java](#)
- [Configuring Oracle XML DB Servlets](#)
- [HTTP Request Processing for Oracle XML DB Servlets](#)
- [The Session Pool and XML DB Servlets](#)
- [Native XML Stream Support](#)
- [Oracle XML DB Servlet APIs](#)
- [Oracle XML DB Servlet Example](#)

Introducing Oracle XML DB Java Applications

Oracle XML DB provides two main architectures for the Java programmer:

- In the database using the Java Virtual Machine (VM)
- In a client or application server, using the Thick JDBC driver

Because Java in the database runs in the context of the database server process, the methods of deploying your Java code are restricted to one of the following ways:

- You can run Java code as a stored procedure invoked from SQL or PL/SQL or
- You can run a Java servlet.

Stored procedures are easier to integrate with SQL and PL/SQL code, and require using Oracle Net Services as the protocol to access Oracle9i database.

Servlets work better as the top level entry point into Oracle9i database, and require using HTTP as the protocol to access Oracle9i database.

Which Oracle XML DB APIs Are Available Inside and Outside the Database?

In this release, some of Oracle XML DB APIs are only available to applications running in the server. [Table 20–1](#) illustrates which Oracle XML DB APIs are available in each architecture in this release. The “NO” fields will become “YES” in a forthcoming release.

Oracle XML DB APIs Available Inside and Outside Oracle9i Database

Table 20–1 Oracle XML DB APIs Inside and Outside Oracle9i Database

Java Oracle XML DB APIs Description	Inside the Database:Java Servlets/Stored Procedures	Outside the Database: Using Thick JDBC (OCI) Drivers
JDBC support for XMLType	YES	YES
XMLType class	YES	YES
Java DOM implementation	YES	YES
JNDI access to repository	YES	NO
Java Bean structured XML access	YES	NO

Design Guidelines: Java Inside or Outside the Database?

When choosing an architecture for writing Java Oracle XML DB applications, consider the following guidelines:

HTTP: Accessing Java Servlets or Directly Accessing XMLType Resources

If the downstream client wants to deal with XML in its textual representation, using HTTP to either access the Java servlets or directly access `XMLType` resources, will perform 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), and in many cases copies data directly from the database buffer cache to the HTTP connection. There is no need to convert data from the buffer cache into the SQL serialization format used by Oracle Net Services, move it to the JDBC client, and then convert to XML. The load-on-demand and LRU cache for `XMLType` are most effective inside the database server.

Accessing Many XMLType Object Elements: Use JDBC XMLType Support

If the downstream client is an application that will programmatically access many or most of the elements of an `XMLType` object using Java, using JDBC `XMLType` support will probably perform the best. It is often easier to debug Java programs outside of the database server, as well.

Use the Servlets to Manipulate and Write Out Data Quickly as XML

Oracle XML DB servlets are intended for writing “HTTP stored procedures” in Java that can be accessed using HTTP. They are not intended as a platform for developing an entire Internet application. In that case, the application servlet should be deployed in Oracle9iAS application server and access data in the database either using JDBC, or by using the `java.net.*` or similar APIs to get XML data through HTTP.

They are best used for applications that want to get into the database, manipulate the data, and write it out quickly as XML, not to format HTML pages for end-users.

Writing Oracle XML DB HTTP Servlets in Java

Oracle XML DB provides a Protocol Server that supports FTP, HTTP 1.1, WebDAV, and Java Servlets. The support for Java Servlets in this release is not complete, and provides a subset designed for easy migration to full compliance in a following release. Currently, Oracle XML DB supports Java Servlet version 2.2, with the following exceptions:

- The Servlet WAR file (`web.xml`) is not supported in its entirety. Some `web.xml` configuration parameters must be handled manually. For example, creating roles must be done using the SQL `CREATE ROLE` command.
- Only one `ServletContext` (and one `web-app`) is currently supported.
- Stateful servlets (and thus the `HttpSession` class methods) are not supported. Servlets must maintain state in the database itself.

Configuring Oracle XML DB Servlets

Oracle XML DB servlets are configured using the `/xdbconfig.xml` file in the Repository. Many of the XML elements in this file are the same as those defined by the Java Servlet 2.2 specification portion of Java 2 Enterprise Edition (J2EE), and have the same semantics. [Table 20-2](#) 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 20-2 XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
<code>auth-method</code>	Java	no	Specifies an HTTP authentication method required for access	--
<code>charset</code>	Oracle	yes	Specifies a IANA character set name	For example: "ISO8859", "UTF8"
<code>charset-mapping</code>	Oracle	yes	Specifies a mapping between a filename extension and a charset	--
<code>context-param</code>	Java	no	Specifies a parameter for a web application	Not yet supported
<code>description</code>	Java	yes	A string for describing a servlet or Web application	Supported for servlets
<code>display-name</code>	Java	yes	A string to display with a servlet or web app	Supported for servlets

Table 20–2 XML Elements Defined for Servlet Deployment Descriptors (Cont.)

XML Element Name	Defined By	Supported?	Description	Comment
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 the Oracle9i database MUST be distributable.
errnum	Oracle	yes	Oracle error number	See <i>Oracle9i Database Error Messages</i>
error-code	Java	yes	HTTP 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 error, an uncaught Java exception, or through an uncaught Oracle error message
exception-type	Java	yes	Classname of a Java exception mapped to an error page	--
extension	Java	yes	A filename extension used to associate with MIME types, character sets, and so on.	--
facility	Oracle	yes	Oracle facility code for mapping error pages	For example: "ORA", "PLS", and so on.
form-error-page	Java	no	Error page for form login attempts	Not yet supported
form-login-config	Java	no	Config spec for form-based login	Not yet supported
form-login-page	Java	no	URL for the form-based login page	Not yet supported
icon	Java	Yes	URL of icon to associate with a servlet	Supported for servlets
init-param	Java	Yes	Initialization parameter for a servlet	--
jsp-file	Java	No	Java Server Page file to use for a servlet	Not supported
lang	Oracle	Yes	IANA language name	For example: "en-US"
lang-mapping	Oracle	Yes	Specifies a mapping between a filename extension and language content	--
large-icon	Java	Yes	Large sized icon for icon display	--

Table 20–2 XML Elements Defined for Servlet Deployment Descriptors (Cont.)

XML Element Name	Defined By	Supported?	Description	Comment
load-on-startup	Java	Yes	Specifies if a servlet is to be loaded on startup	--
location	Java	Yes	Specifies the URL for an error page	Can be a local path name or HTTP URL
login-config	Java	No	Specifies a method for authentication	Not yet supported
mime-mapping	Java	Yes	Specifies a mapping between filename extension and the MIME type of the content	--
mime-type	Java	Yes	MIME type name for resource content	For example: "text/xml" or "application/octet-stream"
OracleError	Oracle	Yes	Specifies an Oracle error to associate with an error page	--
param-name	Java	Yes	Name of a parameter for a Servlet or ServletContext	Supported for servlets
param-value	Java	Yes	Value of a parameter	--
realm-name	Java	No	HTTP realm used for authentication	Not yet supported
role-link	Java	Yes	Specifies a role a particular user must have in order to access a servlet	Refers to a database role name. Make sure to capitalize by default!
role-name	Java	Yes	A servlet name for a role	Just another name to call the database role. Used by the Servlet APIs
security-role	Java	No	Defines a role for a servlet to use	Not supported. You must manually create roles using the SQL CREATE ROLE
security-role-ref	Java	Yes	A reference between a servlet and a role	--
servlet	Java	Yes	Configuration information for a servlet	--

Table 20–2 XML Elements Defined for Servlet Deployment Descriptors (Cont.)

XML Element Name	Defined By	Supported?	Description	Comment
servlet-class	Java	Yes	Specifies the classname for the Java servlet	--
servlet-language	Oracle	Yes	Specifies the programming language in which the servlet is written.	Either “Java”, “C”, or “PL/SQL”. Currently, only Java is supported for customer-defined servlets.
servlet-mapping	Java	Yes	Specifies a filename pattern with which to associate the servlet	All of the mappings defined by Java are supported
servlet-name	Java	Yes	String name for a servlet	Used by Servlet APIs
servlet-schema	Oracle	Yes	The Oracle Schema in which the Java class is loaded. If not specified, the schema is searched using the default resolver specification.	If this is not specified, the servlet must be loaded into the SYS schema to ensure that everyone can access it, or the default Java class resolver must be altered. Note that the servlet-schema is capitalized unless the value is quoted with double-quotes.
session-config	Java	No	Configuration information for an <code>HTTPSession</code>	<code>HTTPSession</code> is not supported
session-timeout	Java	No	Timeout for an HTTP session	<code>HTTPSession</code> is not supported
small-icon	Java	Yes	Small icon to associate with a servlet	--
taglib	Java	No	JSP tag library	JSPs currently not supported
taglib-uri	Java	No	URI for JSP tag library description file relative to the web.xml file	JSPs currently not supported
taglib-location	Java	No	Pathname 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

Table 20–2 XML Elements Defined for Servlet Deployment Descriptors (Cont.)

XML Element Name	Defined By	Supported?	Description	Comment
web-app	Java	No	Configuration for a web application	Only one web application is currently supported
welcome-file	Java	Yes	Specifies a welcome-file name	--
welcome-file-list	Java	Yes	Defines a list of files to display when a folder is referenced through an HTTP GET	Example: "index.html"

Note:

- **Note 1:** The following parameters defined for the `web.xml` file by Java are usable only by J2EE-compliant EJB containers, and are not required for Java Servlet Containers that do not support a full J2EE environment: *env-entry*, *env-entry-name*, *env-entry-value*, *env-entry-type*, *ejb-ref*, *ejb-ref-type*, *home*, *remote*, *ejb-link*, *resource-ref*, *res-ref-name*, *res-type*, *res-auth*
- **Note 2:** The following elements are used to define access control for resources: *security-constraint*, *web-resource-collection*, *web-resource-name*, *http-method*, *user-data-constraint*, *transport-guarantee*, *auth-constrain*. Oracle XML DB provides this functionality through Access Control Lists (ACLs). A future release will support using a `web.xml` file to generate ACLs

See Also: [Appendix A, "Installing and Configuring Oracle XML DB"](#) for more information about configuring the `/XDBconfig.xml` file.

HTTP Request Processing for Oracle XML DB Servlets

Oracle XML DB handles an HTTP request using the following steps:

1. If a connection has not yet been established, 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 XML DB session pool, if available, but otherwise will create a new session.
5. A new database call is started, as well as a new database transaction.
6. If HTTP has included authentication headers, the session will be authenticated as that database user (just as if they logged into SQL*Plus). If no authentication information is included, and the request is GET or HEAD, Oracle XML DB attempts to authenticate the session as the ANONYMOUS user. If that database user account is locked, 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 XML DB Servlet Container is invoked in the Java VM inside Oracle. If the specified servlet has not been initialized yet, the servlet is initialized.
9. The Servlet reads input from the `ServletInputStream`, and writes output to the `ServletOutputStream`, and returns from the `service()` method.
10. If no uncaught Oracle error occurred, the session is put back into the session pool.

See Also: [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#)

The Session Pool and XML DB Servlets

The Oracle database keeps one Java VM for each database session. This means that a session reused from the session pool will have any state in the Java VM (Java static variables) from the last time the 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

The `DOM Node` class has an Oracle-specific method called `write()`, which takes the following arguments, returning `void`:

- `java.io.OutputStream stream`: A Java stream to write the XML text to
- `String charEncoding`: The character encoding to write the XML text in. If null, the database character set is used

- `Short indent` The number of characters to indent nested XML elements

This method has a shortcut implementation if the stream provided is the `ServletOutputStream` provided inside the database. The contents of the Node are written in XML in native code directly to the output socket. This bypasses any conversions into and out of Java objects or Unicode (required for Java strings) and provides very high performance.

Oracle XML DB Servlet APIs

The APIs supported by Oracle XML DB servlets are defined by the Java Servlet 2.2 specification, the Javadoc for which is available, as of the time of writing this, online at:

<http://java.sun.com/products/servlet/2.2/javadoc/index.html>

[Table 20–3](#) lists non-implemented Java Servlet 2.2 methods. In this release they result in runtime exceptions.

Table 20–3 Non-Implemented Java 2.2 Methods

Interface	Methods
<code>HttpServletRequest</code>	<code>getSession()</code> , <code>isRequestedSessionIdValid()</code>
<code>HttpSession</code>	ALL
<code>HttpSessionBindingListener</code>	ALL

Oracle XML DB Servlet Example

The following is a simple servlet example that reads a parameter specified in a URL as a path name, and writes out the content of that XML document to the output stream.

Example 20–1 Writing an Oracle XML DB Servlet

The servlet code looks like:

```
/* test.java */
import javax.servlet.http.*;
import javax.servlet.*;
import java.util.*;
import java.io.*;
import javax.naming.*;
import oracle.xdb.dom.*;
```

```

public class test extends HttpServlet
{
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        OutputStream os = resp.getOutputStream();
        Hashtable env = new Hashtable();
        XDBDocument xt;

        try
        {
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                "oracle.xdb.spi.XDBContextFactory");
            Context ctx = new InitialContext(env);
            String [] docarr = req.getParameterValues("doc");
            String doc;

            if (docarr == null || docarr.length == 0)
                doc = "/foo.txt";
            else
                doc = docarr[0];
            xt = (XDBDocument)ctx.lookup(doc);
            resp.setContentType("text/xml");
            xt.write(os, "ISO8859", (short)2);
        }
        catch (javax.naming.NamingException e)
        {
            resp.sendError(404, "Got exception: " + e);
        }
        finally
        {
            os.close();
        }
    }
}

```

Installing the Oracle XML DB Example Servlet

To install this servlet, compile it, and load it into Oracle9i database using commands such as:

```
% loadjava -grant public -u scott/tiger -r test.class
```

Configuring the Oracle XML DB Example Servlet

To configure Oracle XML DB servlet, update the `/xdbconfig.xml` file by inserting the following XML element tree in the `<servlet-list>` element:

```
<servlet>
  <servlet-name>TestServlet</servlet-name>
  <servlet-language>Java</servlet-language>
  <display-name>XML DB Test Servlet</display-name>
  <servlet-class>test</servlet-class>
  <servlet-schema>scott</servlet-schema>
</servlet>
```

and update the `/xdbconfig.xml` file by inserting the following XML element tree in the `<servlet-mappings>` element:

```
<servlet-mapping>
  <servlet-pattern>/testserv</servlet-pattern>
  <servlet-name>TestServlet</servlet-name>
</servlet-mapping>
```

You can edit the `/xdbconfig.xml` file with any WebDAV-capable text editor, or by using the `updateXML()` SQL operator.

Note: You will not be allowed to actually delete the `/xdbconfig.xml` file, even as SYS!

Testing the Example Servlet

To test the example servlet, load an arbitrary XML file at `/foo.xml`, and type the following URL into your browser, replacing the `hostname` and port number as appropriate:

```
http://hostname:8080/testserv?doc=/foo.xml
```

Part VI

Oracle Tools that Support Oracle XML DB

Part VI of this manual introduces you to Oracle SQL*Loader for loading your XML data. It also describes how to use Oracle Enterprise Manager for managing and administering your XML database applications.

Part VI contains the following chapters:

- [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)
- [Chapter 22, "Loading XML Data into Oracle XML DB"](#)

Managing Oracle XML DB Using Oracle Enterprise Manager

This chapter describes how Oracle Enterprise Manager can be used to manage Oracle XML DB. Oracle Enterprise Manager can be used to configure, create and manage Repository resources, and database objects such as XML schemas and XMLType tables.

It contains the following sections:

- [Introducing Oracle XML DB and Oracle Enterprise Manager](#)
- [Oracle Enterprise Manager Oracle XML DB Features](#)
- [The Enterprise Manager Console for Oracle XML DB](#)
- [Configuring Oracle XML DB with Enterprise Manager](#)
- [Creating and Managing Oracle XML DB Resources with Enterprise Manager](#)
- [Managing XML Schema and Related Database Objects](#)

Introducing Oracle XML DB and Oracle Enterprise Manager

This chapter describes how to use Oracle Enterprise Manager (Enterprise Manager) to administer and manage Oracle XML DB.

Getting Started with Oracle Enterprise Manager and Oracle XML DB

Oracle Enterprise Manager is supplied with Oracle9i database software, both Enterprise and Standard Editions. To run the Enterprise Manager version that supports Oracle XML DB functionality, use Oracle9i Release 2 (9.2) or higher.

Enterprise Manager: Installing Oracle XML DB

Oracle XML DB is installed by default when the Database Configuration Assistant (DBCA) is used to create a new database. The following actions take place during Oracle XML DB installation:

- Oracle registers the configuration XML schema:
`http://www.oracle.com/xdb/xdbconfig.xsd`
- Oracle inserts a default configuration document. It creates resource `/sys/xdbconfig.xml` conforming to the configuration XML schema. This resource contains default values for all Oracle XML DB parameters.

See Also:

- [Chapter 2, "Getting Started with Oracle XML DB"](#)
- [Appendix A, "Installing and Configuring Oracle XML DB"](#)

You Must Register Your XML Schema with Oracle XML DB

Oracle XML DB is typically used for its faster retrieval and search capabilities, access control, and versioning of XML documents. XML instance documents saved in the database can conform to an XML Schema. XML Schema is a schema definition language, also written in XML, that can be used to describe the structure and various other semantics of a conforming XML instance document.

Oracle XML DB provides a mechanism to register XML Schemas with the database.

Assuming that your XML schema document(s) already exists, registering the XML schema is your first task. Before you register the XML schema, you must know the following:

1. *Whether the data for the XML instance documents already exists in relational tables.* This would be the case for legacy applications. If so, Object Views for XML must be created.
2. *What is the storage model?* Are you using LOB storage, object-relational storage, or both? The answer to this question depends on which parts of the document are queried the most often, and hence would need faster retrieval.
3. *Is the XML Schema document annotated with comments to generate object datatypes and object tables?* If not, these objects will have to be created and mapped manually to the database schema.

See Also:

- [Chapter 3, "Using Oracle XML DB"](#)
- [Chapter 5, "Structured Mapping of XMLType"](#)

For most cases, it is assumed that you have XML schema annotated with information to automatically generate object types and object tables. Hence the Oracle9i database, as part of XML schema registration, automatically generates these objects.

See Also: ["Managing XML Schema and Related Database Objects"](#) on page 21-27.

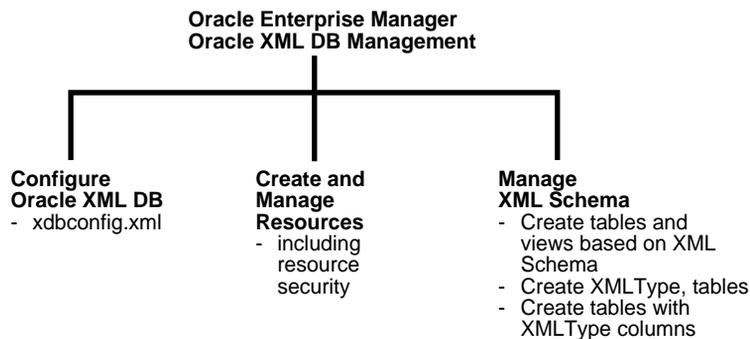
Oracle XML DB is now ready for you to insert conforming XML documents.

Oracle Enterprise Manager Oracle XML DB Features

With Enterprise Manager you can perform the following main Oracle XML DB administrative tasks:

- [Configure Oracle XML DB](#)
- [Create and Manage Resources](#)
- [Manage XML Schema and Related Database Objects](#)

See [Figure 21-1](#) and [Figure 21-2](#).

Figure 21–1 Managing Oracle XML DB with Enterprise Manager: Main Tasks

Configure Oracle XML DB

Oracle XML DB is managed through the configuration file, `xdbconfig.xml`. Through Enterprise Manager, you can view or configure this file's parameters. To access the Oracle XML DB configuration options, from the Enterprise Manager right hand window, select "Configure XML Database". See ["Configuring Oracle XML DB with Enterprise Manager"](#) on page 21-7.

Create and Manage Resources

To access the XML resource management options, select the XML Database object in the Navigator and click "Create a Resource" in the detail view. See ["Creating and Managing Oracle XML DB Resources with Enterprise Manager"](#) on page 21-12.

Manage XML Schema and Related Database Objects

To access the XML schema management options, select the XML Database object in the Navigator and click "Create Table and Views Based on XML Schema" in the detail view. See ["Managing XML Schema and Related Database Objects"](#) on page 21-27.

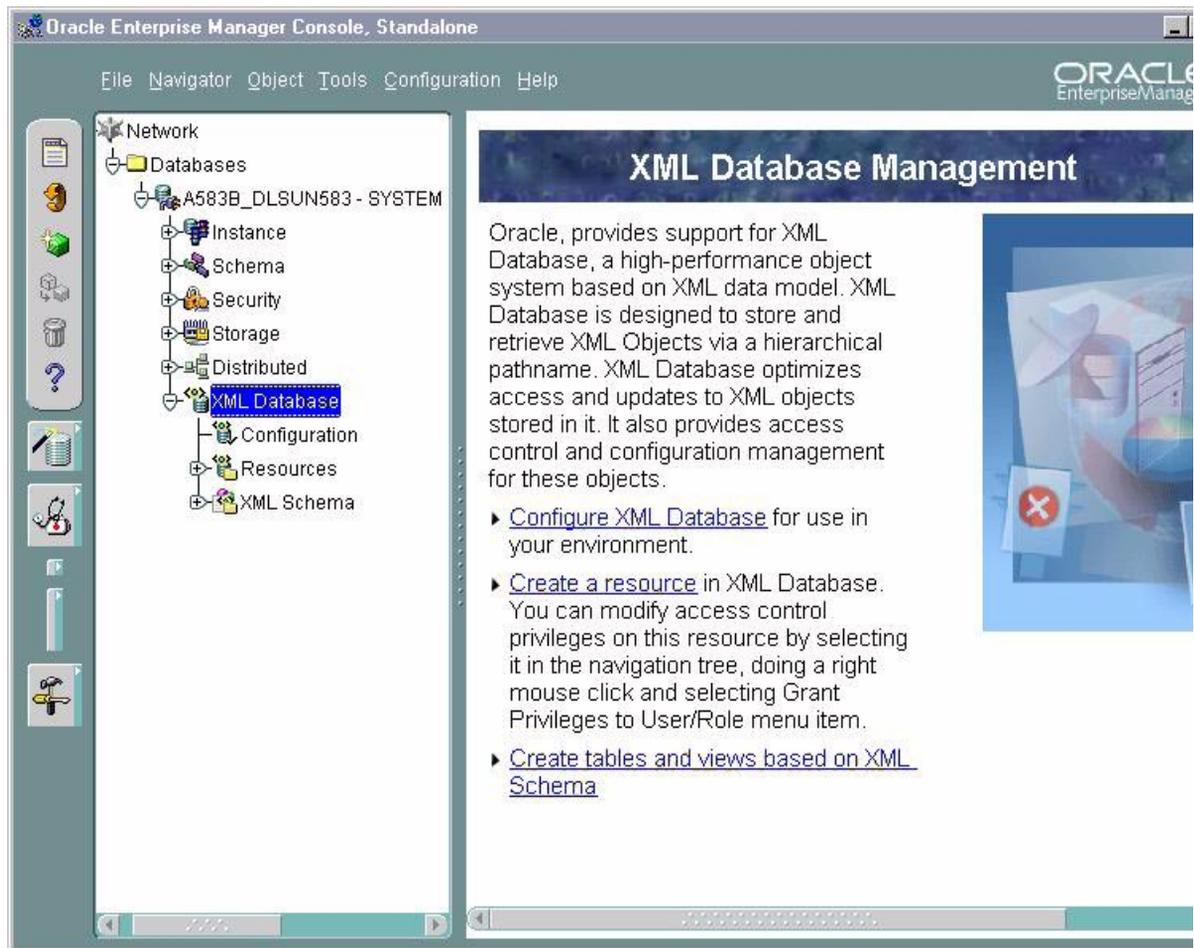
You can register, delete, view, generate (that is, reverse engineer) an XML schema and see its dependencies. You can also view an XML schema's contents and perform actions on its constituent elements.

- **Views.** Create or alter an object view of `XMLType`, and look at the corresponding indexes.

- *Tables.* Create XML schema-based and non-XML schema-based `XMLType` tables and columns, and look at the corresponding indexes. You can also specify LOB storage attributes on these columns.

You can create `XMLType` tables in CLOB or object-relational form, and specify constraints and LOB storage attributes on hidden `XMLType` columns.

- *Indexes.* Create index on the hidden columns of `XMLType`.
- *DML operations.* You can also perform other DML operations such as inserting and updating rows using XML instance documents.
- *XML Schema Elements.* You can view elements in their XML form and the XML instance data stored in the database corresponding to that element. You can also view the XML schema's dependent objects, such as tables, views, indexes, object types, array types, and table types.

Figure 21–2 Enterprise Manager Console: XML Database Management Window

The Enterprise Manager Console for Oracle XML DB

See [Figure 21-2](#). From the Enterprise Manager console you can quickly browse and manage Oracle XML DB objects.

XML Database Management Window: Right-Hand Dialog Windows

From the XML Database Management detail view you can access the XML DB management functionality. From there you can select which task you need to perform.

Hierarchical Navigation Tree: Navigator

Use the left-hand Navigator, to select the Oracle XML DB resources and database objects you need to view or modify.

Configuring Oracle XML DB with Enterprise Manager

Oracle XML DB configuration is an integral part of Oracle XML DB. It is used by protocols such as HTTP/WebDav or FTP, and any other components of Oracle XML DB that can be customized, such as, in the ACL based security.

Oracle XML DB configuration is stored as an XML schema based XML resource, `xdbconfig.xml` in the Oracle XML DB Repository. It conforms to the Oracle XML DB configuration XML schema stored at:

<http://www.oracle.com/xdb/xdbconfig.xsd>

This configuration XML schema is registered when Oracle XML DB is installed. The configuration property sheet has two tabs:

- *For System Configurations.* General parameters and FTP and HTTP protocol specific parameters can be displayed on the System Configurations tab.
- *For User Configurations.* Custom parameters can be displayed on the User Configurations tab.

To configure items in Oracle XML DB, select the **Configuration** node under XML Database in the Navigator. See [Figure 21-3](#) and See [Figure 21-4](#).

The **XML Database Parameters** page displays a list of configuration parameters for the current XML database. You can also access this page from the XML Database Management main window > Configure XML Database.

When you click the **Configuration** node for the XML database in the Enterprise Manager Navigator, the **XML DB Parameters** page appears in the main panel to the right. The XML DB Parameters window lists the following information:

- **Parameter Name** -- Lists the name of the parameter.
- **Value** -- Displays the current value of the parameter. This is an editable field.
- **Default** -- Indicates whether the value is a default value. A check mark indicates a default value.
- **Dynamic** -- Indicates whether or not the value is dynamic. A check mark indicates dynamic.
- **Category** -- Displays the category of the parameter. Category can be HTTP, FTP, or Generic.

You can change the value of a parameter by clicking the **Value** field for the parameter and making the change. Click on the **Apply** button to apply any changes you make. You can access a description of a parameter by clicking on the parameter in the list and then clicking the **Description** button at the bottom of the page. A text **Description** text box displays that describes in greater detail the parameter you selected. You can close the Description box by clicking again on the **Description** button.

Figure 21–3 Enterprise Manager Console: Configuring Oracle XML DB

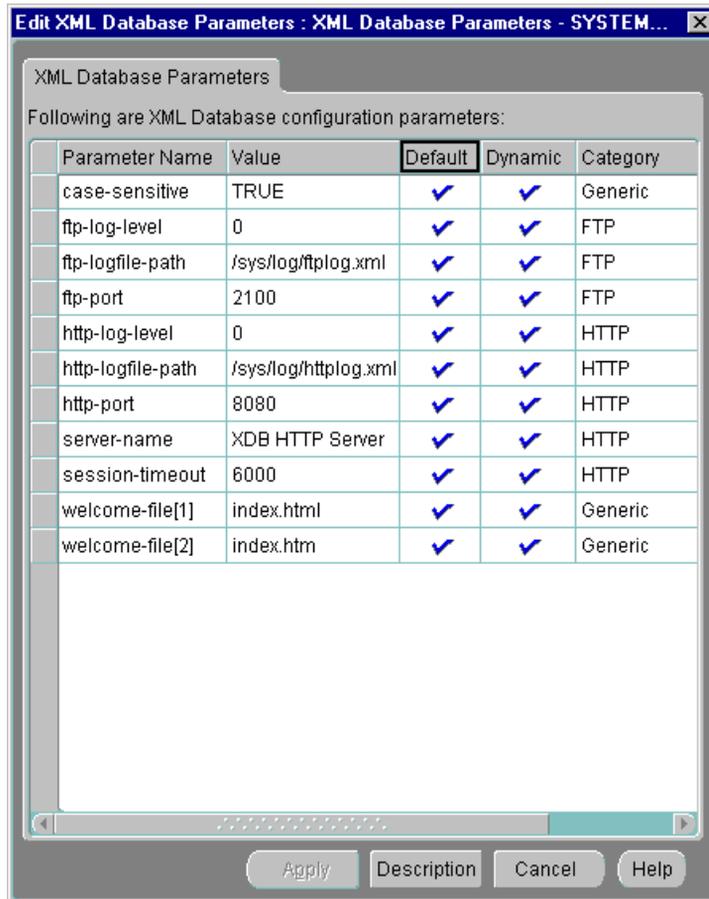
The screenshot shows the Oracle Enterprise Manager console interface. The left pane displays a tree view under 'Network' > 'Databases'. The instance 'XDB2_DLSUN1727 - SYSTEM' is selected, and the 'XML Database' > 'Configuration' folder is highlighted. The right pane, titled 'XML Database Parameters', shows the following configuration parameters:

Parameter Name	Value
case-sensitive	TRUE
ftp-log-level	0
ftp-logfile-path	/sys/log/ftplog.xml
ftp-port	2100
http-log-level	0
http-logfile-path	/sys/log/httplog.xml
http-port	8080
server-name	XDB HTTP Server
session-timeout	6000
welcome-file[1]	index.html
welcome-file[2]	index.htm

Since Oracle XML DB provides support for standard Internet protocols (FTP and WebDAV/HTTP) as a means of accessing the Repository, Enterprise Manager provides you with related information:

- *Oracle XML DB FTP Port*: displays the port number the FTP protocol will be listening to. FTP by default listens on a non-standard, non-protected port.
- *Oracle XML DB HTTP Port*: displays the port number the HTTP protocol will be listening to. HTTP will be managed as a Shared Server presentation, and can be configured through the TNS listener to listen on arbitrary ports. HTTP listens on a non-standard, non-protected port.

Figure 21–4 Enterprise Manager Console: Oracle XML DB Configuration Parameters Dialog



Viewing or Editing Oracle XML DB Configuration Parameters

With Enterprise Manager you can view and edit partial Oracle XML DB configuration parameters, in the following categories:

Category: Generic

- case-sensitive

Note: Oracle XML DB always preserves case.

Category: FTP

- ftp-port: Enterprise Manager manages FTP as a Shared Server presentation. It can be configured using TNS listeners to listen on arbitrary ports.
- ftp-logfile-path: The file path of the FTP Server log file.
- ftp-log-level: The level of logging for FTP error and warning conditions.

Category: HTTP

- http-port: Enterprise Manager manages HTTP as a Shared Server presentation. It can be configured using TNS listeners to listen on arbitrary ports.
- session-timeout: The maximum time the server will wait for client responses before it breaks a connection.
- server-name: Hostname to use by default in HTTP redirects and servlet API.
- http-logfile-path: The file path of the HTTP server log file.
- http-log-level: The level of logging for HTTP error and warning conditions.
- welcome-file-list: The list of welcome files used by the server.

Creating and Managing Oracle XML DB Resources with Enterprise Manager

The **Resources** folder in the Enterprise Manager navigation tree is under the XML Database folder. It contains all the resources in the database regardless of the owner. [Figure 21-5](#) shows a typical Resources tree.

When the Resources folder is selected in the Navigator, the right-hand side of the screen displays all the top level Oracle XML DB resources under root, their names, and their creation and modification dates. See [Figure 21-6](#).

Figure 21–5 Enterprise Manager: Oracle XML DB Resources Tree Showing Resources Folder Selected

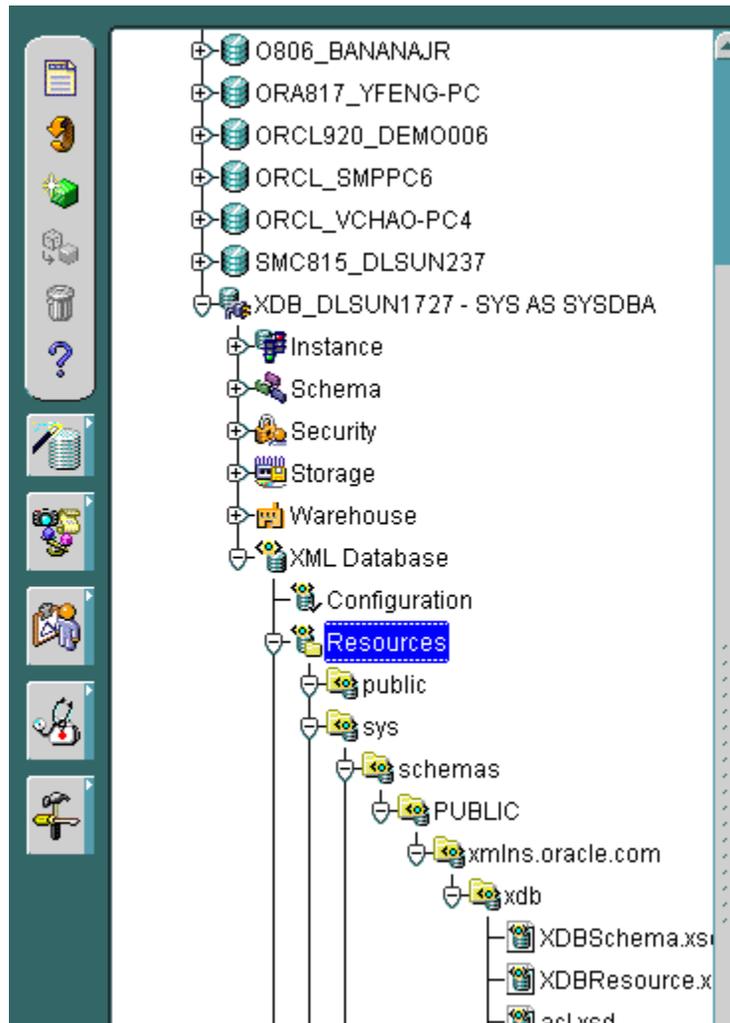
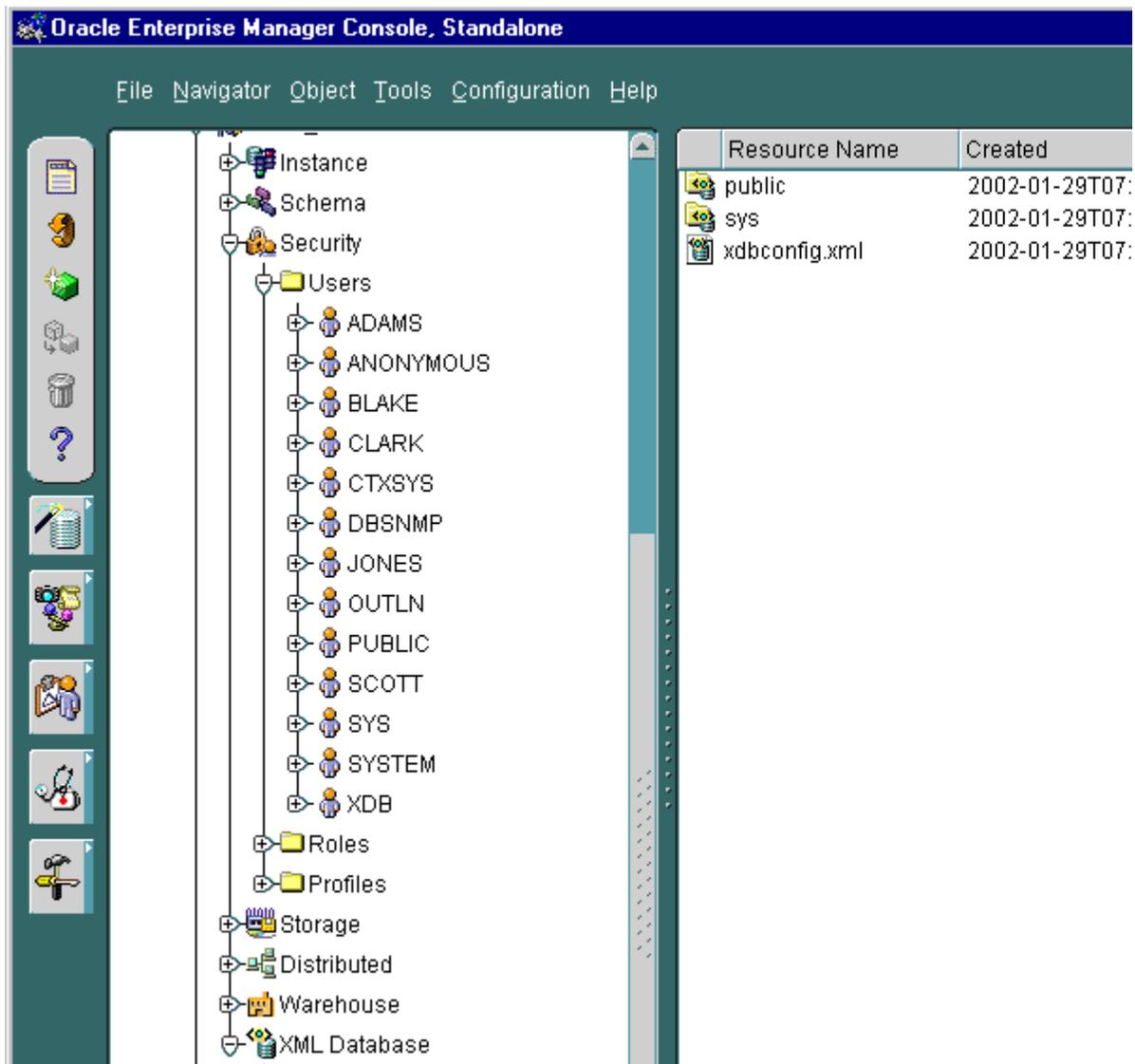


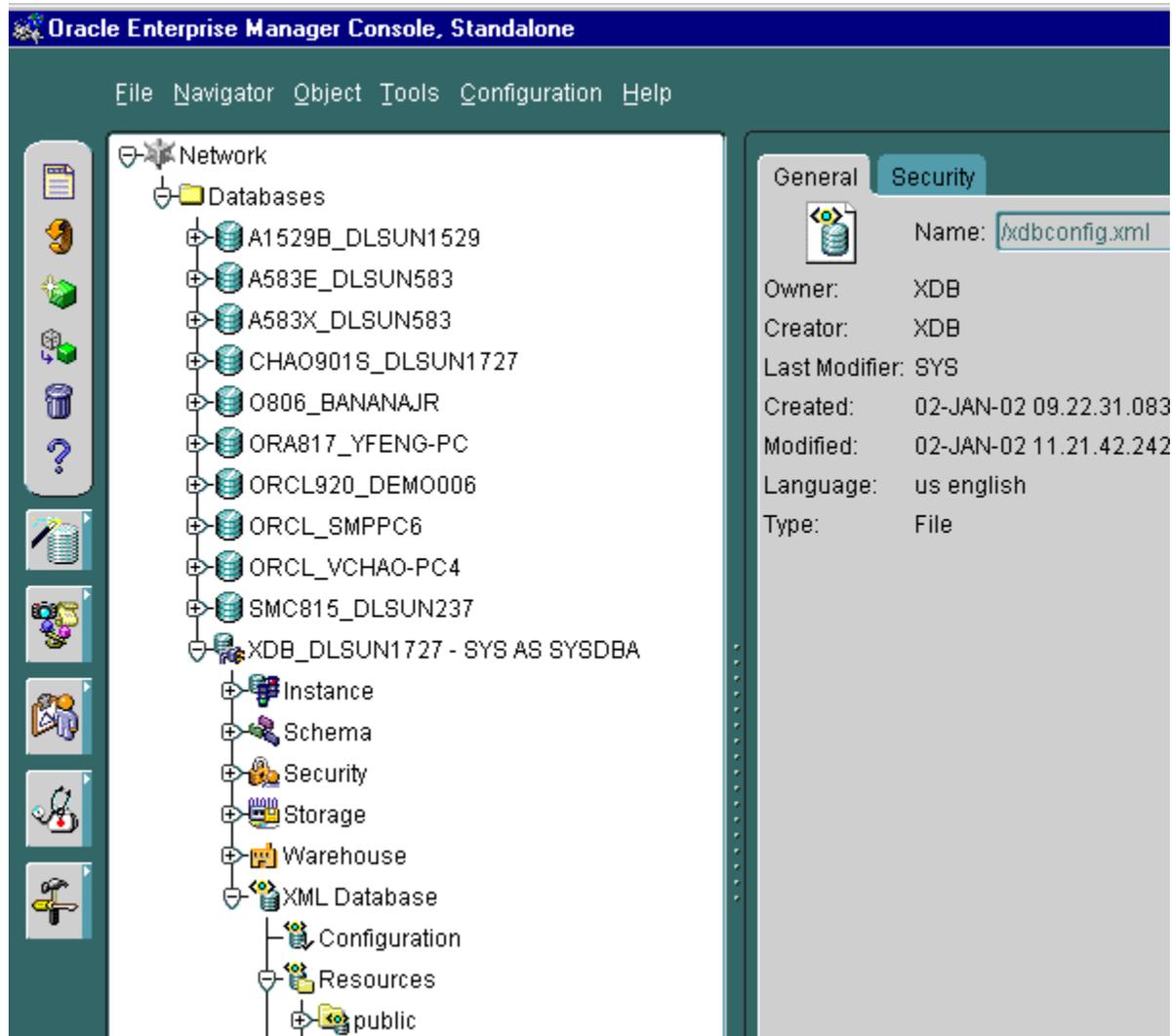
Figure 21–6 Enterprise Manager: Top Level Resources Under Root



Administering Individual Resources

Once you select a resource sub-folder under the main Resources folder in the Navigator, you can see the resources “General Page” in the detail view, as shown in [Figure 21–7](#).

Figure 21–7 Enterprise Manager: Individual Resources - General Page



General Resources Page

The Oracle XML DB Resources General page (also called the XML Resources Page) displays overview information about the resource container or resource file. When you select one of the Oracle XML DB resource containers or files in the Navigator, Enterprise Manager displays the Oracle XML DB Resources General page. This is a read-only page. It displays the following information:

- Name - name of the resource file or container
- Creator - the user or role that created the resource
- Last Modifier - the name of the user who last modified the resource
- Created - the date and time the resource was created
- Modified - the date and time that the resource was last changed
- Language - the resource language, such as, US English
- Type - File or Container

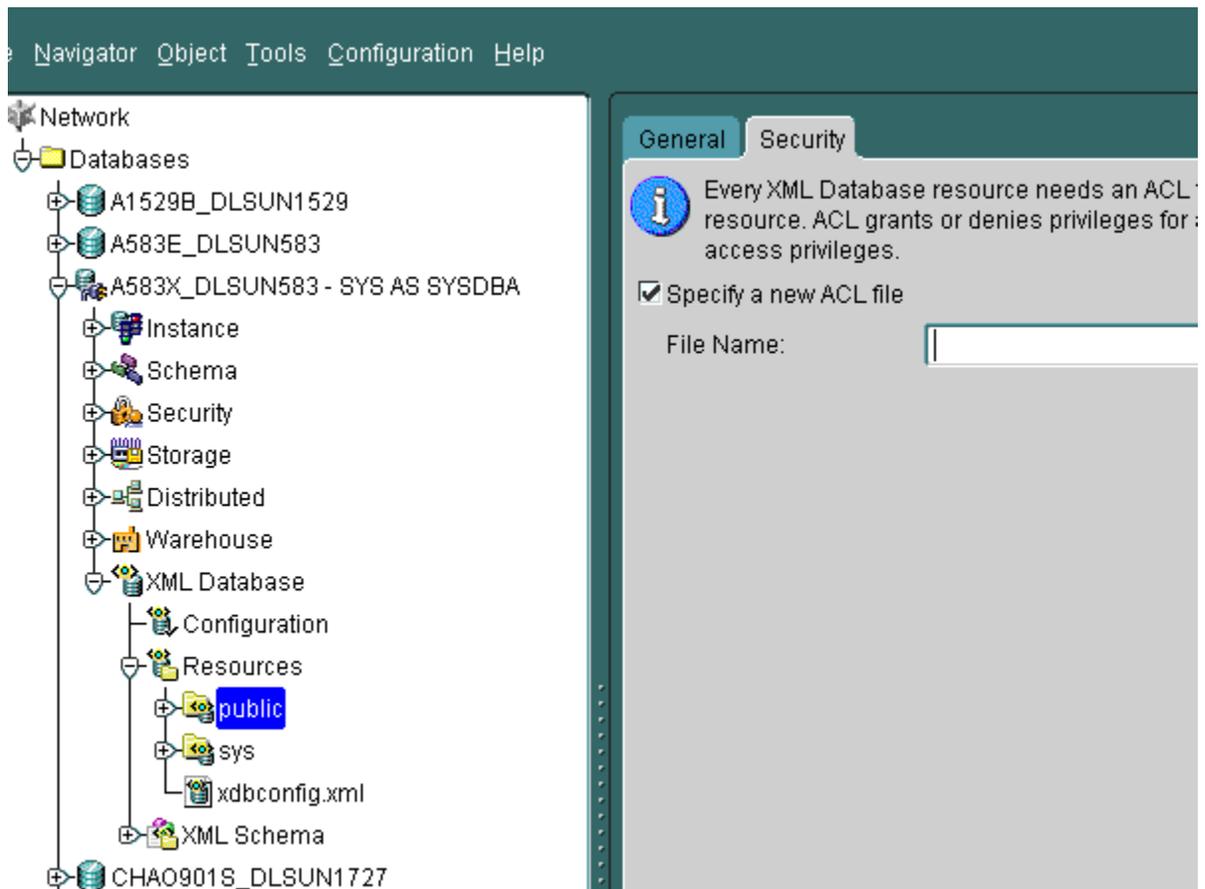
Security Page

Oracle XML DB Resources Security page changes the ACL associated with the XML DB resource container or file. Use the ACL files to restrict access to all XML DB resources. When you change the ACL file for the resource, you change the access privileges for the resource file. See [Figure 21-8](#).

To specify a new ACL file:

1. Click on the **Specify a new ACL File** option box and then choose a new ACL file from the drop down list in the File Name field.
2. Click the **Apply** button to apply the change.
3. Click the **Revert** button to abandon any changes you have made to the File Name field.

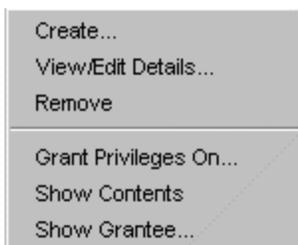
Figure 21–8 Enterprise Manager: Individual Oracle XML DB Resource - Security Page



Individual Resource Content Menu

Figure 21–9 shows the context-sensitive menu displayed when you select and right-click an individual Oracle XML DB resource object in the Navigator.

Figure 21–9 Enterprise Manager: When You Right-Click on an Individual Resource...

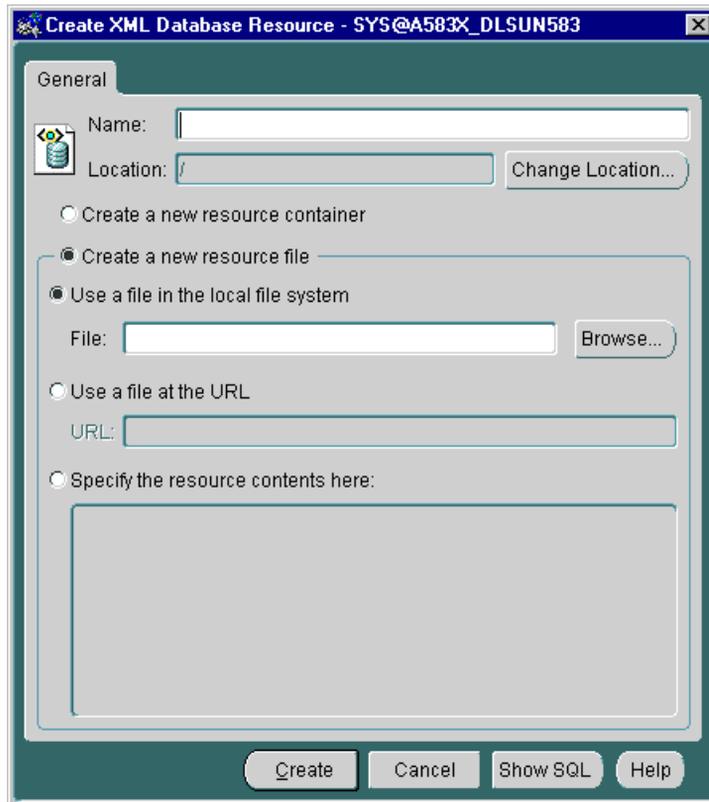


You can perform the following Oracle XML DB tasks from the Enterprise Manager Content Menu:

Create Resource

[Figure 21–10](#) shows how you can use the **Create XML DB Resource** dialog box to create an XML DB resource container or file. From the XML DB Resource dialog you can name the resource and then either create a new resource container or create a new resource file, designating the location of the file as either a local file, a file at a specified URL, or specifying the contents of the file.

1. Access the Create XML DB Resource dialog box by right clicking on the Resources folder or any individual resource node and selecting **Create** from the context menu. When you name the resource in the Name field, you can change the location by clicking on the **Change Location** button to the right of the Location field.
2. Specify whether the resource you are creating is a container or a file. If you create a file by choosing **Create a new resource file**, you can select from one of three file type location options:
 - Local File -- Select **Use a file in the local file system** to browse for a file location on your network.
 - File at URL -- Select **Use a file at the URL** to enter the location of the file on the internet or intranet.
 - File Contents -- Select **Specify the resource contents here** to enter the contents of a file in the edit box located at the bottom of the Create XML DB Resource dialog box.

Figure 21–10 Enterprise Manager: Create Oracle XML DB Resource Dialog Box

Grant Privileges On...

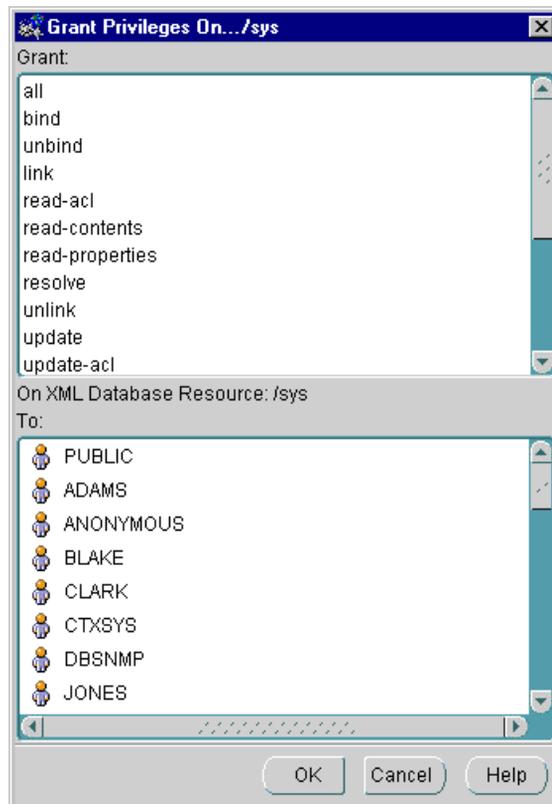
Figure 21–11 shows the **Grant Privileges On** dialog box which assigns privileges on an Oracle XML DB resource to users or roles. You can grant multiple privileges to multiple users or roles. Grant Privileges On dialog box lists the available Oracle XML DB resource privileges in the Grant section at the top of the panel.

1. To grant privileges, select the privileges you want to grant by clicking on a privilege. You can select multiple privileges by holding down the Ctrl key while selecting more than one privilege. You can select consecutive privileges by clicking the first privilege in a sequence and holding down the Shift key while clicking the last privilege in the sequence.

2. Select the user/ group in the To: box at the bottom of the dialog page. Use the same procedure to select multiple users/roles to which to grant privileges.

See Also: ["Enterprise Manager and Oracle XML DB: ACL Security"](#) on page 21-22.

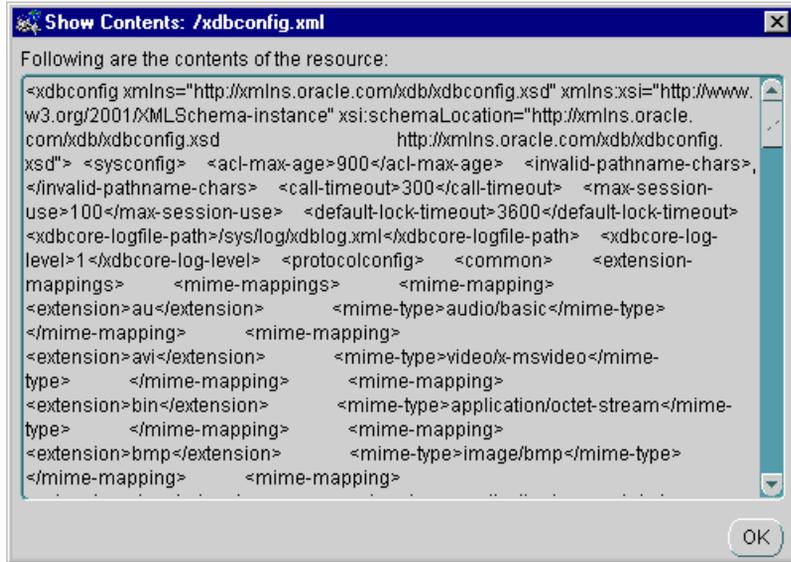
Figure 21–11 Enterprise Manager: Granting Privileges On Oracle XML DB Resources



Show Contents

Figure 21–12 is an example of a **Show Contents** dialog box. It displays the contents of the selected resource file.

Figure 21–12 Enterprise Manager: Show Contents Menus of Individual Oracle XML DB Resource



Show Grantee

Figure 21–11 shows the **Show Grantee of XML DB Resource** dialog box. It displays a list of all granted privileges on a specified XML DB resource for the connected Enterprise Manager user. Show Grantee dialog box lists the connected Enterprise Manager user, privilege, and granted status of the privilege in tabular format.

Figure 21–13 Enterprise Manager: Show Grantee of Oracle XML DB Resources

Following are the resource privileges granted to SYS on: /public

User	Privilege	Granted
SYS	read-properties	✓
SYS	read-contents	✓
SYS	update	✓
SYS	bind	✓
SYS	unbind	✓
SYS	read-acl	✓
SYS	write-acl-ref	✓
SYS	update-acl	✓
SYS	resolve	✓
SYS	link	✓
SYS	unlink	✓

Enterprise Manager and Oracle XML DB: ACL Security

From Enterprise Manager you can restrict access to all XML DB resources by means of ACLs. You can grant XML DB resource privileges to database user and database role separately using the existing `Security/Users/user` and `Security/Roles/role` interface, respectively.

You can access the Enterprise Manager security options in two main ways:

- To view or modify user (or role) security:** In the Navigator, under the Oracle XML DB database in question > Security >Users >user (or > Security > Roles > role). In the detail view, select the XML tag. See [Figure 21–14](#). This user security option is described in more detail in "[Granting and Revoking User Privileges with User > XML Tab](#)" on page 21-23.
- To view or modify a resource's security:** Select the individual resource node under the "Resources" folder in the left navigation panel. Select the Security tag in the detail view. See [Figure 21–15](#).

Granting and Revoking User Privileges with User > XML Tab

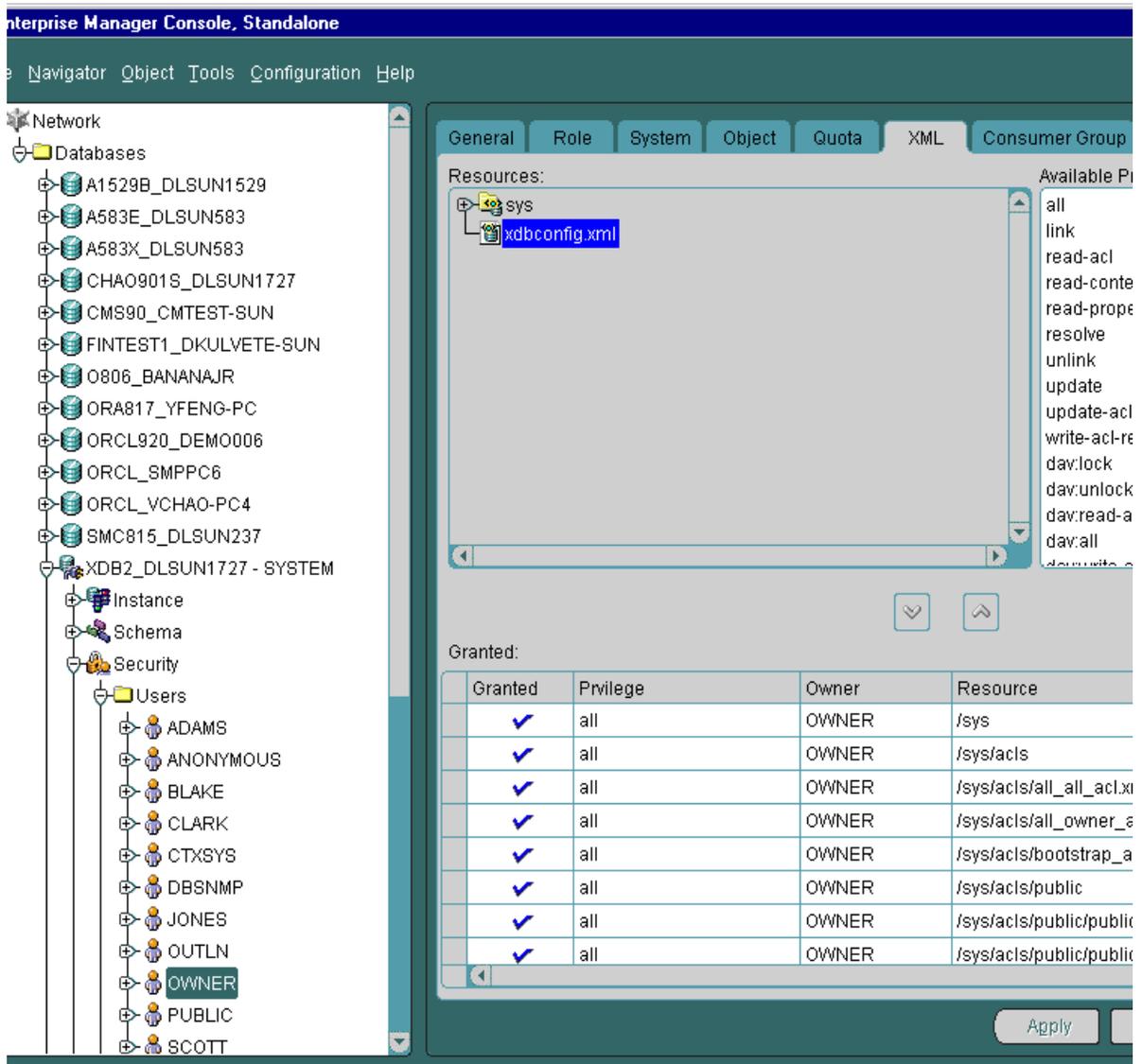
This section describes how to grant and revoke privileges for a “user”, The same procedure applies when granting and revoking privileges for a “role”. To grant privileges to a user follow these steps:

1. Select a particular user from the Enterprise Manager Navigator. The detail view displays an additional tab, **XML**, in the existing property sheet.
2. To view and select resources on which you want to grant privileges to users or roles, select the **XML** tab. Once you select a resource, all available privileges for that resource are displayed in the **Available Privileges** list to the right of the Resources list.
3. Select the privileges required from the **Available Privileges** list and move them down to the **Granted** list that appears at the bottom of the window by clicking on the down arrow.

Conversely, you can revoke privileges by selecting them in the **Granted** list and clicking on the up arrow.

4. After setting the appropriate privileges, click the **Apply** button to save your changes. Before you save your changes, you can click the **Revert** button to abandon your changes.

Figure 21–14 Adding or Revoking Privileges with Users > user > XML



Resources List

The **Resources** list is a tree listing of resources located in Oracle XML DB Repository. You can navigate through the folder hierarchy to locate the resource on which you want to set privileges for the user or role you selected in Navigator. When you select a resource in the tree listing, its privileges appear in the **Available Privileges** list to the right.

Available Privileges List

The **Available Privileges** list displays all privileges available for a resource. Click on a privilege and then press the down arrow button to add the privilege to the **Granted** list. You can select consecutive privileges by clicking on the first privilege and then holding the Shift key down to select the last in the list. Also, you can select non-consecutive privileges by holding the Ctrl-key down while making selections.

Privileges can be either of the following:

- aggregate privileges. They contain other privileges.
- atomic privileges. They cannot be subdivided.

Granted List

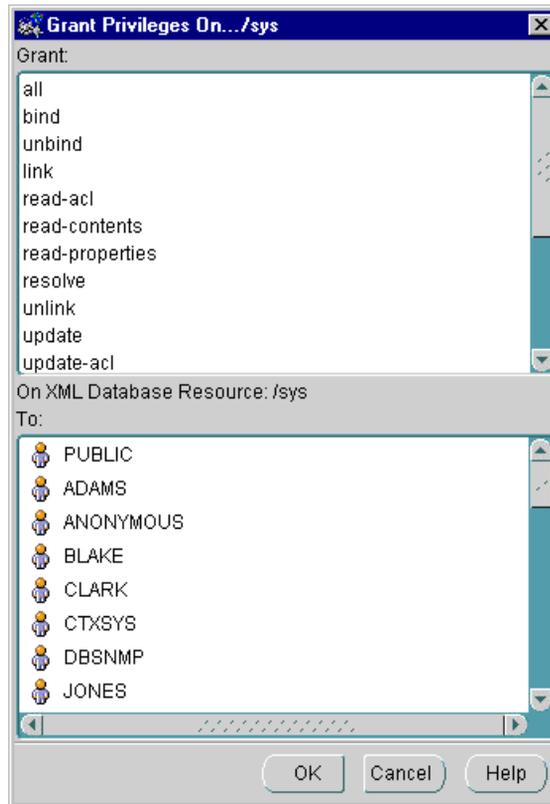
The **Granted** list displays all privileges granted to the user or role for the resource selected in the **Resources** list. You can revoke a privilege by highlighting it and clicking on the up arrow to remove it.

XML Database Resource Privileges

Privilege can be aggregate (contain other privileges) or atomic (cannot be subdivided). The following system privileges are supported:

- **Atomic Privileges**
 - all
 - read-properties
 - read-contents
 - update
 - link (applies only to containers)
 - unlink (applies only to containers)
 - read-acl

- write-acl-Ref
- update-acl
- linkto
- unlinkfrom
- dav:lock
- dav:unlock
- dav:read-acl
- **Aggregate Privileges**
 - dav:read (read-properties, read-contents)
 - dav:write-acl
 - dav:all (dav:read, dav:write, dav:read-acl, dav:write-acl, dav:lock, dav:unlock)

Figure 21–15 Granting Privileges on a Resource

See Also: [Chapter 18, "Oracle XML DB Resource Security"](#) for a list of supported system privileges.

Managing XML Schema and Related Database Objects

From the XML Database Management detail view, when you select “Create tables and views based on XML Schema” the “XML Schema Based Objects” page appears. With this you can:

- Register an XML schema
- Create a structured storage based on XML schema

- Create an XMLType table
- Create a table with XMLType columns
- Create a view based on XML schema
- Create a function-based index based on XPath expressions

Navigating XML Schema in Enterprise Manager

Under the **XML Schema** folder, the tree lists all the XML schema owners. The example here is owner “XDB”. See [Figure 21–16](#):

- *Schema Owners.* Under the individual XML schema owners, the tree lists the XML schemas owned by the owner(s). Here you can see:
 - `http://xmlns.oracle.com/xdm/XDBResource.xsd`
 - `http://xmlns.oracle.com/xdm/XDBSchema.xsd`
 - `http://xmlns.oracle.com/xdm/XDBStandard.xsd`
- *Top level elements.* Under each XML schema, the tree lists the top level elements. These elements are used to create the XMLType tables, tables with XMLType columns, and views. For example, [Figure 21–16](#) shows top level elements **servlet** and **LINK**. The number and names of these elements are dictated by the relevant XML schema definition, which in this case is:
 - `http://xmlns.oracle.com/XDBStandard.xsd`.
- *Dependent objects.* Under each element, the tree lists the created dependent objects, Tables, Views, and User Types respectively. In this example, you can see that top level element **servlet** has dependent XMLType **Tables**, **Views**, and **User types**.
 - **Tables.** For example, under **Tables**, “XDB” is an owner, and XDB owns a table called **SERVLET**.
 - * Table characteristics. Under each table name the tree lists any created **Indexes**, **Materialized View Logs (Snapshots)**, **Partitions**, and **Triggers**.
 - **Views.** Not shown here but under Views you would see any view owners and the name of views owned by these owners:
 - * View characteristics. These are not listed here.

- **User Types.** The tree lists any user types associated with the top level element servlet. These are listed according to the type:
 - * Object types. Under Object types the tree lists the object type owners.
 - * Array types. Under Array types the tree lists the array type owners.
 - * Table types. Under table types the tree lists the table type owners.

Figure 21–16 Enterprise Manager Console: Navigating XML Schema

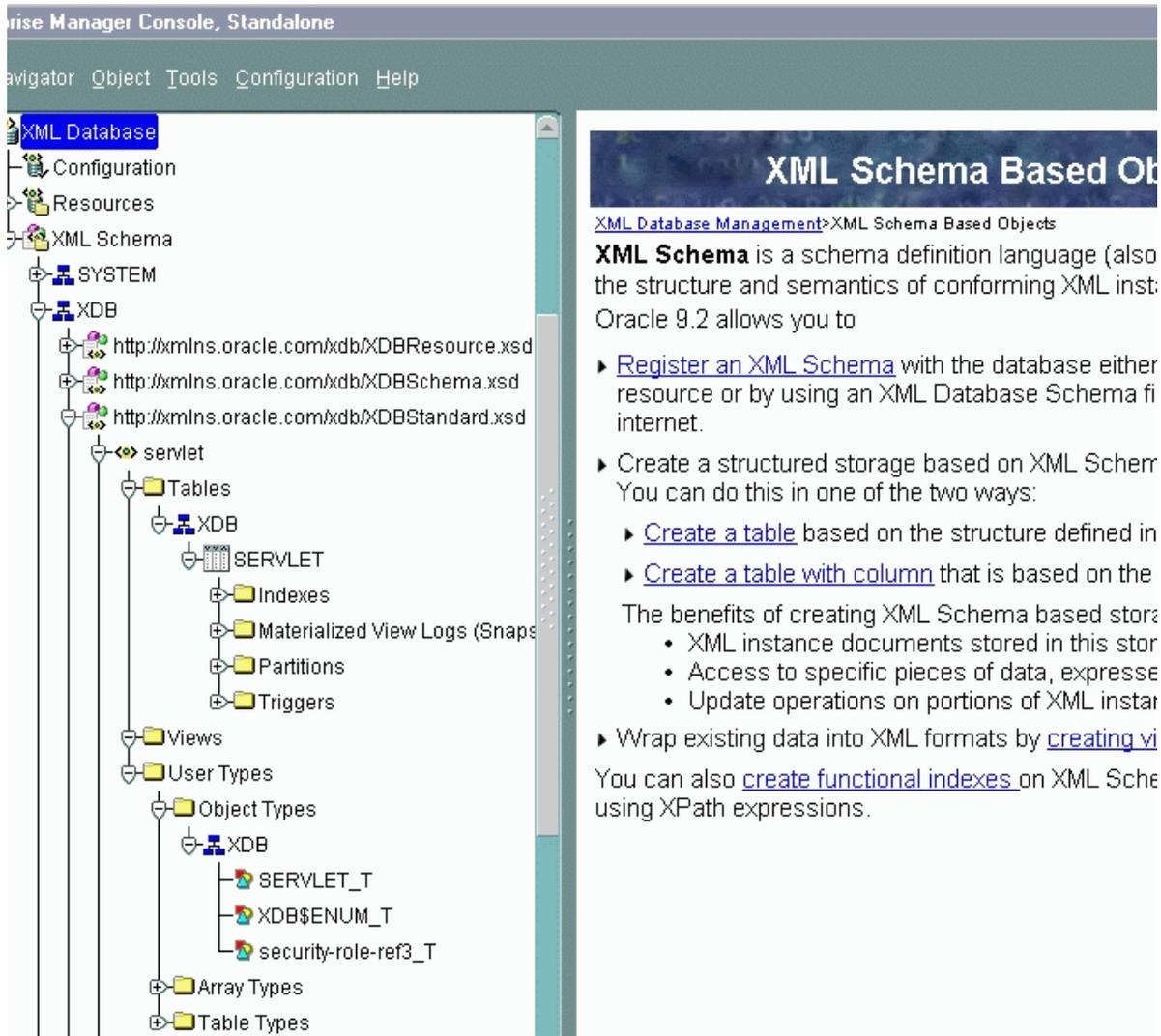
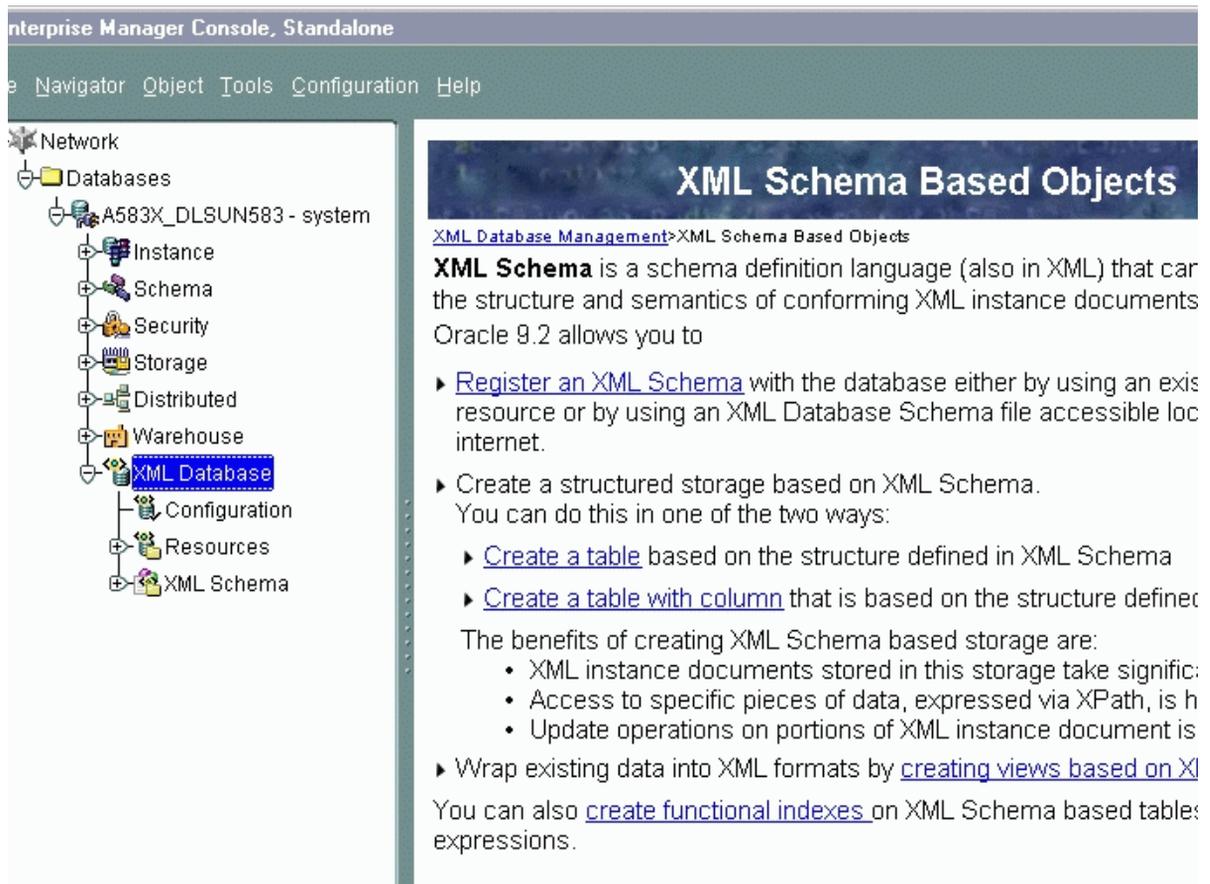


Figure 21–17 Enterprise Manager: Creating XML Schema-Based Objects

Registering an XML Schema

Registering an XML schema is one of the central, and often first, tasks before the you use Oracle XML DB. XML schemas are registered using `DBMS_XMLSCHEMA.registerSchema()`.

See Also: [Chapter 5, "Structured Mapping of XMLType"](#)

[Figure 21–18](#) shows you how to register an XML schema.

General Page

From the GENERAL page, input the XML schema URL and select the Owner of the XML schema from the drop-down list.

Select either:

- **Global**, that is XML schema is visible to public
- **Local**, that is XML schema is visible only to the user creating it

You can obtain the XML schema in one of four ways:

- By specifying the location of the file in the local file system
- By specifying the XML Database (Repository) resource where the XML schema is located
- By specifying the URL location
- By cutting and pasting the text from another screen

Options Page

From the Options page you can select the following options:

- Generate the object types based on this XML schema
- Generate tables based on this XML schema
- Generate Java beans based on this XML schema
- Register this XML schema regardless of any errors

See [Figure 21-19](#). Press Create from either the General Tab or Options Tab to register this XML schema.

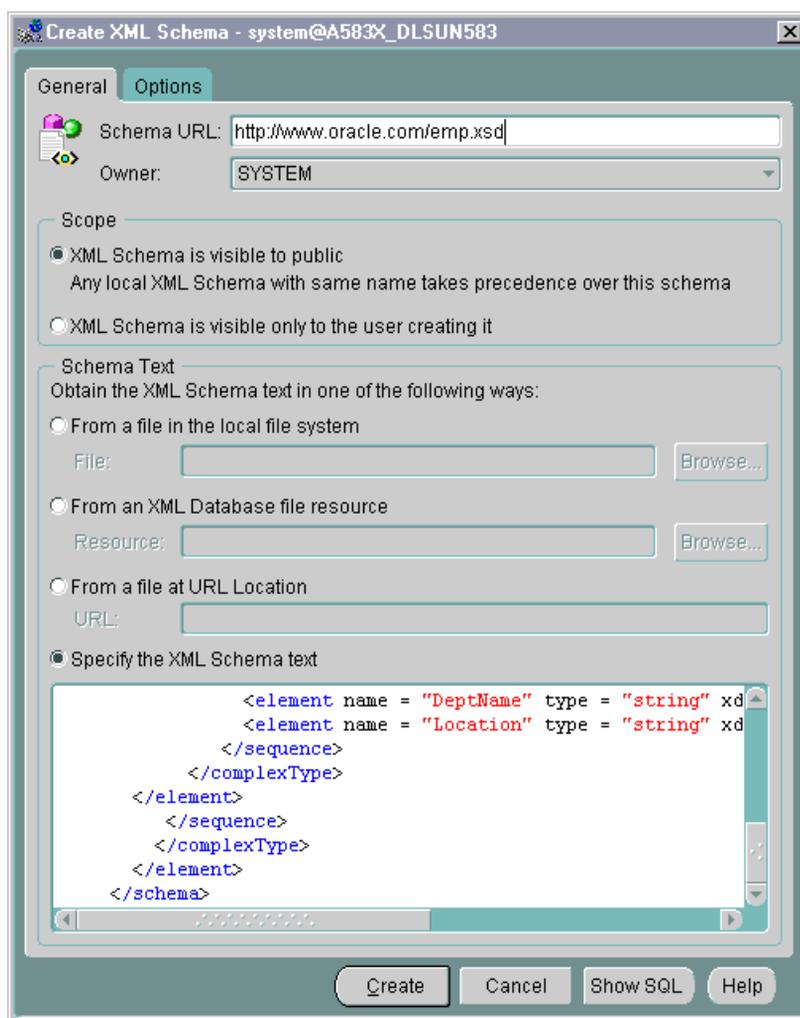
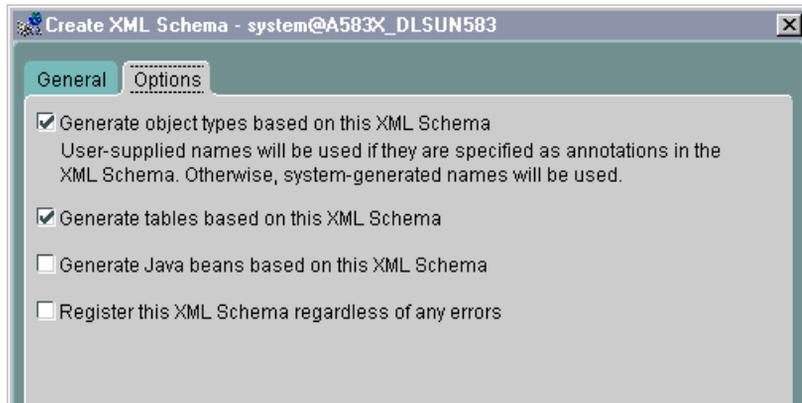
Figure 21–18 Enterprise Manager: Registering an XML Schema - General Page

Figure 21–19 Enterprise Manager: Creating XML Schema - Options Page



Creating Structured Storage Infrastructure Based on XML Schema

This section describes how to use Oracle Enterprise Manager to create `XMLType` tables, views, and indexes.

Creating Tables

You have two main options when creating tables:

- ["Creating an XMLType Table"](#) on page 21-35
- ["Creating Tables with XMLType Columns"](#) on page 21-37

Creating Views

To create an `XMLType` view, see ["Creating a View Based on XML Schema"](#) on page 21-39.

Creating Function-Based Indexes

To create function-based indexes see ["Creating a Function-Based Index Based on XPath Expressions"](#) on page 21-42.

Creating an XMLType Table

From the Create Table property sheet, enter the desired name of the table you are creating on the General page. Select the table owner from the drop-down list “Schema”. Leave Tablespace at the default setting. Select “XMLType” table.

Under the lower “Schema” option, select the XML schema owner, from the drop-down list.

Under “XML Schema”, select the actual XML schema from the drop-down list.

Under “Element”, select the required element to from the XMLType table, from the drop-down list.

Specify the storage:

- *Store as defined by the XML schema.* When you select this option, hidden columns are created based on the XML schema definition annotations. Any SELECTs or UPDATEs on these columns are optimized.
- *Store as CLOB.* When you select CLOB the LOB Storage tab dialog appears. Here you can customize the CLOB storage. See [Figure 21-21](#).

Figure 21–20 Enterprise Manager: Creating XMLType Tables-General Page

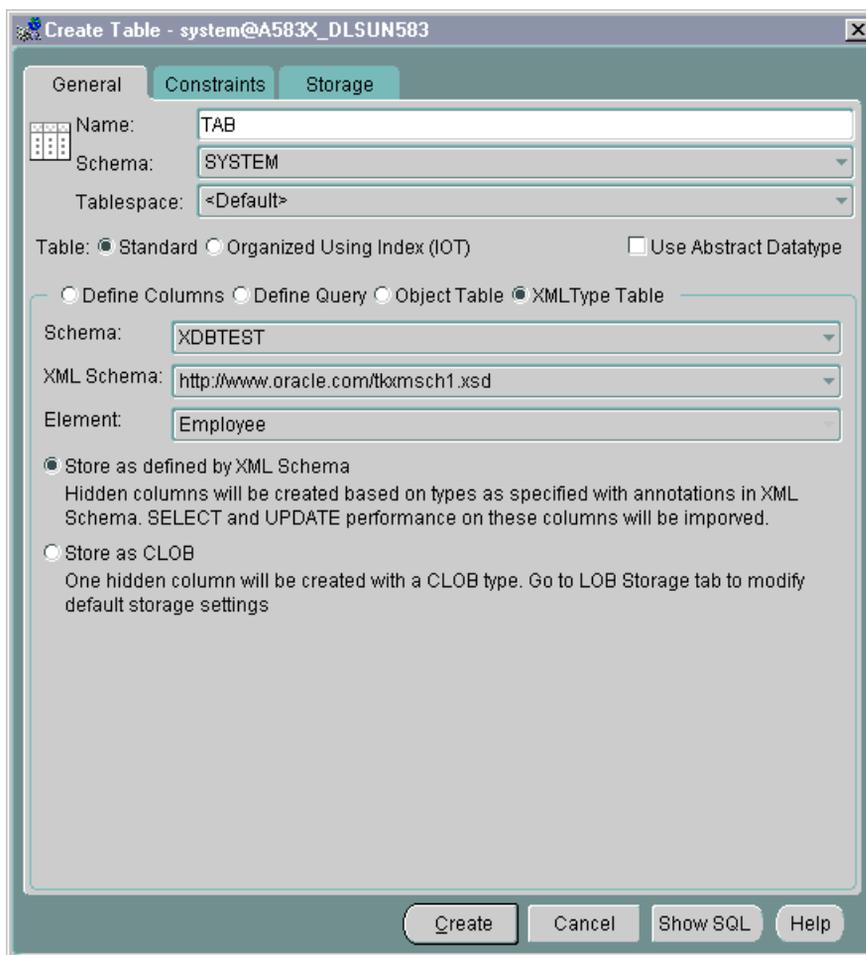
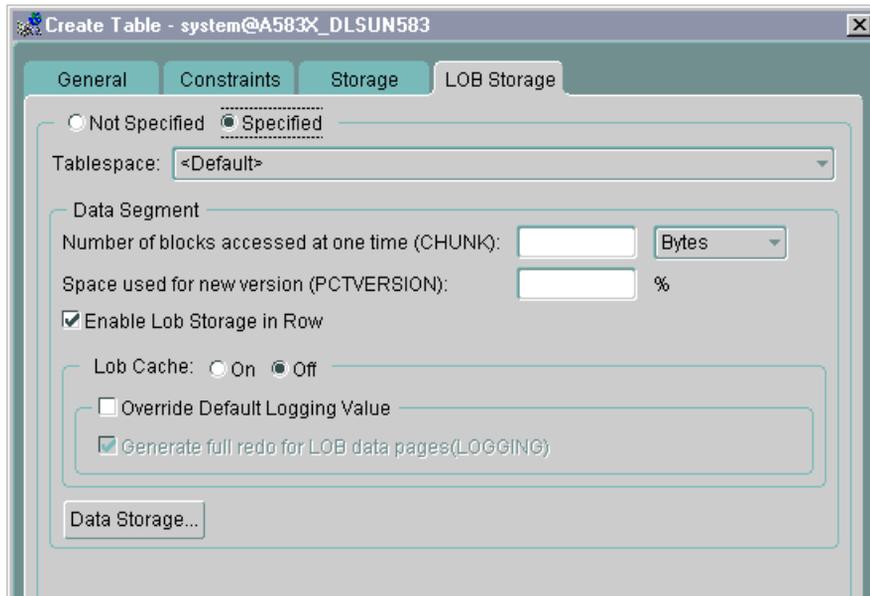


Figure 21–21 Enterprise Manager: Creating XMLType Tables - Specifying LOB Storage

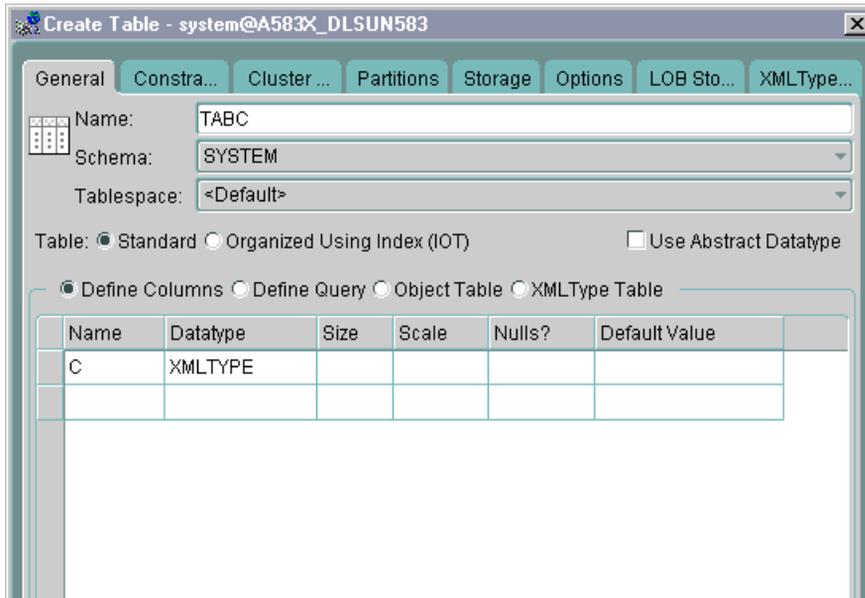


Creating Tables with XMLType Columns

See [Figure 21–22](#) shows the Create Table General page. To create a table with XMLType columns follow these steps:

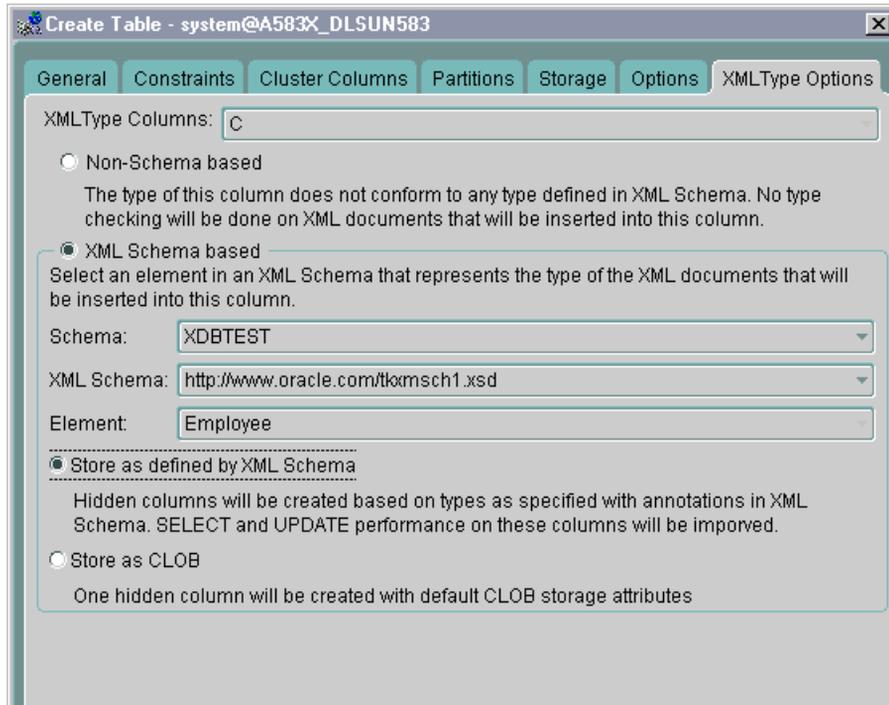
1. From the Create Table property sheet, enter the desired name of the table you are creating on the General page.
2. Select the table owner from the drop-down list “Schema”. Leave Tablespace at the default setting.
3. Select “Define Columns”.
4. Enter the Name. Enter the Datatype; select XMLType from the drop-down list. The XMLType Options dialog window appears. See [Figure 21–23](#).

Figure 21–22 Enterprise Manager: Creating Tables With XMLType Column - General Page



5. From this screen you can specify for a particular XMLType column, whether it is XML schema-based or non-schema-based.
 - If it is XML schema-based:
 - * Under the “Schema” option, select the XML schema owner, from the drop-down list.
 - * Under “XML Schema”, select the actual XML schema from the drop-down list.
 - * Under “Element”, select the required element to form the XMLType column, from the drop-down list.
 - * Specify the storage:
 - * Store as defined by the XML schema.
 - * Store as CLOB. When you select CLOB the LOB Storage tab dialog appears. Here you can customize the CLOB storage.
 - If it is non-schema-based you do not need to change the default settings.

Figure 21–23 Enterprise Manager: Creating Tables With XMLType Column - XMLType Options



Creating a View Based on XML Schema

Figure 21–24 shows the Create View General page. To create a view based on an XML schema, follow these steps:

1. Enter the desired view name. Under “Schema”, select the view owner from the drop-down list.
2. Enter the SQL statement text in the “Query Text” window to create the view. Select the Advanced tab. See Figure 21–25.

From here you can select Force mode

3. Select the “As Object” option. The view can be set to Read Only or With Check Option.

4. Because this is an `XMLType` view, select the “As Object” option. Select “`XMLType`” not “Object Type”.
5. Under the “Schema” option, select the XML schema owner, from the drop-down list.
6. Under “XML Schema”, select the actual XML schema from the drop-down list.
7. Under “Element”, select the required element to form the `XMLType` column, from the drop-down list.
8. Specify the Object Identifier (OID):
 - If your SQL statement to create the view is based on an object type table, then select the “Use default if your query is based on...”
 - Otherwise, select “Specify based on XPath expression on the structure of the element”. Enter your XPath expression. See [Chapter 11, "XMLType Views"](#).

Figure 21–24 Enterprise Manager: Creating an XMLType View - General Page

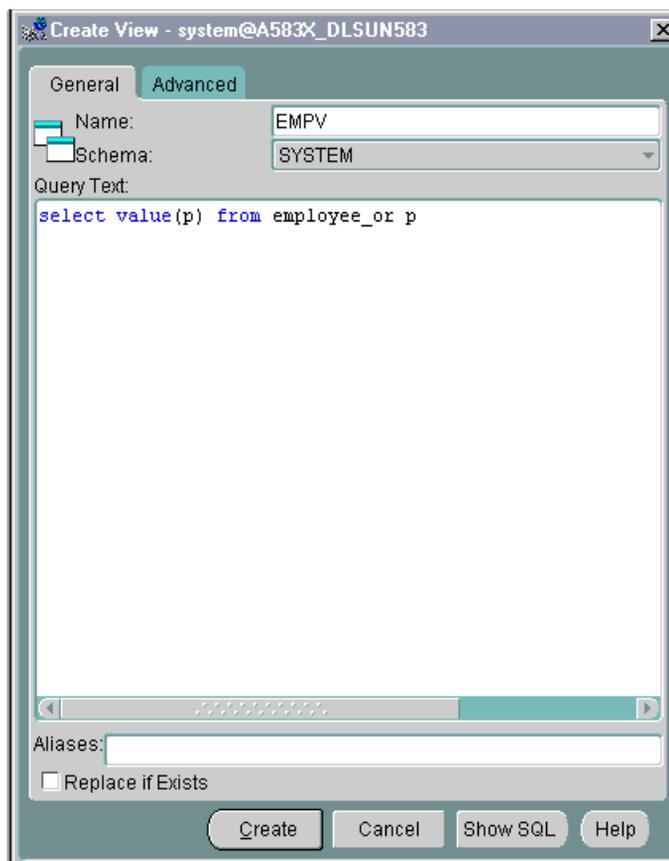
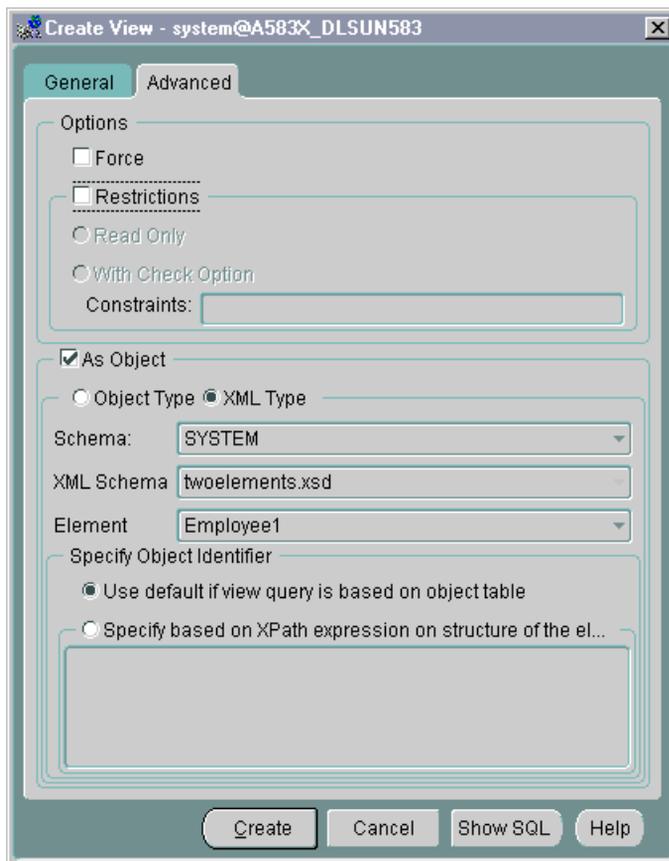


Figure 21–25 Enterprise Manager: Creating XMLType Views - Advanced Page



Creating a Function-Based Index Based on XPath Expressions

See [Figure 21–26](#) shows the Create Index General page. To create a function-based index based on an XPath expression follow these steps:

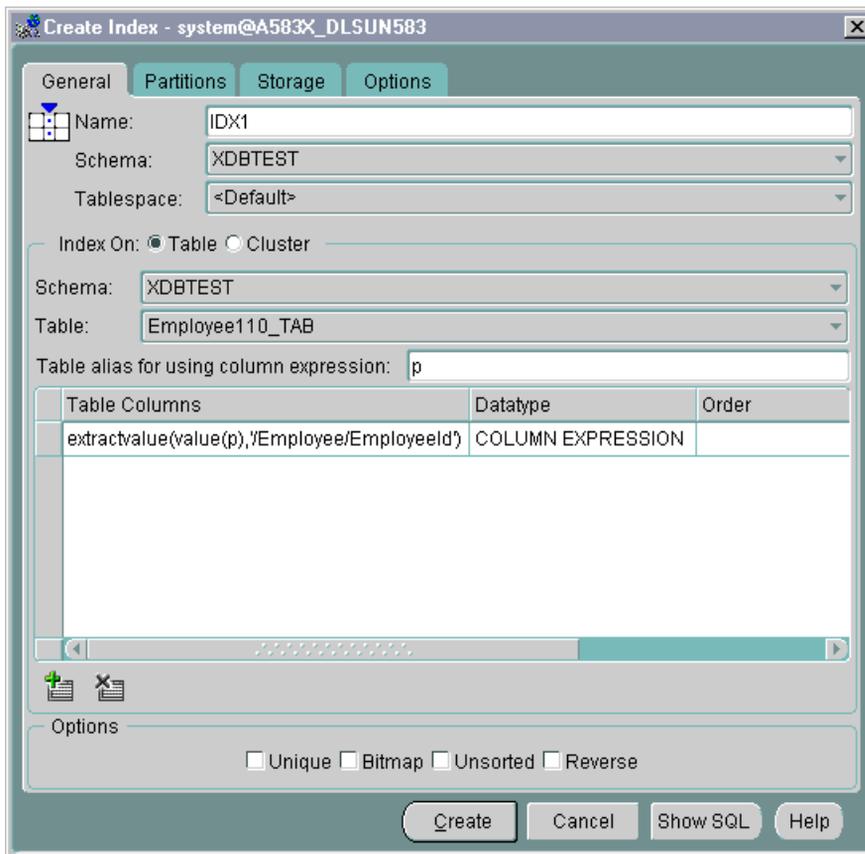
1. Provide the owner and name of the index. Under “Name” enter the name of the required index. Under “Schema” select the owner of the index from the drop-down list.
2. Under “Index On”, select Table.
3. Under the lower “Schema” select the table owner from the drop-down list.

4. Under “Table”, select either the `XMLType` table or table with `XMLType` column from the drop-down list.

You can also enter an alias for a column expression. You can specify this alias inside your function-based index statement.

5. *For tables with **XMLType** columns*, first click the lower left-hand “+” icon. This creates a new row for you to enter your extract XPath expression under “Table Columns”.
6. *For **XMLType** tables*, a new empty row is automatically created for you to create your extract XPath expression under “Table Columns”.
7. “Datatype” defaults to “Column Expression”. You do not need to change this.

Figure 21–26 Enterprise Manager: Creating a Function-Based Index Based on XPath Expressions



Loading XML Data into Oracle XML DB

This chapter describes how XML data can be loaded into Oracle XML DB using SQL*Loader.

It contains the following sections:

- [Loading XMLType Data into Oracle9i Database](#)
- [Using SQL*Loader to Load XMLType Columns](#)

Loading XMLType Data into Oracle9i Database

In Oracle9i Release 1 (9.0.1) and higher, both Export/Import utilities and SQL*Loader support XMLType as a column type. In other words, if your table has a column of type XMLType, it can be properly Exported and Imported from and to Oracle9i database, and you can load XML data into that column using SQL*Loader.

See Also: *Oracle9i Database Utilities*

Restoration

In the current release, Oracle XML DB Repository information is not exported when user data is exported. This means that the resources and all information is not exported.

Using SQL*Loader to Load XMLType Columns

XML columns are columns declared to be of type XMLType.

SQL*Loader treats XML columns as if they are CLOBs. All methods described in the following sections for loading LOB data from the primary datafile or from LOBFILES are applicable to loading XML columns.

See Also: *Oracle9i Database Utilities*

Note: You cannot specify an SQL string for LOB fields. This is true even if you specify LOBFILE_spec.

Because LOBs can be quite large, SQL*Loader is able to load LOB data from either a primary datafile (in line with the rest of the data) or from LOBFILES. This section addresses the following topics:

- Loading LOB Data from a Primary Datafile
- Loading LOB Data from an External LOBFILE (BFILE)
- Loading LOB Data from LOBFILES
- Loading LOB Data from a Primary Datafile

To load internal LOBs (BLOBs, CLOBs, and NCLOBs) or XML columns from a primary datafile, you can use the following standard SQL*Loader formats:

- Predetermined size fields
- Delimited fields
- Length-value pair fields

These formats is described in the following sections and in *Oracle9i Database Utilities*.

LOB Data in Predetermined Size Fields

This is a very fast and conceptually simple format in which to load LOBs.

Note: Because the LOBs you are loading may 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.

LOB Data in Delimited Fields

This format handles LOBs of different sizes within the same column (datafile field) without problem. However, this added flexibility can affect performance, because SQL*Loader must scan through the data, looking for the delimiter string.

As with single-character delimiters, when you specify string delimiters, you should consider the character set of the datafile. When the character set of the datafile is different than that of the control file, you can specify the delimiters in hexadecimal (that is, 'hexadecimal string'). If the delimiters are specified in hexadecimal notation, 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, the delimiter specification is considered to be in the client's (that is, the control file's) character set. In this case, the delimiter is converted into the datafile's character set before SQL*Loader searches for the delimiter in the datafile.

Loading LOB Data from LOBFILES

LOB data can be lengthy enough so that it makes sense to load it from a LOBFILE instead of from a primary datafile. In LOBFILES, LOB data instances are still considered to be in fields (predetermined size, delimited, length-value), but these fields are not organized into records (the concept of a record does not exist within LOBFILES). Therefore, the processing overhead of dealing with records is avoided. This type of organization of data is ideal for LOB loading.

There is no requirement that a LOB from a LOBFILE fit in memory. SQL*Loader reads LOBFILES in 64 KB chunks.

In LOBFILES the data can be in any of the following types of fields:

- 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)--VARRAW, VARCHAR, or VARCHARC loader datatypes--are used for loading from this type of field.

All of the previously mentioned field types can be used to load XML columns.

Dynamic Versus Static LOBFILE Specifications

You can specify LOBFILES either statically (you specify the actual 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 further 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, typically, the two fields will read the data independently.

Part VII

XML Data Exchange Using Advanced Queueing

Part VII of this manual describes XML data exchange using Oracle Advanced Queueing (AQ) and the AQs `XMLType` support. Part VII contains the following chapter:

- [Chapter 23, "Exchanging XML Data Using Advanced Queueing \(AQ\)"](#)

Exchanging XML Data Using Advanced Queueing (AQ)

This chapter describes how XML data can be exchanged using Oracle Advanced Queueing. It contains the following sections:

- [What Is AQ?](#)
- [How Do AQ and XML Complement Each Other?](#)
- [Internet Data Access Presentation \(IDAP\)](#)
- [IDAP Architecture](#)
- [Enqueue Using AQ XML Servlet](#)
- [Dequeue Using AQ XML Servlet](#)
- [IDAP and AQ XML Schemas](#)
- [Frequently Asked Questions \(FAQs\): XML and Advanced Queueing](#)

What Is AQ?

Oracle Advanced Queuing (AQ) provides database integrated message queuing functionality. AQ:

- Enables and manages asynchronous communication of two or more applications using messages
- Supports point-to-point and publish/subscribe communication models

Integration of message queuing with Oracle9i database brings the integrity, reliability, recoverability, scalability, performance, and security features of Oracle9i to message queuing. Integration with Oracle9i also facilitates the extraction of intelligence from message flows.

How Do AQ and XML Complement Each Other?

XML has emerged as a standard format for business communications. XML is being used not only to represent data communicated between business applications, but also, the business logic that is encapsulated in the XML.

In Oracle9i, AQ supports native XML messages and also allows AQ operations to be defined in the XML-based Internet-Data-Access-Presentation (IDAP) format. IDAP, an extensible message invocation protocol, is built on Internet standards, using HTTP and email protocols as the transport mechanism, and XML as the language for data presentation. Clients can access AQ using this.

AQ and XML Message Payloads

Figure 23-1 shows an Oracle9i database using AQ to communicate with three applications, with XML as the message payload. The general tasks performed by AQ in this scenario are:

- Message flow using subscription rules
- Message management
- Extracting business intelligence from messages
- Message transformation

This is an *intra-* and *inter-*business scenario where XML messages are passed asynchronously among applications using AQ.

- Intra-business. Typical examples of this kind of scenario include sales order fulfillment and supply-chain management.

- Inter-business processes. Here multiple integration hubs can communicate over the Internet backplane. Examples of inter-business scenarios include travel reservations, coordination between manufacturers and suppliers, transferring of funds between banks, and insurance claims settlements, among others.

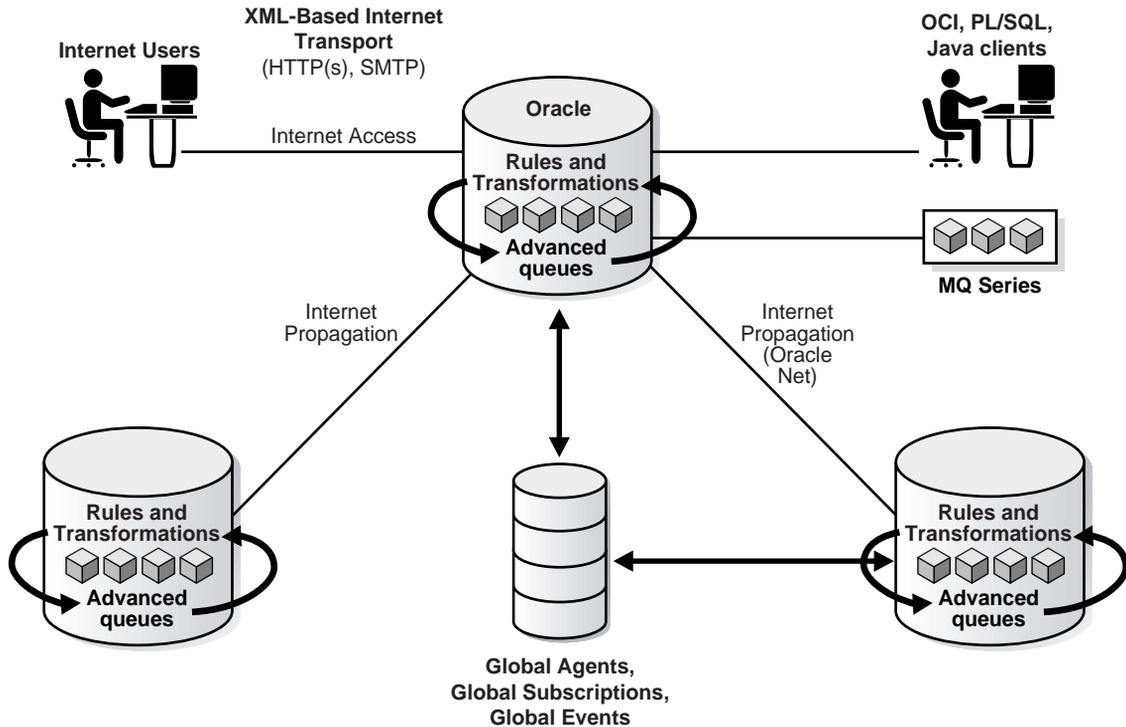
Oracle uses this in its enterprise application integration products. XML messages are sent from applications to an Oracle AQ hub. This serves as a "message server" for any application that wants the message. Through this hub-and-spoke architecture, XML messages can be communicated asynchronously to multiple loosely coupled receiving applications.

Figure 23–1 shows XML payload messages transported using AQ in the following ways:

- Web-based application that uses an AQ operation over an HTTP connection using IDAP
- An application that uses AQ to propagate an XML message over a Net* connection
- An application that uses AQ to propagate an internet/XML message directly to the database over HTTP or SMTP

The figure also shows that AQ clients can access data using OCI, Java, or PL/SQL.

Figure 23-1 Advanced Queuing and XML Message Payloads



AQ Enables Hub-and-Spoke Architecture for Application Integration

A critical challenge facing enterprises today is application integration. Application integration involves getting multiple departmental applications to cooperate, coordinate, and synchronize in order to execute complex business transactions.

AQ enables hub-and-spoke architecture for application integration. It makes integrated solution easy to manage, easy to configure, and easy to modify with changing business needs.

Messages Can Be Retained for Auditing, Tracking, and Mining

Message management provided by AQ is not only used to manage the flow of messages between different applications, but also, messages can be retained for future auditing and tracking, and extracting business intelligence.

Viewing Message Content with SQL Views

AQ also provides SQL views to look at the messages. These SQL views can be used to analyze the past, current, and future trends in the system.

Advantages of Using AQ

AQ provides the flexibility of *configuring communication* between different applications.

XMLType Attributes in Object Types

You can now create queues that use Oracle object types containing attributes of the new, opaque type, `XMLType`. 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 CLOBs
- Store more than one type of payload in a queue
- Query `XMLType` columns using the operators `ExistsNode()` and `SchemaMatch()`
- Specify the operators in subscriber rules or dequeue selectors

Internet Data Access Presentation (IDAP)

You can access AQ over the Internet by using Simple Object Access Protocol (SOAP). Internet Data Access Presentation (IDAP) is the SOAP specification for AQ operations. IDAP defines XML message structure for the body of the SOAP request. An IDAP-structured message is transmitted over the Internet using transport protocols such as HTTP or 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 the value:
`http://schemas.xmlsoap.org/soap/envelope/`
- The IDAP namespace has the value:
`http://ns.oracle.com/AQ/schemas/access`

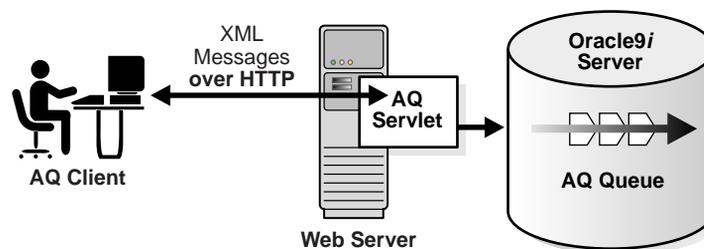
See Also: *Oracle9i Application Developer's Guide - Advanced Queuing*

IDAP Architecture

Figure 23–2 shows the following components needed to send HTTP messages:

- A client program that sends XML messages, conforming to IDAP format, to the AQ Servlet. This can be any HTTP client, such as, Web browsers.
- The Web server or ServletRunner which hosts the AQ servlet that can interpret the incoming XML messages, for example, Apache/Jserv or Tomcat.
- Oracle9i Server/Database. The AQ servlet connects to Oracle9i database to perform operations on your queues.

Figure 23–2 IDAP Architecture for Performing AQ Operations Using HTTP



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 CLOBs.
- Selectively dequeue messages with `XMLType` attributes using the operators `existsNode()`, `extract()`, and so on.
- Define transformations to convert Oracle objects to `XMLType`.
- Define rule-based subscribers that query message content using `XMLType` operators such as `existsNode()` and `extract()`.

Example 23-1 Using AQ and XMLType Queue Payloads: Creating the Overseas Shipping Queue Table and Queue and Transforming the Representation

In the BooksOnline application, assume that the Overseas Shipping site represents the order as `SYS.XMLType`. The Order Entry site represents the order as an Oracle object, `ORDER_TYP`.

The Overseas queue table and queue are created as follows:

```
BEGIN
dbms_aqadm.create_queue_table(
  queue_table      => 'OS_orders_pr_mqtab',
  comment          => 'Overseas Shipping MultiConsumer Orders queue table',
  multiple_consumers => TRUE,
  queue_payload_type => 'SYS.XMLType',
  compatible       => '8.1');
END;

BEGIN
dbms_aqadm.create_queue (
  queue_name      => 'OS_bookedorders_que',
  queue_table     => 'OS_orders_pr_mqtab');
END;
```

Since the representation of orders at the Overseas Shipping site is different from the representation of orders at the Order Entry site, a transformation is applied before messages are propagated from the Order Entry site to the Overseas Shipping site.

```
/* Add a rule-based subscriber (for Overseas Shipping) to the Booked orders
queues with Transformation. Overseas Shipping handles all non-US orders: */
DECLARE
  subscriber      aq$_agent;
BEGIN
  subscriber := aq$_agent('Overseas_Shipping', 'OS.OS_bookedorders_que', null);
```

```

dbms_aqadm.add_subscriber(
  queue_name      => 'OE.OE_bookedorders_que',
  subscriber      => subscriber,
  rule            => 'tab.user_data.orderregion = ''INTERNATIONAL''',
  transformation  => 'OS.OE2XML');
END;

```

For more details on defining transformations that convert the type used by the Order Entry application to the type used by Overseas shipping, see *Oracle9i Application Developer's Guide - Advanced Queuing* the section on Creating Transformations in Chapter 8.

Example 23–2 Using AQ and XMLType Queue Payloads: Dequeuing Messages

Assume that an application processes orders for customers in Canada. This application can dequeue messages using the following procedure:

```

/* Create procedures 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;
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 := 'tab.user_data.extract(
  ''/ORDER_TYP/CUSTOMER/COUNTRY/text()'').getStringVal()='CANADA''';

  dopt.consumer_name := 'Overseas_Shipping';

  WHILE (new_orders) LOOP
    BEGIN
      dbms_aq.dequeue(
        queue_name      => 'OS.OS_bookedorders_que',
        dequeue_options => dopt,
        message_properties => mprop,
        payload         => deq_order_data,
        msgid           => deq_msgid);
      commit;
    END;
  END LOOP;

```

```

        dbms_output.put_line(' Order for Canada - Order: ' ||
                               deq_order_data.getStringVal());

EXCEPTION
    WHEN no_messages THEN
        dbms_output.put_line (' ---- NO MORE ORDERS ---- ');
        new_orders := FALSE;

    END;
END LOOP;

end;
```

Enqueue Using AQ XML Servlet

You can perform enqueue requests over the Internet using IDAP.

See Also: *Oracle9i Application Developer's Guide - Advanced Queuing* for further information about sending AQ requests using IDAP.

Scenario

In the BooksOnLine application, the Order Entry system uses a priority queue to store booked orders. Booked orders are propagated to the regional booked orders queues. At each region, orders in these regional booked orders queues are processed in the order of the shipping priorities.

Example 23–3 PL/SQL (DBMS_AQADM Package)

The following calls create the priority queues for the Order Entry application.

```

/* Create a priority queue table for OE: */
EXECUTE dbms_aqadm.create_queue_table( \
    queue_table          => 'OE_orders_pr_mqtab', \
    sort_list            => 'priority,enq_time', \
    comment              => 'Order Entry Priority \
                          MultiConsumer Orders queue table', \
    multiple_consumers  => TRUE, \
    queue_payload_type  => 'BOLADM.order_typ', \
    compatible          => '8.1', \
    primary_instance    => 2, \
    secondary_instance  => 1);

EXECUTE dbms_aqadm.create_queue ( \
```

```
queue_name      => 'OE_bookedorders_que', \
queue_table     => 'OE_orders_pr_mqtab' );
```

Assume that a customer, John, wants to send an enqueue request using SOAP. The XML message will have the following format.

```
<?xml version="1.0"?>
  <Envelope xmlns= "http://schemas.xmlsoap.org/soap/envelope/">
    <Body>
      <AQXmlSend xmlns = "http://ns.oracle.com/AQ/schemas/access">
        <producer_options>
          <destination>OE.OE_bookedorders_que</destination>
        </producer_options>

        <message_set>
          <message_count>1</message_count>

          <message>
            <message_number>1</message_number>
            <message_header>
              <correlation>ORDER1</correlation>
              <priority>1</priority>
              <sender_id>
                <agent_name>john</agent_name>
              </sender_id>
            </message_header>

            <message_payload>
              <ORDER_TYP>
                <ORDERNO>100</ORDERNO>
                <STATUS>NEW</STATUS>
                <ORDERTYPE>URGENT</ORDERTYPE>
                <ORDERREGION>EAST</ORDERREGION>
                <CUSTOMER>
                  <CUSTNO>1001233</CUSTNO>
                  <CUSTID>JOHN</CUSTID>
                  <NAME>JOHN DASH</NAME>
                  <STREET>100 EXPRESS STREET</STREET>
                  <CITY>REDWOOD CITY</CITY>
                  <STATE>CA</STATE>
                  <ZIP>94065</ZIP>
                  <COUNTRY>USA</COUNTRY>
                </CUSTOMER>
                <PAYMENTMETHOD>CREDIT</PAYMENTMETHOD>
                <ITEMS>
```

```

<ITEMS_ITEM>
  <QUANTITY>10</QUANTITY>
  <ITEM>
    <TITLE>Perl handbook</TITLE>
    <AUTHORS>Randal</AUTHORS>
    <ISBN>345620200</ISBN>
    <PRICE>19</PRICE>
  </ITEM>
  <SUBTOTAL>190</SUBTOTAL>
</ITEMS_ITEM>
<ITEMS_ITEM>
  <QUANTITY>10</QUANTITY>
  <ITEM>
    <TITLE>JDBC guide</TITLE>
    <AUTHORS>Taylor</AUTHORS>
    <ISBN>123420212</ISBN>
    <PRICE>59</PRICE>
  </ITEM>
  <SUBTOTAL>590</SUBTOTAL>
</ITEMS_ITEM>
</ITEMS>
<CCNUMBER>NUMBER01</CCNUMBER>
<ORDER_DATE>08/23/2000 12:45:00</ORDER_DATE>
</ORDER_TYP>
</message_payload>
</message>
</message_set>

<AQXmlCommit/>
</AQXmlSend>
</Body>
</Envelope>

```

Dequeue Using AQ XML Servlet

You can perform dequeue requests over the Internet using SOAP.

See Also: *Oracle9i Application Developer's Guide - Advanced Queuing* for information about receiving AQ messages using SOAP.

In the BooksOnline scenario, assume that the East shipping application receives AQ messages with a correlation identifier 'RUSH' over the Internet.

Example 23–4 Receiving and Dequeuing AQ XML Messages

The dequeue request will have the following format:

```
<?xml version="1.0"?>
<Envelope xmlns= "http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <AQXmlReceive xmlns = "http://ns.oracle.com/AQ/schemas/access">
      <consumer_options>
        <destination>ES_ES_bookedorders_que</destination>
        <consumer_name>East_Shipping</consumer_name>
        <wait_time>0</wait_time>
        <selector>
          <correlation>RUSH</correlation>
        </selector>
      </consumer_options>

      <AQXmlCommit/>

    </AQXmlReceive>
  </Body>
</Envelope>
```

IDAP and AQ XML Schemas

IDAP exposes a SOAP and AQ XML schema to the client. All documents sent by the parser are validated against these schemas:

- SOAP schema — <http://schemas.xmlsoap.org/soap/envelope/>
- AQ XML schema — <http://ns.oracle.com/AQ/schemas/access>

See Also:

- *Oracle9i Application Developer's Guide - Advanced Queuing*, Chapter 8, for more detail on how to implement structured message payloads applications using either DBMS_AQADM or Java (JDBC)
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information about DBMS_TRANSFORM.

Frequently Asked Questions (FAQs): XML and Advanced Queuing

Can I Store AQ XML Messages with Many PDFs as One Record?

I am exchanging XML documents from one business area to another using Oracle Advanced Queuing. Each message received or sent includes an XML header, XML attachment (XML data stream), DTDs, and PDF files. I need to store all this information, including some imagery files, in the database table, in this case, the `queuetable`.

Can I enqueue this message into an Oracle queue table as one record or one piece? Or do I have to enqueue this message as multiple records, such as one record for XML data streams as CLOB type, one record for PDF files as RAW type? Then somehow specify that these sets of records are correlated? Also, I want to ensure that I dequeue this.

Answer: You can achieve this in the following ways:

- You can either define an object type with (CLOB, RAW,...) attributes, and store it as a single message
- You can use the AQ message grouping feature and store it in multiple messages. But the message properties will be associated with a group. To use the message grouping feature, all messages must be the same payload type.

Do I Specify Payload Type as CLOB First, Then Enqueue and Store?

[Follow on question from preceding FAQ] Do I specify the payload type as CLOB first, then enqueue and store all the pieces, XML message data stream, DTDs, and PDF,... as a single message payload in the Queue table? If so, how can I separate this single message into individual pieces when I dequeue this message?

Answer: No. You 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.

Can I Add New Recipients After Messages Are Enqueued?

I want to use the queue table to support message assignments. For example, when other business areas send messages to Oracle, they do not know who should be assigned to process these messages, but they know the messages are for Human Resources (HR). So all messages will go to the HR supervisor.

At this point, the message has been enqueued in the queue table. The HR supervisor is the only recipient of this message, and the entire HR staff have been pre-defined as subscribers for this queue). Can the HR supervisor add new recipients, namely additional staff, to the message_properties.recipient_list on the existing the message in the queue table?

I do not have multiple consumers (recipients) when the messages are enqueued, but I want to replace the old recipient, or add new recipients after the message has already been in the queue table. This new message will then be dequeued by the new recipient. Is this workable? Or do I have to remove the message from old recipient, then enqueue the same message contents to the new recipient?

Answer: You cannot change the recipient list after the message is enqueued. If you do not specify a recipient list then subscribers can subscribe to the queue and dequeue the message.

In your case, the new recipient should be a subscriber to the queue. Otherwise, you will have to dequeue the message and enqueue it again with the new recipient.

How Does Oracle Enqueue and Dequeue and Process XML Messages?

In an OTN document, it says that an Oracle database can enqueue and dequeue XML messages and process them. How does it do this?

Do I have to use XML SQL Utility (XSU) in order to insert an XML file into a table before processing it, or can I enqueue an XML file directly, parse it, and dispatch its messages through the AQ process? Must I use XML SQL Utility every time I want to INSERT or UPDATE XML data into Oracle9i Database?

Answer: AQ supports enqueueing and dequeuing objects. These objects can have an attribute of type `XMLType` containing an XML Document, as well as other interested “factored out” metadata attributes that might make sense to send along with the

message. Refer to the latest AQ document, *Oracle9i Application Developer's Guide - Advanced Queuing* to get specific details and see more examples.

How Can I Parse Messages with XML Content from AQ Queues?

I need a tool to parse messages with XML content, from an AQ queue and then update tables and fields in an ODS (Operational Data Store). In short, I want to retrieve and parse XML documents and map specific fields to database tables and columns. Is Oracle9i Text a solution?

I can use XML SQL Utility (XSU) if I go with a custom solution. My main concentration is supply-chain. I want to get metadata information such as, AQ enqueue/dequeue times, JMS header information,... based on queries on certain XML tag values. Can I just store the XML in a CLOB and issue queries using Oracle9i Text?

Answer: The easiest way to do this is using Oracle XML Parser for Java and Java Stored Procedures in tandem with AQ inside Oracle9i.

Regarding the use of XSU:

- If you store XML as CLOBs then you can definitely search it using Oracle9i Text (aka *interMedia Text*), but this only helps you find a particular message that matches a criteria.
- If you need to do aggregation operations over the metadata, view the metadata from existing relational tools, or use normal SQL predicates on the metadata, then having it “only” stored as XML in a CLOB is not going to be good enough.

You can combine Oracle9i Text XML searching with some amount of redundant metadata storage as “factored out” columns and use SQL statements that combine normal SQL predicates with the Oracle9i Text CONTAINS() clause to have the best of both.

See Also: [Chapter 7, "Searching XML Data with Oracle Text"](#).

Can I Prevent the Listener from Stopping Until the XML Document Is Processed?

I receive XML messages from clients as messages and need to process them as soon as they come in. Each XML document takes about 15 seconds to process. I am using PL/SQL. One PL/SQL procedure starts the listener and dequeues the message and calls another procedure to process the XML document. The problem is that the listener is held up until the XML document is processed. Meanwhile messages accumulate in the queue.

What is the best way to handle this? Is there a way for the listener program to call the XML processing procedure asynchronously and return to listening? Java is not an option at this point.

Answer: After receiving the message, you can submit a job using the `DBMS_JOB` package. The job will be invoked asynchronously in a different database session.

Oracle9i has added PL/SQL callbacks in the AQ notification framework. This allows you register a PL/SQL callback which is invoked asynchronously when a message shows up in a queue.

How Can I Use HTTPS with AQ?

I need to send XML messages to suppliers using HTTPS and get a response back.

Sending a message using HTTP does not appear to much of a problem using something like `java.net.URLConnection`, however there does not seem to be anything about making a HTTPS connection. Products like Portal and OC4J seem to have an HTTP client in the source code does Oracle/anyone have one for HTTPS connections which can be reused?

Answer: You can use Internet access functionality of Oracle9i AQ. Using this functionality, you can enqueue and dequeue messages over HTTP(S) securely and transactionally using XML. You can get more details on this functionality at:

<http://otn.oracle.com/products/aq>

What Are the Options for Storing XML in AQ Message Payloads?

When storing XML in AQ message payloads, is there any other way of natively doing this other than having an ADT as the payload with `sys.xmltype` as part of the ADT? For example, create or replace type object `xml_data_typ` AS object (`xml_data sys.XMLType`); My understanding is that you can ONLY have either a RAW or ADT as message payloads.

Answer: In Oracle9.0.1, this is the only way to store `XMLTypes` in queues. In Oracle9i Release 2 (9.2), you can create queues with payload and attributes as `XMLType`.

Can We Compare IDAP and SOAP?

Answer: IDAP is the SOAP specification for AQ operations. IDAP is the XML specification for AQ operations. SOAP defines a generic mechanism to invoke a service. IDAP defines these mechanisms to perform AQ operations.

IDAP in addition has the following key properties not defined by SOAP:

- **Transactional behavior.** You can perform AQ operations in a transactional manner. Your transaction can span multiple IDAP requests.
- **Security.** All the IDAP operations can be done only by authorized and authenticated users.

Installing and Configuring Oracle XML DB

This appendix describes the ways you can manage and configure your Oracle XML DB applications. It contains the following sections:

- [Installing Oracle XML DB](#)
- [Installing or Reinstalling Oracle XML DB from Scratch](#)
- [Upgrading an Existing Oracle XML DB Installation](#)
- [Configuring Oracle XML DB](#)
- [Oracle XML DB Configuration Example](#)
- [Oracle XML DB Configuration API](#)

Installing Oracle XML DB

You will need to install Oracle XML DB under the following conditions:

- ["Installing or Reinstalling Oracle XML DB from Scratch"](#) on page A-2
- ["Upgrading an Existing Oracle XML DB Installation"](#) on page A-4

Installing or Reinstalling Oracle XML DB from Scratch

You can perform a new installation of Oracle XML DB with or without Database Configuration Assistant (DBCA):

Installing a New Oracle XML DB with DBCA

Oracle XML DB is part of the seed database and installed by DBCA as part of database installation by default. No additional steps are required to install Oracle XML DB, however, if you choose to install “Customized” database, you can configure Oracle XML DB tablespace and FTP, HTTP, and WebDAV port numbers.

By default DBCA performs the following tasks:

- Creates an Oracle XML DB tablespace for Oracle XML DB Repository
- Enables all protocol access
- Configures FTP at port 2100
- Configures HTTP/WebDAV at port 8080

The Oracle XML DB tablespace holds the data that is stored in Oracle XML DB Repository. This includes data that is stored in the Repository using:

- SQL, for example using `resource_view` and `path_view`
- Protocols such as FTP, HTTP, and WebDAV.

It is possible to store data in tables outside this tablespace and access that data through the Repository by having REFs to that data stored in the tables in this tablespace.

Warning: The Oracle XML DB tablespace should not be dropped. If it is dropped it renders all Repository data inaccessible.

Post Installation

Oracle XML DB uses dynamic protocol registration to setup FTP and HTTP listener service with the local listener. So, make certain that the listener is up, when accessing any Oracle XML DB protocols.

To allow for unauthenticated access to your Oracle XML DB Repository data through HTTP, you must unlock the ANONYMOUS user account.

Note: If the Listener is running on a non-standard port (for example, not 1521) then in order for the protocols to register with the correct listener the `init.ora` file must contain a `local_listener` entry. This references a `TNSNAME` entry that points to the correct listener. After editing the `init.ora` parameter you must regenerate the `SPFILE` entry using `CREATE SPFILE`.

Installing a New Oracle XML DB Manually Without DBCA

After the database installation, you must run the following SQL scripts in `rdbms/admin` connecting to `SYS` to install Oracle XML DB after creating a new tablespace for Oracle XML DB Repository. Here is the syntax for this:

```
catqm.sql <xdb_pass> <XDB_TS_NAME> <TEMP_TS_NAME> #Create the tables and views
needed to run XDB
```

For example:

```
catqm.sql change_on_install XDB TEMP
```

Reconnect to `SYS` again and run the following:

```
catxdbj.sql          #Load xdb java library
```

Note: Make sure that the database is started with Oracle9i Release 2 (9.2.0) compatibility or higher.

Post Installation

After the manual installation, carry out these tasks:

1. Add the following dispatcher entry to the `init.ora` file:

```
dispatchers="(PROTOCOL=TCP) (SERVICE=<sid>XDB)"
```

2. Restart database and listener to enable Oracle XML DB protocol access.
3. To allow for unauthenticated access to your Oracle XML DB Repository data through HTTP, you must also unlock the `ANONYMOUS` user account.

Reinstalling Oracle XML DB

To reinstall Oracle XML DB, run following SQL commands connecting to SYS to drop Oracle XML D user and tablespace:

Note: All user data stored in Oracle XML DB Repository is also last when you drop xdb user!

```
drop user xdb cascade;
alter tablespace <XDB_TS_NAME> offline;
drop tablespace <XDB_TS_NAME> including contents;
```

Install Oracle XML DB manually as described in ["Installing a New Oracle XML DB Manually Without DBCA"](#) on page A-3.

Upgrading an Existing Oracle XML DB Installation

Run the script, `catproc.sql`, as always.

As a post upgrade step, if you want Oracle XML DB functionality, you must install Oracle XML DB manually as described in ["Installing a New Oracle XML DB Manually Without DBCA"](#) on page A-2.

Configuring Oracle XML DB

The following sections describe how to configure Oracle XML DB. You can also configure Oracle XML DB using Oracle Enterprise Manager.

See Also: [Chapter 21, "Managing Oracle XML DB Using Oracle Enterprise Manager"](#)

Oracle XML DB is managed through a configuration resource stored in Oracle XML DB Repository, `/sys/xdbconfig.xml`.

The Oracle XML DB configuration file is alterable at runtime. Simply updating the configuration file, causes a new version of the file to be generated. At the start of

each session, the current version of the configuration is bound to that session. The session will use this configuration for its life, unless you invoke an explicit call to refresh to the latest configuration.

Oracle XML DB Configuration File, xdbconfig.xml

Oracle XML DB configuration is stored as an XML resource, /xdbconfig.xml conforming to the Oracle XML DB configuration XML schema:

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

To configure or modify the configuration of Oracle XML DB, update the /xdbconfig.xml file by inserting, removing, or editing the appropriate XML elements in xdbconfig.xml.

Oracle XML DB configuration XML schema has the following structure:

Top Level Tag <xdbconfig>

A top level tag, <xdbconfig> is divided into two sections:

- <sysconfig> This keeps system-specific, built-in parameters.
- <userconfig> This allows users to store new custom parameters.

The following describes the syntax:

```
<xdbconfig>
  <sysconfig> ... </sysconfig>
  <userconfig> ... </userconfig>
</xdbconfig>
```

<sysconfig>

The <sysconfig> section is further subdivided as follows:

```
<sysconfig>
  General parameters
  <protocolconfig> ... </protocolconfig>
</sysconfig>
```

It stores several general parameters that apply to all Oracle XML DB, for example, the maximum age for an ACL, whether Oracle XML DB should be case sensitive, and so on.

Protocol-specific parameters are grouped inside the <protocolconfig> tag.

<userconfig>

The <userconfig> section contains any parameters that you may want to add.

<protocolconfig>

The structure of the <protocolconfig> section is as follows:

```
<protocolconfig>
  <common> ... </common>
  <httpconfig> ... </httpconfig>
  <ftpconfig> ... </ftpconfig>
</protocolconfig>
```

Under <common> Oracle9i stores parameters that apply to all protocols, such as MIME type information. There are also HTTP and FTP specific parameters under sections <httpconfig> and <ftpconfig> respectively.

<httpconfig>

Inside <httpconfig> there is a further subsection, <webappconfig> that corresponds to Web-based applications. It includes Web application specific parameters, for example, icon name, display name for the application, list of servlets in Oracle XML DB, and so on.

See Also:

- [Chapter 19, "Using FTP, HTTP, and WebDAV Protocols"](#), [Table 19-1](#), [Table 19-2](#), and [Table 19-3](#), for a list of protocol configuration parameters.
- [Chapter 20, "Writing Oracle XML DB Applications in Java"](#), ["Configuring the Oracle XML DB Example Servlet"](#) on page 20-12.

Oracle XML DB Configuration Example

The following is a sample Oracle XML DB configuration file:

Example A-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>
  <invalid-pathname-chars>,</invalid-pathname-chars>
  <call-timeout>300</call-timeout>
  <max-session-use>100</max-session-use>
  <default-lock-timeout>3600</default-lock-timeout>
  <xdbcore-logfile-path>/sys/log/xdblog.xml</xdbcore-logfile-path>
  <xdbcore-log-level>1</xdbcore-log-level>

  <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>

        <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>
      </common>
    </protocolconfig>
  </sysconfig>

```

```
        </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>
</ftpconfig>

<httpconfig>
    <http-port>8080</http-port>
    <http-listener>local_listener</http-listener>
    <http-protocol>tcp</http-protocol>
    <session-timeout>6000</session-timeout>
    <server-name>XDB HTTP Server</server-name>
    <max-header-size>16384</max-header-size>
    <max-request-body>2000000000</max-request-body>
    <logfile-path>/sys/log/httplog.xml</logfile-path>
    <log-level>0</log-level>
    <servlet-realm>Basic realm="XDB"</servlet-realm>
<webappconfig>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
    </welcome-file-list>
    <error-pages>
</error-pages>
    <servletconfig>
        <servlet-mappings>
            <servlet-mapping>
                <servlet-pattern>/oradb/*</servlet-pattern>
                <servlet-name>DBURIServlet</servlet-name>
            </servlet-mapping>
        </servlet-mappings>

        <servlet-list>
            <servlet>
```

```

        <servlet-name>DBURIServlet</servlet-name>
        <display-name>DBURI</display-name>
        <servlet-language>C</servlet-language>
        <description>Servlet for accessing DBURIs</description>
        <security-role-ref>
            <role-name>authenticatedUser</role-name>
            <role-link>authenticatedUser</role-link>
        </security-role-ref>
    </servlet>
</servlet-list>
</servletconfig>
</webappconfig>
</httpconfig>
</protocolconfig>
</sysconfig>

<userconfig><numusers>40</numusers></userconfig>

</xdbcconfig>

```

Oracle XML DB Configuration API

The Oracle XML DB Configuration API can be accessed just like any other XML schema-based resource in the hierarchy. It can be accessed and manipulated using FTP, HTTP, WebDav, Oracle Enterprise Manager, or any of the resource and DOM APIs for Java or PL/SQL.

For convenience, there is a PL/SQL API provided as part of the `DBMS_XDB` package for configuration access. It exposes the following functions:

Get Configuration, `cfg_get()`

The `cfg_get()` function returns a copy of the configuration as an `XMLType`:

```
DBMS_XDB.CFG_GET() RETURN SYS.XMLTYPE
```

`cfg_get()` is auto-commit.

Update Configuration, `cfg_update()`

The `cfg_update()` function updates the configuration with a new one:

```
DBMS_XDB.CFG_UPDATE(newconfig SYS.XMLTYPE)
```

Example A-2 Updating the Configuration File Using `cfg_update()` and `cfg_get()`

If you have a few parameters to update in the configuration file, you can use the following :

```
BEGIN
DBMS_XMLDB.CFG_UPDATE(UPDATEXML(UPDATEXML
    (DBMS_XMLDB.CFG_GET(),
    '/xdbconfig/descendant::ftp-port/text()', '2121'),
    '/xdbconfig/descendant::http-port/text()',
    '19090'))
END;
/
```

If you have many parameters to update, the preceding example may prove too cumbersome. Use instead FTP, HTTP, or Oracle Enterprise Manager.

Refresh Configuration, `cfg_refresh()`

The `cfg_refresh()` function updates the configuration snapshot to correspond to the latest version on disk at that instant:

```
DBMS_XMLDB.CFG_REFRESH()
```

Typically, `cfg_refresh()` is called in one of the following scenarios:

- 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.
- If updates to the configuration are made, Oracle XML DB Configuration API is aware of them.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*
the chapter on `DBMS_XMLDB`

XML Schema Primer

This appendix includes introductory information about the W3C XML Schema Recommendation. It contains the following sections:

- [Introducing XML Schema](#)
- [XML Schema Components](#)
- [Simple Types](#)
- [Anonymous Type Definitions](#)
- [Element Content](#)
- [Annotations](#)
- [Building Content Models](#)
- [Attribute Groups](#)
- [Nil Values](#)
- [Building Content Models](#)
- [XML Schema Example, PurchaseOrder.xsd](#)

Introducing XML Schema

Parts of this introduction are extracted from W3C XML Schema notes.

See Also:

- <http://www.w3.org/TR/xmlschema-0/> **Primer**
- <http://www.w3.org/TR/xmlschema-1/> **Structures**
- <http://www.w3.org/TR/xmlschema-2/> **Datatypes**
- <http://w3.org/XML/Schema>
- <http://www.oasis-open.org/cover/schemas.html>
- <http://www.xml.com/pub/a/2000/11/29/schemas/part1.html>

An XML schema (referred to in this appendix as schema) defines a class of XML documents. The term “instance document” is often used to describe an XML document that conforms to a particular XML schema. However, neither instances nor schemas need to exist as documents, they may exist as streams of bytes sent between applications, as fields in a database record, or as collections of XML Infoset “Information Items”. But to simplify the description in this appendix, instances and schemas are referred to as if they are documents and files.

Purchase Order, po.xml

Consider the following instance document in an XML file `po.xml`. It describes a purchase order generated by a home products ordering and billing application:

```
<?xml version="1.0"?>
  <purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
      <name>Alice Smith</name>
      <street>123 Maple Street</street>
      <city>Mill Valley</city>
      <state>CA</state>
      <zip>90952</zip>
    </shipTo>
    <billTo country="US">
      <name>Robert Smith</name>
      <street>8 Oak Avenue</street>
      <city>Old Town</city>
      <state>PA</state>
      <zip>95819</zip>
```

```

</billTo>
<comment>Hurry, my lawn is going wild!</comment>
<items>
  <item partNum="872-AA">
    <productName>Lawnmower</productName>
    <quantity>1</quantity>
    <USPrice>148.95</USPrice>
    <comment>Confirm this is electric</comment>
  </item>
  <item partNum="926-AA">
    <productName>Baby Monitor</productName>
    <quantity>1</quantity>
    <USPrice>39.98</USPrice>
    <shipDate>1999-05-21</shipDate>
  </item>
</items>
</purchaseOrder>

```

The purchase order consists of a main element, `purchaseOrder`, and the subelements `shipTo`, `billTo`, `comment`, and `items`. These subelements (except `comment`) in turn contain other subelements, and so on, until a subelement such as `USPrice` contains a number rather than any subelements.

- **Complex Type Elements.** Elements that contain subelements or carry attributes are said to have complex types
- **Simple Type Elements.** Elements that contain numbers (and strings, and dates, and so on) but do not contain any subelements are said to have simple types. Some elements have attributes; attributes always have simple types.

The complex types in the instance document, and some simple types, are defined in the purchase order schema. The other simple types are defined as part of XML Schema's repertoire of built-in simple types.

Association Between the Instance Document and Purchase Order Schema

The purchase order schema is not mentioned in the XML instance document. An instance is not actually required to reference an XML schema, and although many will. It is assumed that any processor of the instance document can obtain the purchase order XML schema without any information from the instance document. Later, you will see the explicit mechanisms for associating instances and XML schemas.

Purchase Order Schema, po.xsd

The purchase order schema is contained in the file `po.xsd`:

Purchase Order Schema, po.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```

    <xsd:element name="quantity">
      <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
          <xsd:maxExclusive value="100"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="USPrice" type="xsd:decimal"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="partNum" type="SKU" use="required"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>

<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>

```

The purchase order schema consists of a schema element and a variety of subelements, most notably elements, `complexType`, and `simpleType` which determine the appearance of elements and their content in the XML instance documents.

Prefix `xsd`:

Each of the elements in the schema has a prefix `xsd`: which is associated with the XML Schema namespace through the declaration, `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`, that appears in the schema element. The prefix `xsd`: is used by convention to denote the XML Schema namespace, although any prefix can be used. The same prefix, and hence the same association, also appears on the names of built-in simple types, such as, `xsd:string`. This identifies the elements and simple types as belonging to the vocabulary of the XML Schema language rather than the vocabulary of the schema author. For clarity, this description uses the names of elements and simple types, for example, `simpleType`, and omits the prefix.

XML Schema Components

Schema component is the generic term for the building blocks that comprise the abstract data model of the schema. An XML Schema is a set of -schema components-. There are 13 kinds of component in all, falling into three groups.

Primary Components

The primary components, which may (type definitions) or must (element and attribute declarations) have names are as follows:

- Simple type definitions
- Complex type definitions
- Attribute declarations
- Element declarations

Secondary Components

The secondary components, which must have names, are as follows:

- Attribute group definitions
- Identity-constraint definitions
- Model group definitions
- Notation declarations

Helper Components

Finally, the “helper” components provide small parts of other components; they are not independent of their context:

- Annotations
- Model groups
- Particles
- Wildcards
- Attribute Uses

Complex Type Definitions, Element and Attribute Declarations

In XML Schema, there is a basic difference between complex and simple types:

- Complex types, allow elements in their content and may carry attributes
- Simple types, cannot have element content and cannot carry attributes.

There is also a major distinction between the following:

- *Definitions* which create new types (both simple and complex)
- *Declarations* which enable elements and attributes with specific names and types (both simple and complex) to appear in document instances

This section defines complex types and declares elements and attributes that appear within them.

New complex types are defined using the `complexType` element and such definitions typically contain a set of element declarations, element references, and attribute declarations. The declarations are not themselves types, but rather an association between a name and the constraints which govern the appearance of that name in documents governed by the associated schema. Elements are declared using the `element` element, and attributes are declared using the `attribute` element.

Defining the USAddress Type

For example, `USAddress` is defined as a complex type, and within the definition of `USAddress` you see five element declarations and one attribute declaration:

```
<xsd:complexType name="USAddress" >
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
```

Hence any element appearing in an instance whose type is declared to be `USAddress`, such as, `shipTo` in `po.xml`, must consist of five elements and one attribute. These elements must:

- Be called `name`, `street`, `city`, `state`, and `zip` as specified by the values of the declarations' name attributes
- Appear in the same sequence (order) in which they are declared. The first four of these elements will each contain a string, and the fifth will contain a number.

The element whose type is declared to be `USAddress` may appear with an attribute called `country` which must contain the string `US`.

The `USAddress` definition contains only declarations involving the simple types: `string`, `decimal`, and `NMTOKEN`.

Defining `PurchaseOrderType`

In contrast, the `PurchaseOrderType` definition contains element declarations involving complex types, such as, `USAddress`, although both declarations use the same type attribute to identify the type, regardless of whether the type is simple or complex.

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

In defining `PurchaseOrderType`, two of the element declarations, for `shipTo` and `billTo`, associate different element names with the *same complex type*, namely `USAddress`. The consequence of this definition is that any element appearing in an instance document, such as, `po.xml`, whose type is declared to be `PurchaseOrderType` must consist of elements named `shipTo` and `billTo`, each containing the five subelements (`name`, `street`, `city`, `state`, and `zip`) that were declared as part of `USAddress`. The `shipTo` and `billTo` elements may also carry the `country` attribute that was declared as part of `USAddress`.

The `PurchaseOrderType` definition contains an `orderDate` attribute declaration which, like the `country` attribute declaration, identifies a simple type. In fact, *all attribute declarations must reference simple types* because, unlike element declarations, attributes cannot contain other elements or other attributes.

The element declarations we have described so far have each associated a name with an existing type definition. Sometimes it is preferable to use an existing element rather than declare a new element, for example:

```
<xsd:element ref="comment" minOccurs="0"/>
```

This declaration references an existing element, `comment`, declared elsewhere in the purchase order schema. In general, the value of the `ref` attribute must reference a

global element, on other words, one that has been declared under schema rather than as part of a complex type definition. The consequence of this declaration is that an element called `comment` may appear in an instance document, and its content must be consistent with that element's type, in this case, string.

Occurrence Constraints: `minOccurs` and `maxOccurs`

The `comment` element is optional in `PurchaseOrderType` because the value of the `minOccurs` attribute in its declaration is 0. In general, an element is required to appear when the value of `minOccurs` is 1 or more. The maximum number of times an element may appear is determined by the value of a `maxOccurs` attribute in its declaration. This value may be a positive integer such as 41, or the term unbounded to indicate there is no maximum number of occurrences. The default value for both the `minOccurs` and the `maxOccurs` attributes is 1.

Thus, when an element such as `comment` is declared without a `maxOccurs` attribute, the element may not occur more than once. If you specify a value for only the `minOccurs` attribute, make certain that it is less than or equal to the default value of `maxOccurs`, that is, it is 0 or 1.

Similarly, if you specify a value for only the `maxOccurs` attribute, it must be greater than or equal to the default value of `minOccurs`, that is, 1 or more. If both attributes are omitted, the element must appear exactly once.

Attributes may appear once or not at all, but no other number of times, and so the syntax for specifying *occurrences of attributes* is different from the syntax for elements. In particular, attributes can be declared with a `use` attribute to indicate whether the attribute is required, optional, or even prohibited. Recall for example, the `partNum` attribute declaration in `po.xsd`:

```
<xsd:attribute name="partNum" type="SKU" use="required"/>
```

Default Attributes

Default values of both attributes and elements are declared using the default attribute, although this attribute has a slightly different consequence in each case. When an attribute is declared with a default value, the value of the attribute is whatever value appears as the attribute's value in an instance document; if the attribute does not appear in the instance document, the schema processor provides the attribute with a value equal to that of the default attribute.

Note: Default values for attributes only make sense if the attributes themselves are optional, and so it is an error to specify both a default value and anything other than a value of optional for use.

Default Elements

The schema processor treats defaulted elements slightly differently. When an element is declared with a default value, the value of the element is whatever value appears as the element's content in the instance document.

If the element appears without any content, the schema processor provides the element with a value equal to that of the default attribute. However, if the element does not appear in the instance document, the schema processor does not provide the element at all.

In summary, the differences between element and attribute defaults can be stated as:

- Default attribute values apply when attributes are missing
- Default element values apply when elements are empty

The fixed attribute is used in both attribute and element declarations to ensure that the attributes and elements are set to particular values. For example, `po.xsd` contains a declaration for the `country` attribute, which is declared with a fixed value `US`. This declaration means that the appearance of a `country` attribute in an instance document is optional (the default value of use is optional), although if the attribute does appear, its value must be `US`, and if the attribute does not appear, the schema processor will provide a `country` attribute with the value `US`.

Note: The concepts of a fixed value and a default value are mutually exclusive, and so it is an error for a declaration to contain both fixed and default attributes.

[Table B-1](#) summarizes the attribute values used in element and attribute declarations to constrain their occurrences.

Table B-1 Occurrence Constraints for XML Schema Elements and Attributes

Elements (minOccurs, maxOccurs) fixed, default	Attributes use, fixed, default	Notes
(1, 1) -, -	required, -, -	element/attribute must appear once, it may have any value
(1, 1) 37, -	required, 37, -	element/attribute must appear once, its value must be 37
(2, unbounded) 37, -	n/a	element must appear twice or more, its value must be 37; in general, minOccurs and maxOccurs values may be positive integers, and maxOccurs value may also be “unbounded”
(0, 1) -, -	optional, -, -	element/attribute may appear once, it may have any value
(0, 1) 37, -	optional, 37, -	element/attribute may appear once, if it does appear its value must be 37, if it does not appear its value is 37
(0, 1) -, 37	optional, -, 37	element/attribute may appear once; if it does not appear its value is 37, otherwise its value is that given
(0, 2) -, 37	n/a	element may appear once, twice, or not at all; if the element does not appear it is not provided; if it does appear and it is empty, its value is 37; otherwise its value is that given; in general, minOccurs and maxOccurs values may be positive integers, and maxOccurs value may also be “unbounded”
(0, 0) -, -	prohibited, -, -	element/attribute must not appear

Note: Neither `minOccurs`, `maxOccurs`, nor `use` may appear in the declarations of global elements and attributes.

Global Elements and Attributes

Global elements, and global attributes, are created by declarations that appear as the children of the schema element. Once declared, a global element or a global attribute can be referenced in one or more declarations using the `ref` attribute as described in the preceding section.

A declaration that references a global element enables the referenced element to appear in the instance document in the context of the referencing declaration. So, for example, the `comment` element appears in `po.xml` at the same level as the `shipTo`, `billTo` and `items` elements because the declaration that references `comment` appears in the complex type definition at the same level as the declarations of the other three elements.

The declaration of a global element also enables the element to appear at the top-level of an instance document. Hence `purchaseOrder`, which is declared as a global element in `po.xsd`, can appear as the top-level element in `po.xml`.

Note: This rationale also allows a comment element to appear as the top-level element in a document like `po.xml`.

Caveats: One caveat is that global declarations cannot contain references; global declarations must identify simple and complex types directly. Global declarations cannot contain the `ref` attribute, they must use the `type` attribute, or, be followed by an `anonymous` type definition.

A second caveat is that cardinality constraints cannot be placed on global declarations, although they can be placed on local declarations that reference global declarations. In other words, global declarations cannot contain the attributes `minOccurs`, `maxOccurs`, or `use`.

Naming Conflicts

The preceding section described how to:

- Define new complex types, such as, `PurchaseOrderType`
- Declare elements, such as, `purchaseOrder`
- Declare attributes, such as, `orderDate`

These involve naming. If two things are given the same name, in general, the more similar the two things are, the more likely there will be a naming conflict.

For example:

If the two things are both types, say a complex type called `USStates` and a simple type called `USStates`, there is a conflict.

If the two things are a type and an element or attribute, such as when defining a complex type called `USAddress` and declaring an element called `USAddress`, there is no conflict.

If the two things are elements within different types, that is, not global elements, say declare one element called `name` as part of the `USAddress` type and a second element called `name` as part of the `Item` type, there is no conflict. Such elements are sometimes called local element declarations.

If the two things are both types and you define one and XML Schema has defined the other, say you define a simple type called `decimal`, there is no conflict. The reason for the apparent contradiction in the last example is that the two types belong to different namespaces. Namespaces are described in ["Introducing the W3C Namespaces in XML Recommendation"](#) on page C-18.

Simple Types

The purchase order schema declares several elements and attributes that have simple types. Some of these simple types, such as `string` and `decimal`, are built into XML Schema, while others are derived from the built-in's.

For example, the `partNum` attribute has a type called `SKU` (Stock Keeping Unit) that is derived from `string`. Both built-in simple types and their derivations can be used in all element and attribute declarations. [Table B-2](#) lists all the simple types built into XML Schema, along with examples of the different types.

Table B-2 Simple Types Built into XML Schema

Simple Type	Examples (delimited by commas)	Notes
<code>string</code>	Confirm this is electric	--
<code>normalizedString</code>	Confirm this is electric	3
<code>token</code>	Confirm this is electric	4
<code>byte</code>	-1, 126	2
<code>unsignedByte</code>	0, 126	2
<code>base64Binary</code>	GpM7	--
<code>hexBinary</code>	0FB7	--
<code>integer</code>	-126789, -1, 0, 1, 126789	2
<code>positiveInteger</code>	1, 126789	2
<code>negativeInteger</code>	-126789, -1	2
<code>nonNegativeInteger</code>	0, 1, 126789	2
<code>nonPositiveInteger</code>	-126789, -1, 0	2
<code>int</code>	-1, 126789675	2
<code>unsignedInt</code>	0, 1267896754	2
<code>long</code>	-1, 12678967543233	2

Table B-2 Simple Types Built into XML Schema (Cont.)

Simple Type	Examples (delimited by commas)	Notes
unsignedLong	0, 12678967543233	2
short	-1, 12678	2
unsignedShort	0, 12678	2
decimal	-1.23, 0, 123.4, 1000.00	2
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	equivalent to single-precision 32-bit floating point, NaN is "not a number". Note: 2
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	equivalent to double-precision 64-bit floating point. Note: 2
boolean	true, false 1, 0	--
time	13:20:00.000, 13:20:00.000-05:00	2
dateTime	1999-05-31T13:20:00.000-05:00	May 31st 1999 at 1.20pm Eastern Standard Time which is 5 hours behind Co-Ordinated Universal Time, see 2
duration	P1Y2M3DT10H30M12.3S	1 year, 2 months, 3 days, 10 hours, 30 minutes, and 12.3 seconds
date	1999-05-31	2
gMonth	--05--	May, Notes: 2, 5
gYear	1999	1999, Notes: 2, 5
gYearMonth	1999-02	the month of February 1999, regardless of the number of days. Notes: 2, 5
gDay	---31	the 31st day. Notes: 2, 5
gMonthDay	--05-31	every May 31st. Notes: 2, 5
Name	shipTo	XML 1.0 Name type
QName	po:USAddress	XML Namespace QName

Table B–2 Simple Types Built into XML Schema (Cont.)

Simple Type	Examples (delimited by commas)	Notes
NCName	USAddress	XML Namespace NCName, that is, QName without the prefix and colon
anyURI	http://www.example.com/, http://www.example.com/doc.html#ID5	--
language	en-GB, en-US, fr	valid values for xml:lang as defined in XML 1.0
ID	--	XML 1.0 ID attribute type. Note: 1
IDREF	--	XML 1.0 IDREF attribute type. Note: 1
IDREFS	--	XML 1.0 IDREFS attribute type, see (1)
ENTITY	--	XML 1.0 ENTITY attribute type. Note: 1
ENTITIES	--	XML 1.0 ENTITIES attribute type. Note: 1
NOTATION	--	XML 1.0 NOTATION attribute type. Note: 1
NMTOKEN	US,BrÃ©sil	XML 1.0 NMTOKEN attribute type. Note: 1
NMTOKENS	US UK,BrÃ©sil Canada Mexique	XML 1.0 NMTOKENS attribute type, that is, a whitespace separated list of NMTOKEN's. Note: 1

Notes:

(1) To retain compatibility between XML Schema and XML 1.0 DTDs, the simple types `ID`, `IDREF`, `IDREFS`, `ENTITY`, `ENTITIES`, `NOTATION`, `NMTOKEN`, `NMTOKENS` should only be used in attributes.

(2) A value of this type can be represented by more than one lexical format. For example, `100` and `1.0E2` are both valid float formats representing “one hundred”.

However, rules have been established for this type that define a canonical lexical format, see XML Schema Part 2.

(3) Newline, tab and carriage-return characters in a `normalizedString` type are converted to space characters before schema processing.

(4) As `normalizedString`, and adjacent space characters are collapsed to a single space character, and leading and trailing spaces are removed.

(5) The “g” prefix signals time periods in the Gregorian calendar.

New simple types are defined by deriving them from existing simple types (built-in's and derived). In particular, you can derive a new simple type by restricting an existing simple type, in other words, the legal range of values for the new type are a subset of the existing type's range of values.

Use the `simpleType` element to define and name the new simple type. Use the `restriction` element to indicate the existing (base) type, and to identify the “facets” that constrain the range of values. A complete list of facets is provided in Appendix B of XML Schema Primer, <http://www.w3.org/TR/xmlschema-0/>.

Suppose you want to create a new type of integer called `myInteger` whose range of values is between 10000 and 99999 (inclusive). Base your definition on the built-in simple type `integer`, whose range of values also includes integers less than 10000 and greater than 99999.

To define `myInteger`, restrict the range of the `integer` base type by employing two *facets* called `minInclusive` and `maxInclusive`:

Defining `myInteger`, Range 10000-99999

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

The example shows one particular combination of a base type and two facets used to define `myInteger`, but a look at the list of built-in simple types and their facets should suggest other viable combinations.

The purchase order schema contains another, more elaborate, example of a simple type definition. A new simple type called `SKU` is derived (by restriction) from the simple type `string`. Furthermore, you can constrain the values of `SKU` using a facet called `pattern` in conjunction with the regular expression `"\d{3}-[A-Z]{2}"`

that is read “three digits followed by a hyphen followed by two upper-case ASCII letters”:

Defining the Simple Type “SKU”

```
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

This regular expression language is described more fully in Appendix D of <http://www.w3.org/TR/xmlschema-0/>.

XML Schema defines fifteen facets which are listed in Appendix B of <http://www.w3.org/TR/xmlschema-0/>. Among these, the enumeration facet is particularly useful and it can be used to constrain the values of almost every simple type, except the boolean type. The enumeration facet limits a simple type to a set of distinct values. For example, you can use the enumeration facet to define a new simple type called `USState`, derived from string, whose value must be one of the standard US state abbreviations:

Using the Enumeration Facet

```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

`USState` would be a good replacement for the string type currently used in the state element declaration. By making this replacement, the legal values of a state element, that is, the state subelements of `billTo` and `shipTo`, would be limited to one of AK, AL, AR, and so on. Note that the enumeration values specified for a particular type must be unique.

List Types

XML Schema has the concept of a list type, in addition to the so-called atomic types that constitute most of the types listed in [Table B-3](#). Atomic types, list types, and the union types described in the next section are collectively called simple types. The

value of an atomic type is indivisible from XML Schema's perspective. For example, the `NMTOKEN` value `US` is indivisible in the sense that no part of `US`, such as the character `"S"`, has any meaning by itself. In contrast, list types are comprised of sequences of atomic types and consequently the parts of a sequence (the "atoms") themselves are meaningful. For example, `NMTOKENS` is a list type, and an element of this type would be a white-space delimited list of `NMTOKEN`'s, such as `"US UK FR"`. XML Schema has three built-in list types:

- `NMTOKENS`
- `IDREFS`
- `ENTITIES`

In addition to using the built-in list types, you can create new list types by derivation from existing atomic types. You cannot create list types from existing list types, nor from complex types. For example, to create a list of `myInteger`'s:

Creating a List of `myInteger`'s

```
<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

And an element in an instance document whose content conforms to `listOfMyIntType` is:

```
<listOfMyInt>20003 15037 95977 95945</listOfMyInt>
```

Several facets can be applied to list types: `length`, `minLength`, `maxLength`, and `enumeration`. For example, to define a list of exactly six US states (`SixUSStates`), we first define a new list type called `USStateList` from `USState`, and then we derive `SixUSStates` by restricting `USStateList` to only six items:

List Type for Six US States

```
<xsd:simpleType name="USStateList">
  <xsd:list itemType="USState"/>
</xsd:simpleType>
<xsd:simpleType name="SixUSStates">
  <xsd:restriction base="USStateList">
    <xsd:length value="6"/>
  </xsd:restriction>
</xsd:simpleType>
```

Elements whose type is `SixUSStates` must have six items, and each of the six items must be one of the (atomic) values of the enumerated type `USState`, for example:

```
<sixStates>PA NY CA NY LA AK</sixStates>
```

Note that it is possible to derive a list type from the atomic type string. However, a string may contain white space, and white space delimits the items in a list type, so you should be careful using list types whose base type is string. For example, suppose we have defined a list type with a length facet equal to 3, and base type string, then the following 3 item list is legal:

```
Asie Europe Afrique
```

But the following 3 “item” list is illegal:

```
Asie Europe AmÃ©rique Latine
```

Even though "AmÃ©rique Latine" may exist as a single string outside of the list, when it is included in the list, the whitespace between AmÃ©rique and Latine effectively creates a fourth item, and so the latter example will not conform to the 3-item list type.

Union Types

Atomic types and list types enable an element or an attribute value to be one or more instances of one atomic type. In contrast, a union type enables an element or attribute value to be one or more instances of one type drawn from the union of multiple atomic and list types. To illustrate, we create a union type for representing American states as singleton letter abbreviations or lists of numeric codes. The `zipUnion` union type is built from one atomic type and one list type:

Union Type for Zipcodes

```
<xsd:simpleType name="zipUnion">
  <xsd:union memberTypes="USState listOfMyIntType"/>
</xsd:simpleType>
```

When we define a union type, the `memberTypes` attribute value is a list of all the types in the union.

Now, assuming we have declared an element called `zips` of type `zipUnion`, valid instances of the element are:

```
<zips>CA</zips>
<zips>95630 95977 95945</zips>
<zips>AK</zips>
```

Two facets, `pattern` and `enumeration`, can be applied to a union type.

Anonymous Type Definitions

Schemas can be constructed by defining sets of named types such as `PurchaseOrderType` and then declaring elements such as `purchaseOrder` that reference the types using the `type=` construction. This style of schema construction is straightforward but it can be unwieldy, especially if you define many types that are referenced only once and contain very few constraints. In these cases, a type can be more succinctly defined as an anonymous type which saves the overhead of having to be named and explicitly referenced.

The definition of the type `Items` in `po.xsd` contains two element declarations that use anonymous types (`item` and `quantity`). In general, you can identify anonymous types by the lack of a `type=` in an element (or attribute) declaration, and by the presence of an un-named (simple or complex) type definition:

Two Anonymous Type Definitions

```
<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice" type="xsd:decimal"/>
          <xsd:element ref="comment" minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

In the case of the `item` element, it has an anonymous complex type consisting of the elements `productName`, `quantity`, `USPrice`, `comment`, and `shipDate`, and an attribute called `partNum`. In the case of the `quantity` element, it has an anonymous simple type derived from integer whose value ranges between 1 and 99.

Element Content

The purchase order schema has many examples of elements containing other elements (for example, `items`), elements having attributes and containing other elements (such as, `shipTo`), and elements containing only a simple type of value (for example, `USPrice`). However, we have not seen an element having attributes but containing only a simple type of value, nor have we seen an element that contains other elements mixed with character content, nor have we seen an element that has no content at all. In this section we'll examine these variations in the content models of elements.

Complex Types from Simple Types

Let us first consider how to declare an element that has an attribute and contains a simple value. In an instance document, such an element might appear as:

```
<internationalPrice currency="EUR">423.46</internationalPrice>
```

The purchase order schema declares a `USPrice` element that is a starting point:

```
<xsd:element name="USPrice" type="decimal"/>
```

Now, how do we add an attribute to this element? As we have said before, simple types cannot have attributes, and `decimal` is a simple type.

Therefore, we must define a complex type to carry the attribute declaration. We also want the content to be simple type `decimal`. So our original question becomes: How do we define a complex type that is based on the simple type `decimal`? The answer is to derive a new complex type from the simple type `decimal`:

Deriving a Complex Type from a Simple Type

```
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="currency" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

```
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

We use the `complexType` element to start the definition of a new (anonymous) type. To indicate that the content model of the new type contains only character data and no elements, we use a `simpleContent` element. Finally, we derive the new type by extending the simple decimal type. The extension consists of adding a currency attribute using a standard attribute declaration. (We cover type derivation in detail in Section 4.) The `internationalPrice` element declared in this way will appear in an instance as shown in the example at the beginning of this section.

Mixed Content

The construction of the purchase order schema may be characterized as elements containing subelements, and the deepest subelements contain character data. XML Schema also provides for the construction of schemas where character data can appear alongside subelements, and character data is not confined to the deepest subelements.

To illustrate, consider the following snippet from a customer letter that uses some of the same elements as the purchase order:

Snippet of Customer Letter

```
<letterBody>
    <salutation>Dear Mr.<name>Robert Smith</name>.</salutation>
    Your order of <quantity>1</quantity> <productName>Baby
    Monitor</productName> shipped from our warehouse on
    <shipDate>1999-05-21</shipDate>. . . .
</letterBody>
```

Notice the text appearing between elements and their child elements. Specifically, text appears between the elements `salutation`, `quantity`, `productName` and `shipDate` which are all children of `letterBody`, and text appears around the element name which is the child of a child of `letterBody`. The following snippet of a schema declares `letterBody`:

Snippet of Schema for Customer Letter

```
<xsd:element name="letterBody">
    <xsd:complexType mixed="true">
        <xsd:sequence>
            <xsd:element name="salutation">
```

```

        <xsd:complexType mixed="true">
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="quantity" type="xsd:positiveInteger"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
      <!-- and so on -->
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

The elements appearing in the customer letter are declared, and their types are defined using the element and complexType element constructions we previously seen. To enable character data to appear between the child-elements of letterBody, the mixed attribute on the type definition is set to true.

Note that the mixed model in XML Schema differs fundamentally from the mixed model in XML 1.0. Under the XML Schema mixed model, the order and number of child elements appearing in an instance must agree with the order and number of child elements specified in the model. In contrast, under the XML 1.0 mixed model, the order and number of child elements appearing in an instance cannot be constrained. In summary, XML Schema provides full validation of mixed models in contrast to the partial schema validation provided by XML 1.0.

Empty Content

Now suppose that we want the internationalPrice element to convey both the unit of currency and the price as attribute values rather than as separate attribute and content values. For example:

```
<internationalPrice currency="EUR" value="423.46"/>
```

Such an element has no content at all; its content model is empty.

An Empty Complex Type

To define a type whose content is empty, we essentially define a type that allows only elements in its content, but we do not actually declare any elements and so the type's content model is empty:

```
<xsd:element name="internationalPrice">
  <xsd:complexType>

```

```
<xsd:complexContent>
  <xsd:restriction base="xsd:anyType">
    <xsd:attribute name="currency" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:decimal"/>
  </xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
</xsd:element>
```

In this example, we define an (anonymous) type having `complexContent`, that is, only elements. The `complexContent` element signals that the intent to restrict or extend the content model of a complex type, and the restriction of `anyType` declares two attributes but does not introduce any element content (see Section 4.4 of the XML Schema Primer, for more details on restriction). The `internationalPrice` element declared in this way may legitimately appear in an instance as shown in the preceding example.

Shorthand for an Empty Complex Type

The preceding syntax for an empty-content element is relatively verbose, and it is possible to declare the `internationalPrice` element more compactly:

```
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:attribute name="currency" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:decimal"/>
  </xsd:complexType>
</xsd:element>
```

This compact syntax works because a complex type defined without any `simpleContent` or `complexContent` is interpreted as shorthand for complex content that restricts `anyType`.

AnyType

The `anyType` represents an abstraction called the `ur-type` which is the base type from which all simple and complex types are derived. An `anyType` type does not constrain its content in any way. It is possible to use `anyType` like other types, for example:

```
<xsd:element name="anything" type="xsd:anyType"/>
```

The content of the element declared in this way is unconstrained, so the element value may be 423.46, but it may be any other sequence of characters as well, or

indeed a mixture of characters and elements. In fact, `anyType` is the default type when none is specified, so the preceding could also be written as follows:

```
<xsd:element name="anything"/>
```

If unconstrained element content is needed, for example in the case of elements containing prose which requires embedded markup to support internationalization, then the default declaration or a slightly restricted form of it may be suitable. The text type described in Section 5.5 is an example of such a type that is suitable for such purposes.

Annotations

XML Schema provides three elements for annotating schemas for the benefit of both human readers and applications. In the purchase order schema, we put a basic schema description and copyright information inside the `documentation` element, which is the recommended location for human readable material. We recommend you use the `xml:lang` attribute with any documentation elements to indicate the language of the information. Alternatively, you may indicate the language of all information in a schema by placing an `xml:lang` attribute on the schema element.

The `appInfo` element, which we did not use in the purchase order schema, can be used to provide information for tools, stylesheets and other applications. An interesting example using `appInfo` is a schema that describes the simple types in XML Schema Part 2: Datatypes.

Information describing this schema, for example, which facets are applicable to particular simple types, is represented inside `appInfo` elements, and this information was used by an application to automatically generate text for the XML Schema Part 2 document.

Both `documentation` and `appInfo` appear as subelements of annotation, which may itself appear at the beginning of most schema constructions. To illustrate, the following example shows annotation elements appearing at the beginning of an element declaration and a complex type definition:

Annotations in Element Declaration & Complex Type Definition

```
<xsd:element name="internationalPrice">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      element declared with anonymous type
    </xsd:documentation>
  </xsd:annotation>
```

```
<xsd:complexType>
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      empty anonymous type with 2 attributes
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:restriction base="xsd:anyType">
      <xsd:attribute name="currency" type="xsd:string"/>
      <xsd:attribute name="value" type="xsd:decimal"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
</xsd:element>
```

The annotation element may also appear at the beginning of other schema constructions such as those indicated by the elements `schema`, `simpleType`, and `attribute`.

Building Content Models

The definitions of complex types in the purchase order schema all declare sequences of elements that must appear in the instance document. The occurrence of individual elements declared in the so-called content models of these types may be optional, as indicated by a 0 value for the attribute `minOccurs` (for example, in comment), or be otherwise constrained depending upon the values of `minOccurs` and `maxOccurs`.

XML Schema also provides constraints that apply to groups of elements appearing in a content model. These constraints mirror those available in XML 1.0 plus some additional constraints. Note that the constraints do not apply to attributes.

XML Schema enables groups of elements to be defined and named, so that the elements can be used to build up the content models of complex types (thus mimicking common usage of parameter entities in XML 1.0). Un-named groups of elements can also be defined, and along with elements in named groups, they can be constrained to appear in the same order (sequence) as they are declared. Alternatively, they can be constrained so that only one of the elements may appear in an instance.

To illustrate, we introduce two groups into the `PurchaseOrderType` definition from the purchase order schema so that purchase orders may contain either separate shipping and billing addresses, or a single address for those cases in which the shippee and billee are co-located:

Nested Choice and Sequence Groups

```

<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:group ref="shipAndBill"/>
      <xsd:element name="singleUSAddress" type="USAddress"/>
    </xsd:choice>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:group name="shipAndBill">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:group>

```

The choice group element allows only one of its children to appear in an instance. One child is an inner group element that references the named group `shipAndBill` consisting of the element sequence `shipTo`, `billTo`, and the second child is `singleUSAddress`. Hence, in an instance document, the `purchaseOrder` element must contain either a `shipTo` element followed by a `billTo` element or a `singleUSAddress` element. The choice group is followed by the `comment` and `items` element declarations, and both the choice group and the element declarations are children of a sequence group. The effect of these various groups is that the address element(s) must be followed by `comment` and `items` elements in that order.

There exists a third option for constraining elements in a group: All the elements in the group may appear once or not at all, and they may appear in any order. The `all` group (which provides a simplified version of the SGML `&-Connector`) is limited to the top-level of any content model.

Moreover, the group's children must all be individual elements (no groups), and no element in the content model may appear more than once, that is, the permissible values of `minOccurs` and `maxOccurs` are 0 and 1.

For example, to allow the child elements of `purchaseOrder` to appear in any order, we could redefine `PurchaseOrderType` as:

An 'All' Group

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:all>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:all>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

By this definition, a comment element may optionally appear within purchaseOrder, and it may appear before or after any shipTo, billTo and items elements, but it can appear only once. Moreover, the stipulations of an all group do not allow us to declare an element such as comment outside the group as a means of enabling it to appear more than once. XML Schema stipulates that an all group must appear as the sole child at the top of a content model. In other words, the following is illegal:

Illegal Example with an 'All' Group

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:all>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element name="items" type="Items"/>
    </xsd:all>
  <xsd:sequence>
    <xsd:element ref="comment" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

Finally, named and un-named groups that appear in content models (represented by group and choice, sequence, all respectively) may carry minOccurs and maxOccurs attributes. By combining and nesting the various groups provided by XML Schema, and by setting the values of minOccurs and maxOccurs, it is possible to represent any content model expressible with an XML 1.0 DTD. Furthermore, the all group provides additional expressive power.

Attribute Groups

To provide more information about each item in a purchase order, for example, each item's weight and preferred shipping method, you can add `weightKg` and `shipBy` attribute declarations to the item element's (anonymous) type definition:

Adding Attributes to the Inline Type Definition

```
<xsd:element name="Item" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity">
        <xsd:simpleType>
          <xsd:restriction base="xsd:positiveInteger">
            <xsd:maxExclusive value="100"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="USPrice" type="xsd:decimal"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="partNum" type="SKU" use="required"/>
    <!-- add weightKg and shipBy attributes -->
    <xsd:attribute name="weightKg" type="xsd:decimal"/>
    <xsd:attribute name="shipBy">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="air"/>
          <xsd:enumeration value="land"/>
          <xsd:enumeration value="any"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

Alternatively, you can create a named attribute group containing all the desired attributes of an item element, and reference this group by name in the item element declaration:

Adding Attributes Using an Attribute Group

```
<xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
```

```
<xsd:complexType>
  <xsd:sequence>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:element name="quantity">
      <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
          <xsd:maxExclusive value="100"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="USPrice" type="xsd:decimal"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
  </xsd:sequence>

  <!-- attributeGroup replaces individual declarations -->
  <xsd:attributeGroup ref="ItemDelivery"/>
</xsd:complexType>
</xsd:element>

<xsd:attributeGroup name="ItemDelivery">
  <xsd:attribute name="partNum" type="SKU" use="required"/>
  <xsd:attribute name="weightKg" type="xsd:decimal"/>
  <xsd:attribute name="shipBy">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="air"/>
        <xsd:enumeration value="land"/>
        <xsd:enumeration value="any"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:attributeGroup>
```

Using an attribute group in this way can improve the readability of schemas, and facilitates updating schemas because an attribute group can be defined and edited in one place and referenced in multiple definitions and declarations. These characteristics of attribute groups make them similar to parameter entities in XML 1.0. Note that an attribute group may contain other attribute groups. Note also that both attribute declarations and attribute group references must appear at the end of complex type definitions.

Nil Values

One of the purchase order items listed in `po.xml`, the `Lawnmower`, does not have a `shipDate` element. Within the context of our scenario, the schema author may have intended such absences to indicate items not yet shipped. But in general, the absence of an element does not have any particular meaning: It may indicate that the information is unknown, or not applicable, or the element may be absent for some other reason. Sometimes it is desirable to represent an unshipped item, unknown information, or inapplicable information explicitly with an element, rather than by an absent element.

For example, it may be desirable to represent a “null” value being sent to or from a relational database with an element that is present. Such cases can be represented using XML Schema's nil mechanism which enables an element to appear with or without a non-nil value.

XML Schema's nil mechanism involves an “out of band” nil signal. In other words, there is no actual nil value that appears as element content, instead there is an attribute to indicate that the element content is nil. To illustrate, we modify the `shipDate` element declaration so that nils can be signalled:

```
<xsd:element name="shipDate" type="xsd:date" nillable="true"/>
```

And to explicitly represent that `shipDate` has a nil value in the instance document, we set the nil attribute (from the XML Schema namespace for instances) to true:

```
<shipDate xsi:nil="true"></shipDate>
```

The nil attribute is defined as part of the XML Schema namespace for instances, <http://www.w3.org/2001/XMLSchema-instance>, and so it must appear in the instance document with a prefix (such as `xsi:`) associated with that namespace. (As with the `xsd:` prefix, the `xsi:` prefix is used by convention only.) Note that the nil mechanism applies only to element values, and not to attribute values. An element with `xsi:nil="true"` may not have any element content but it may still carry attributes.

How DTDs and XML Schema Differ

DTD is a mechanism provided by XML 1.0 for declaring constraints on XML markup. DTDs enable you to specify the following:

- Elements that can appear in your XML documents
- Elements (or sub-elements) that can be in the elements

- The order in which the elements can appear

The XML Schema language serves a similar purpose to DTDs, but it is more flexible in specifying XML document constraints and potentially more useful for certain applications.

XML Example

Consider the XML document:

```
<?xml version="1.0">
<publisher pubid="ab1234">
  <publish-year>2000</publish-year>
  <title>The Cat in the Hat</title>
  <author>Dr. Seuss</author>
  <artist>Ms. Seuss</artist>
  <isbn>123456781111</isbn>
</publisher>
```

DTD Example

Consider a typical DTD for the foregoing XML document:

```
<!ELEMENT publisher (year,title, author+, artist?, isbn)>
<!ELEMENT publisher (year,title, author+, artist?, isbn)>
<!ELEMENT publish-year (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT artist (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
...
```

XML Schema Example

The XML schema definition equivalent to the preceding DTD example is:

```
<?xml version="1.0" encoding="UTF-8"?><schema
xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="publisher"> <complexType>
    <sequence>
      <element name="publish-year" type="short"/>
      <element name="title" type="string"/>
      <element name="author" type="string" maxOccurs="unbounded"/>
      <element name="artist" type="string" nillable="true" minOccurs="0"/>
      <element name="isbn" type="long"/>
    </sequence>
    <attribute name="pubid" type="hexBinary" use="required"/>
  </complexType>
</element>
```

```
</complexType>  
</element></schema>
```

DTD Limitations

DTDs, also known as XML Markup Declarations, are considered deficient in handling certain applications which include the following:

- Document authoring and publishing
- Exchange of metadata
- E-commerce
- Inter-database operations

DTD limitations include:

- No integration with Namespace technology, meaning that users cannot import and reuse code.
- No support of datatypes other than character data, a limitation for describing metadata standards and database schemas.
- Applications need to specify document structure constraints more flexibly than the DTD allows for.

XML Schema Features Compared to DTD Features

[Table B-3](#) lists XML Schema features. Note that XML Schema features include DTD features.

Table B-3 XML Schema Features Compared to DTD Features

XML Schema Feature	DTD Features
Built-In Datatypes	
XML schemas specify a set of built-in datatypes. Some of them are defined and called primitive datatypes, and they form the basis of the type system: string, boolean, float, decimal, double, duration, dateTime, time, date, gYearMonth, gYear, gMonthDat, gMonth, gDay, Base64Binary, HexBinary, anyURI, NOTATION, QName	DTDs do not support datatypes other than character strings.
Others are derived datatypes that are defined in terms of primitive types.	
User-Defined Datatypes	
Users can derive their own datatypes from the built-in datatypes. There are three ways of datatype derivation: restriction, list, and union. Restriction defines a more restricted datatype by applying constraining facets to the base type, list simply allows a list of values of its item type, and union defines a new type whose value can be of any of its member types.	The publish-year element in the DTD example cannot be constrained further.
For example, to specify that the value of publish-year type to be within a specific range:	--
<pre><element name="publish-year"> <simpleType> <restriction base="short" <minInclusive value="1970"/ <maxInclusive value="2000"/> </restriction> </simpleType> </element></pre>	
Constraining facets are: length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, totalDigits, fractionDigits	
Note that some facets only apply to certain base types.	

Table B-3 XML Schema Features Compared to DTD Features (Cont.)

XML Schema Feature	DTD Features
<p>Occurrence Indicators (Content Model or Structure)</p> <p>In XML Schema, the structure (called <code>complexType</code>) of an instance document or element is defined in terms of model and attribute groups. A model group may further contain model groups or element particles, while an attribute group contains attributes.</p> <p>Wildcards can be used in both model and attribute groups. There are three types of model group: sequence, all, and choice, representing the sequence, conjunction, and disjunction relationships among particles, respectively. The range of the number of occurrences of each particle can also be specified.</p>	--
<p>Like the datatype, <code>complexType</code> can be derived from other types. The derivation method can be either restriction or extension. The derived type inherits the content of the base type plus corresponding modifications. In addition to inheritance, a type definition can make references to other components. This feature allows a component to be defined once and used in many other structures.</p> <p>The type declaration and definition mechanism in XML Schema is much more flexible and powerful than in DTDs.</p>	--

Table B-3 XML Schema Features Compared to DTD Features (Cont.)

XML Schema Feature	DTD Features
minOccurs, maxOccurs	Control by DTDs over the number of child elements in an element are assigned with the following symbols: <ul style="list-style-type: none"> ■ ? = zero or one. In "DTD Example" on page B-32, artist? implied that artist is optional. ■ * = zero or more. ■ + = one or more in the "DTD Example" on page B-32, author+ implies that more than one author is possible. ■ (none) = exactly one.
Identity Constraints	None.
XML Schema extends the concept of the XML ID/IDREF mechanism with the declarations of unique, key and keyref. They are part of the type definition and allow not only attributes, but also element content as keys. Each constraint has a scope. Constraint comparison is in terms of their value rather than lexical strings.	
Import/Export Mechanisms (Schema Import, Inclusion and Modification)	
All components of a schema need not be defined in a single schema file. XML Schema provides a mechanism for assembling multiple XML schemas. Import is used to integrate XML schemas that use different namespaces, while inclusion is used to add components that have the same namespace. When components are included, they can be modified using redefinition.	You cannot use constructs defined in external schemas.

XML schema can be used to define a class of XML documents.

Instance XML Documents

An *instance XML document* describes an XML document that conforms to a particular XML schema. Although these instances and XML schemas need not exist specifically as *documents*, they are commonly referred to as *files*. They may however exist as any of the following:

- Streams of bytes
- Fields in a database record

- Collections of XML Infoset *information items*

Oracle XML DB supports the W3C XML Schema Recommendation specifications of May 2, 2001: <http://www.w3.org/2001/XMLSchema>

Converting Existing DTDs to XML Schema?

Some XML editors, such as XMLSpy, facilitate the conversion of existing DTDs to XML schemas, however you will still need to add further typing and validation declarations to the resulting XML schema definition file before it will be useful or can be used as an XML schema.

XML Schema Example, PurchaseOrder.xsd

The following example PurchaseOrder.xsd, is a W3C XML Schema example, in its native form, as an XML Document. PurchaseOrder.xsd XML schema is used for the examples described in [Chapter 3, "Using Oracle XML DB"](#):

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="ActionsType" >
    <xs:sequence>
      <xs:element name="Action" maxOccurs="4" >
        <xs:complexType >
          <xs:sequence >
            <xs:sequence>
              <xs:element ref="User"/>
              <xs:element ref="Date"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  <xs:complexType name="RejectType" >
    <xs:all>
      <xs:element ref="User" minOccurs="0"/>
      <xs:element ref="Date" minOccurs="0"/>
      <xs:element ref="Comments" minOccurs="0"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="ShippingInstructionsType" >
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="address"/>
      <xs:element ref="telephone"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

```

</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 ref="Description" />
    <xs:element ref="Part" />
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" />
</xs:complexType>
<!--
-->
<xs:element name="PurchaseOrder" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Reference" />
      <xs:element name="Actions"
        type="ActionsType" />
      <xs:element name="Reject"
        type="RejectType"
        minOccurs="0" />
      <xs:element ref="Requestor" />
      <xs:element ref="User" />
      <xs:element ref="CostCenter" />
      <xs:element name="ShippingInstructions"
        type="ShippingInstructionsType" />
      <xs:element ref="SpecialInstructions" />
      <xs:element name="LineItems"
        type="LineItemsType" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:simpleType name="money">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2" />
    <xs:totalDigits value="12" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantity">

```

```
<xs:restriction base="xs:decimal">
  <xs:fractionDigits value="4"/>
  <xs:totalDigits value="8"/>
</xs:restriction>
</xs:simpleType>
<xs:element name="User" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="10"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="Requestor" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="0"/>
      <xs:maxLength value="128"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="Reference" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="26"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="CostCenter" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="4"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="Vendor" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="0"/>
      <xs:maxLength value="20"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
<xs:element name="PONumber" >
  <xs:simpleType>
    <xs:restriction base="xs:integer"/>
  </xs:simpleType>
</xs:element>
<xs:element name="SpecialInstructions" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="0"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="name" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="20"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="address" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="telephone" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="24"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="Date" type="xs:date" />
<xs:element name="Comments" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
</xs:element>
<xs:element name="Description" >
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="Part" >
  <xs:complexType>
    <xs:attribute name="Id" >
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:minLength value="12"/>
          <xs:maxLength value="14"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="Quantity" type="money"/>
    <xs:attribute name="UnitPrice" type="quantity"/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XPath and Namespace Primer

This appendix describes introductory information about the W3C XPath Recommendation, Namespace Recommendation, and the Information Set (infoset). It contains the following sections:

- [Introducing the W3C XML Path Language \(XPath\) 1.0 Recommendation](#)
- [The XPath Expression](#)
- [Location Paths](#)
- [XPath 1.0 Data Model](#)
- [Introducing the W3C XML Path Language \(XPath\) 2.0 Working Draft](#)
- [Introducing the W3C Namespaces in XML Recommendation](#)
- [Introducing the W3C XML Information Set](#)

Introducing the W3C XML Path Language (XPath) 1.0 Recommendation

XML Path Language (XPath) is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer. It can be used as a searching or query language as well as in hypertext linking. Parts of this brief XPath primer are extracted from the W3C XPath Recommendation.

XPath also facilitates the manipulation of strings, numbers and booleans.

XPath uses a compact, non-XML syntax to facilitate use of XPath in URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. It gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.

In addition to its use for addressing, XPath is also designed so that it has a natural subset that can be used for matching, that is, testing whether or not a node matches a pattern. This use of XPath is described in the W3C XSLT Recommendation.

Note: In this release, Oracle XML DB supports a subset of the XPath 1.0 Recommendation. It does not support XPaths that return booleans, numbers, or strings. However, Oracle XML DB does support these XPath types within predicates.

XPath Models an XML Document as a Tree of Nodes

XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes, and text nodes. XPath defines a way to compute a string-value for each type of node. Some types of nodes also have names. XPath fully supports XML Namespaces. Thus, the name of a node is modeled as a pair consisting of a local part and a possibly null namespace URI; this is called an expanded-name. The data model is described in detail in "[XPath 1.0 Data Model](#)" on page C-10. A summary of XML Namespaces is provided in "[Introducing the W3C Namespaces in XML Recommendation](#)" on page C-18.

See Also:

- <http://www.w3.org/TR/xpath>
- <http://www.w3.org/TR/xpath20/>
- <http://www.zvon.org/xxl/XPathTutorial/General/examples.html>
- <http://www.mulberrytech.com/quickref/XSLTquickref.pdf>
- <http://www.oreilly.com/catalog/xmlnut/chapter/ch09.html>
- <http://www.w3.org/TR/2002/NOTE-unicode-xml-20020218/> for information about using unicode in XML.

The XPath Expression

The primary syntactic construct in XPath is the expression. An expression matches the production `Expr`. An expression is evaluated to yield an object, which has one of the following four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

Evaluating Expressions with Respect to a Context

Expression evaluation occurs with respect to a context. XSLT and XPointer specify how the context is determined for XPath expressions used in XSLT and XPointer respectively. The context consists of the following:

- **Node**, the context node
- **Pair of nonzero positive integers**, context position and context size. Context position is always less than or equal to the context size.
- **Set of variable bindings**. These consist of a mapping from variable names to variable values. The value of a variable is an object, which can be of any of the types possible for the value of an expression, can also be of additional types not specified here.

- **Function library.** This consists of a mapping from function names to functions. Each function takes zero or more arguments and returns a single result. See the XPath Recommendation for the core function library definition, that all XPath implementations must support. For a function in the core function library, arguments and result are of the four basic types:

- Node Set functions
- String Functions
- Boolean functions
- Number functions

Both XSLT and XPointer extend XPath by defining additional functions; some of these functions operate on the four basic types; others operate on additional data types defined by XSLT and XPointer.

- **Set of namespace declarations in scope for the expression.** These consist of a mapping from prefixes to namespace URIs.

Evaluating Subexpressions

The variable bindings, function library, and namespace declarations used to evaluate a *subexpression* are always the same as those used to evaluate the containing *expression*.

The context node, context position, and context size used to evaluate a subexpression are sometimes different from those used to evaluate the containing expression. Several kinds of expressions change the context node; only predicates change the context position and context size. When the evaluation of a kind of expression is described, it will always be explicitly stated if the context node, context position, and context size change for the evaluation of subexpressions; if nothing is said about the context node, context position, and context size, they remain unchanged for the evaluation of subexpressions of that kind of expression.

XPath Expressions Often Occur in XML Attributes

The grammar specified here applies to the attribute value after XML 1.0 normalization. So, for example, if the grammar uses the character `<`, this must not appear in the XML source as `<` but must be quoted according to XML 1.0 rules by, for example, entering it as `<`.

Within expressions, literal strings are delimited by single or double quotation marks, which are also used to delimit XML attributes. To avoid a quotation mark in

an expression being interpreted by the XML processor as terminating the attribute value:

- The quotation mark can be entered as a character reference (`"` ; or `'` ;)
- The expression can use single quotation marks if the XML attribute is delimited with double quotation marks or vice-versa

Location Paths

One important kind of expression is a location path. A location path is the 'route' to be taken. The route can consist of directions and several steps, each step being separated by a `'/'`.

A location path selects a set of nodes relative to the context node. The result of evaluating an expression that is a location path is the node-set containing the nodes selected by the location path.

Location paths can recursively contain expressions used to filter sets of nodes. A location path matches the production `LocationPath`.

Expressions are parsed by first dividing the character string to be parsed into tokens and then parsing the resulting sequence of tokens. Whitespace can be freely used between tokens.

Although location paths are not the most general grammatical construct in the XPath language (a `LocationPath` is a special case of an `Expr`), they are the most important construct.

Location Path Syntax Abbreviations

Every location path can be expressed using a straightforward but rather verbose syntax. There are also a number of syntactic abbreviations that allow common cases to be expressed concisely. The next sections:

- "[Location Path Examples Using Unabbreviated Syntax](#)" on page C-5 describes the semantics of location paths using the unabbreviated syntax
- "[Location Path Examples Using Abbreviated Syntax](#)" on page C-7 describes the unabbreviated syntax

Location Path Examples Using Unabbreviated Syntax

[Table C-1](#) lists examples of location paths using the unabbreviated syntax.

Table C-1 XPath: Location Path Examples Using Unabbreviated Syntax

Unabbreviated Location Path	Description
child::para	Selects the para element children of the context node
child::*	Selects all element children of the context node
child::text()	Selects all text node children of the context node
child::node()	Selects all the children of the context node, whatever their node type
attribute::name	Selects the name attribute of the context node
attribute::*	Selects all the attributes of the context
nodedescendant::para	Selects the para element descendants of the context node
ancestor::div	Selects all div ancestors of the context node
ancestor-or-self::div	Selects the div ancestors of the context node and, if the context node is a div element, the context node as well
descendant-or-self::para	Selects the para element descendants of the context node and, if the context node is a para element, the context node as well
self::para	Selects the context node if it is a para element, and otherwise selects nothing
child::chapter/descendant::para	Selects the para element descendants of the chapter element children of the context node
child::*/*/child::para	Selects all para grandchildren of the context node
/	Selects the document root which is always the parent of the document element
/descendant::para	Selects all the para elements in the same document as the context node
/descendant::olist/child::item	Selects all the item elements that have an olist parent and that are in the same document as the context node
child::para[position()=1]	Selects the first para child of the context node
child::para[position()=last()]	Selects the last para child of the context node
child::para[position()=last()-1]	Selects the last but one para child of the context node
child::para[position()>1]	Selects all the para children of the context node other than the first para child of the context node
following-sibling::chapter[position()=1]	Selects the next chapter sibling of the context node
preceding-sibling::chapter[position()=1]	Selects the previous chapter sibling of the context node

Table C–1 XPath: Location Path Examples Using Unabbreviated Syntax (Cont.)

Unabbreviated Location Path	Description
/descendant::figure[position()=42]	Selects the forty-second figure element in the document
/child::doc/child::chapter[position()=5]/child::section[position()=2]	Selects the second section of the fifth chapter of the doc document element
child::para[attribute::type="warning"]	Selects all para children of the context node that have a type attribute with value warning
child::para[attribute::type='warning'][position()=5]	Selects the fifth para child of the context node that has a type attribute with value warning
child::para[position()=5][attribute::type="warning"]	Selects the fifth para child of the context node if that child has a type attribute with value warning
child::chapter[child::title='Introduction']	Selects the chapter children of the context node that have one or more title children with string-value equal to Introduction
child::chapter[child::title]	Selects the chapter children of the context node that have one or more title children
child::*[self::chapter or self::appendix]	Selects the chapter and appendix children of the context node
child::*[self::chapter or self::appendix][position()=last()]	Selects the last chapter or appendix child of the context node

Location Path Examples Using Abbreviated Syntax

Table C–2 lists examples of location paths using abbreviated syntax.

Table C–2 XPath: Location Path Examples Using Abbreviated Syntax

Abbreviated Location Path	Description
para	Selects the para element children of the context node
*	Selects all element children of the context node
text()	Selects all text node children of the context node
@name	Selects the name attribute of the context node
@*	Selects all the attributes of the context node
para[1]	Selects the first para child of the context node
para[last()]	Selects the last para child of the context node
*/para	Selects all para grandchildren of the context node
/doc/chapter[5]/section[2]	Selects the second section of the fifth chapter of the doc

Table C-2 XPath: Location Path Examples Using Abbreviated Syntax (Cont.)

Abbreviated Location Path	Description
chapter//para	Selects the para element descendants of the chapter element children of the context node
//para	Selects all the para descendants of the document root and thus selects all para elements in the same document as the context node
//olist/item	Selects all the item elements in the same document as the context node that have an olist parent
.	Selects the context node
./para	Selects the para element descendants of the context node
..	Selects the parent of the context node
../@lang	Selects the lang attribute of the parent of the context node
para[@type="warning"]	Selects all para children of the context node that have a type attribute with value warning
para[@type="warning"][5]	Selects the fifth para child of the context node that has a type attribute with value warning
para[5][@type="warning"]	Selects the fifth para child of the context node if that child has a type attribute with value warning
chapter[title="Introduction"]	Selects the chapter children of the context node that have one or more title children with string-value equal to Introduction
chapter[title]	Selects the chapter children of the context node that have one or more title children
employee[@secretary and @assistant]	Selects all the employee children of the context node that have both a secretary attribute and an assistant attribute

The most important abbreviation is that `child::` can be omitted from a location step. In effect, `child` is the default axis. For example, a location path `div/para` is short for `child::div/child::para`.

Attribute Abbreviation @

There is also an abbreviation for attributes: `attribute::` can be abbreviated to `@`.

For example, a location path `para[@type="warning"]` is short for `child::para[attribute::type="warning"]` and so selects para children with a type attribute with value equal to warning.

Path Abbreviation //

// is short for /descendant-or-self::node(). For example, //para is short for /descendant-or-self::node()/child::para and so will select any para element in the document (even a para element that is a document element will be selected by //para since the document element node is a child of the root node);

div//para is short for div/descendant-or-self::node()/child::para and so will select all para descendants of div children.

Note: Location path //para[1] does not mean the same as the location path /descendant::para[1]. The latter selects the first descendant para element; the former selects all descendant para elements that are the first para children of their parents.

Location Step Abbreviation .

A location step of . is short for self::node(). This is particularly useful in conjunction with //. For example, the location path ./para is short for:

```
self::node()/descendant-or-self::node()/child::para
```

and so will select all para descendant elements of the context node.

Location Step Abbreviation ..

Similarly, a location step of .. is short for parent::node(). For example, ../title is short for:

```
parent::node()/child::title
```

and so will select the title children of the parent of the context node.

Abbreviation Summary

AbbreviatedAbsolutePath ::= '/' RelativeLocationPath

AbbreviatedRelativeLocationPath ::= RelativeLocationPath '/' Step

AbbreviatedStep ::= '.' | '..'

AbbreviatedAxisSpecifier ::= '@'?

Relative and Absolute Location Paths

There are two kinds of location path:

- **Relative location paths.** A relative location path consists of a sequence of one or more location steps separated by /. The steps in a relative location path are composed together from left to right. Each step in turn selects a set of nodes relative to a context node. An initial sequence of steps is composed together with a following step as follows. The initial sequence of steps selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are unioned together. The set of nodes identified by the composition of the steps is this union.

For example, `child::div/child::para` selects the `para` element children of the `div` element children of the context node, or, in other words, the `para` element grandchildren that have `div` parents.

- **Absolute location paths.** An absolute location path consists of / optionally followed by a relative location path. A / by itself selects the root node of the document containing the context node. If it is followed by a relative location path, then the location path selects the set of nodes that would be selected by the relative location path relative to the root node of the document containing the context node.

Location Path Syntax Summary

Location path provides a means to search for target nodes. Here is the general syntax for location path:

```
axisname :: nodetest expr1 expr2 ...
```

```
LocationPath ::= RelativeLocationPath  
                | AbsoluteLocationPath  
AbsoluteLocationPath ::= '/' RelativeLocationPath?  
                | AbbreviatedAbsoluteLocationPath  
RelativeLocationPath ::= Step  
                | RelativeLocationPath '/' Step  
                | AbbreviatedRelativeLocationPath
```

XPath 1.0 Data Model

XPath operates on an XML document as a tree. This section describes how XPath models an XML document as a tree. The relationship of this model to the XML documents operated on by XPath must conform to the XML Namespaces Recommendation.

See Also: [Introducing the W3C Namespaces in XML Recommendation](#) on page C-18

Nodes

The tree contains nodes. There are seven types of node:

- [Root Nodes](#)
- [Element Nodes](#)
- [Text Nodes](#)
- [Attribute Nodes](#)
- [Namespace Nodes](#)
- [Processing Instruction Nodes](#)
- [Comment Nodes](#)

Root Nodes

The root node is the root of the tree. It does not occur except as the root of the tree. The element node for the document element is a child of the root node. The root node also has as children processing instruction and comment nodes for processing instructions and comments that occur in the prolog and after the end of the document element. The string-value of the root node is the concatenation of the string-values of all text node descendants of the root node in document order. The root node does not have an expanded-name.

Element Nodes

There is an element node for every element in the document. An element node has an expanded-name computed by expanding the `QName` of the element specified in the tag in accordance with the XML Namespaces Recommendation. The namespace URI of the element's expanded-name will be null if the `QName` has no prefix and there is no applicable default namespace.

Note: In the notation of Appendix A.3 of <http://www.w3.org/TR/REC-xml-names/>, the local part of the expanded-name corresponds to the type attribute of the ExpEType element; the namespace URI of the expanded-name corresponds to the ns attribute of the ExpEType element, and is null if the ns attribute of the ExpEType element is omitted.

The children of an element node are the element nodes, comment nodes, processing instruction nodes and text nodes for its content. Entity references to both internal and external entities are expanded. Character references are resolved. The string-value of an element node is the concatenation of the string-values of all text node descendants of the element node in document order.

Unique IDs. An element node may have a unique identifier (ID). This is the value of the attribute that is declared in the DTD as type ID. No two elements in a document may have the same unique ID. If an XML processor reports two elements in a document as having the same unique ID (which is possible only if the document is invalid) then the second element in document order must be treated as not having a unique ID.

Note: If a document does not have a DTD, then no element in the document will have a unique ID.

Text Nodes

Character data is grouped into text nodes. As much character data as possible is grouped into each text node: a text node never has an immediately following or preceding sibling that is a text node. The string-value of a text node is the characterdata. A text node always has at least one character of data. Each character within a CDATA section is treated as character data. Thus, `<![CDATA[<]]>` in the source document will be treated the same as `<`; `>`. Both will result in a single `<` character in a text node in the tree. Thus, a CDATA section is treated as if the `<![CDATA[and]]>` were removed and every occurrence of `<` and `&` were replaced by `<` and `&` respectively.

Note: When a text node that contains a < character is written out as XML, the < character must be escaped by for example, using <, or including it in a CDATA section. Characters inside comments, processing instructions and attribute values do not produce text nodes. Line-endings in external entities are normalized to #xA as specified in the XML Recommendation. A text node does not have an expanded-name.

Attribute Nodes

Each element node has an associated set of attribute nodes; the element is the parent of each of these attribute nodes; however, an attribute node is not a child of its parent element.

Note: This is different from the DOM, which does not treat the element bearing an attribute as the parent of the attribute.

Elements never share attribute nodes: if one element node is not the same node as another element node, then none of the attribute nodes of the one element node will be the same node as the attribute nodes of another element node.

Note: The = operator tests whether two nodes have the same value, not whether they are the same node. Thus attributes of two different elements may compare as equal using =, even though they are not the same node.

A defaulted attribute is treated the same as a specified attribute. If an attribute was declared for the element type in the DTD, but the default was declared as #IMPLIED, and the attribute was not specified on the element, then the element's attribute set does not contain a node for the attribute.

Some attributes, such as `xml:lang` and `xml:space`, have the semantics that they apply to all elements that are descendants of the element bearing the attribute, unless overridden with an instance of the same attribute on another descendant element. However, this does not affect where attribute nodes appear in the tree: an element has attribute nodes only for attributes that were explicitly specified in the start-tag or empty-element tag of that element or that were explicitly declared in the DTD with a default value.

An attribute node has an expanded-name and a string-value. The expanded-name is computed by expanding the `QName` specified in the tag in the XML document in accordance with the XML Namespaces Recommendation. The namespace URI of the attribute's name will be null if the `QName` of the attribute does not have a prefix.

Note: In the notation of Appendix A.3 of XML Namespaces Recommendation, the local part of the expanded-name corresponds to the name attribute of the `ExpAName` element; the namespace URI of the expanded-name corresponds to the `ns` attribute of the `ExpAName` element, and is null if the `ns` attribute of the `ExpAName` element is omitted.

An attribute node has a string-value. The string-value is the normalized value as specified by the XML Recommendation. An attribute whose normalized value is a zero-length string is not treated specially: it results in an attribute node whose string-value is a zero-length string.

Note: It is possible for default attributes to be declared in an external DTD or an external parameter entity. The XML Recommendation does not require an XML processor to read an external DTD or an external parameter unless it is validating. A stylesheet or other facility that assumes that the XPath tree contains default attribute values declared in an external DTD or parameter entity may not work with some non-validating XML processors.

There are no attribute nodes corresponding to attributes that declare namespaces.

Namespace Nodes

Each element has an associated set of namespace nodes, one for each distinct namespace prefix that is in scope for the element (including the `xml` prefix, which is implicitly declared by the XML Namespaces Recommendation) and one for the default namespace if one is in scope for the element. The element is the parent of each of these namespace nodes; however, a namespace node is not a child of its parent element.

Elements never share namespace nodes: if one element node is not the same node as another element node, then none of the namespace nodes of the one element node

will be the same node as the namespace nodes of another element node. This means that an element will have a namespace node:

- For every attribute on the element whose name starts with `xmlns:`;
- For every attribute on an ancestor element whose name starts `xmlns:` unless the element itself or a nearer ancestor redeclares the prefix;
- For an `xmlns` attribute, if the element or some ancestor has an `xmlns` attribute, and the value of the `xmlns` attribute for the nearest such element is non-empty

Note: An attribute `xmlns=""` "undeclares" the default namespace.

A namespace node has an expanded-name: the local part is the namespace prefix (this is empty if the namespace node is for the default namespace); the namespace URI is always NULL.

The string-value of a namespace node is the namespace URI that is being bound to the namespace prefix; if it is relative, it must be resolved just like a namespace URI in an expanded-name.

Processing Instruction Nodes

There is a processing instruction node for every processing instruction, except for any processing instruction that occurs within the document type declaration. A processing instruction has an expanded-name: the local part is the processing instruction's target; the namespace URI is NULL. The string-value of a processing instruction node is the part of the processing instruction following the target and any whitespace. It does not include the terminating `?>`.

Note: The XML declaration is not a processing instruction. Therefore, there is no processing instruction node corresponding to the XML declaration.

Comment Nodes

There is a comment node for every comment, except for any comment that occurs within the document type declaration. The string-value of comment is the content of the comment not including the opening `<!--` or the closing `-->`. A comment node does not have an expanded-name.

For every type of node, there is a way of determining a string-value for a node of that type. For some types of node, the string-value is part of the node; for other types of node, the string-value is computed from the string-value of descendant nodes.

Note: For element nodes and root nodes, the string-value of a node is not the same as the string returned by the DOM `nodeValue` method.

Expanded-Name

Some types of node also have an expanded-name, which is a pair consisting of:

- A local part. This is a string.
- A namespace URI. The namespace URI is either null or a string. If specified in the XML document it can be a URI reference as defined in RFC2396; this means it can have a fragment identifier and be relative. A relative URI should be resolved into an absolute URI during namespace processing; the namespace URIs of expanded-names of nodes in the data model should be absolute.

Two expanded-names are equal if they have the same local part, and either both have a null namespace URI or both have non-null namespace URIs that are equal.

Document Order

There is an ordering, document order, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Thus, the root node will be the first node.

Element nodes occur before their children. Thus, document order orders element nodes in order of the occurrence of their start-tag in the XML (after expansion of entities). The attribute nodes and namespace nodes of an element occur before the children of the element. The namespace nodes are defined to occur before the attribute nodes.

The relative order of namespace nodes is implementation-dependent.

The relative order of attribute nodes is implementation-dependent.

Reverse document order is the reverse of document order.

Root nodes and element nodes have an ordered list of child nodes. Nodes never share children: if one node is not the same node as another node, then none of the

children of the one node will be the same node as any of the children of another node.

Every node other than the root node has exactly one parent, which is either an element node or the root node. A root node or an element node is the parent of each of its child nodes. The descendants of a node are the children of the node and the descendants of the children of the node.

Introducing the W3C XML Path Language (XPath) 2.0 Working Draft

XPath 2.0 is the result of joint work by the W3C XSL and XML Query Working Groups. XPath 2.0 is a language derived from both XPath 1.0 and XQuery. The XPath 2.0 and XQuery 1.0 Working Drafts are generated from a common source. These languages are closely related and share much of the same expression syntax and semantics. The two Working Drafts in places are identical.

XPath is designed to be embedded in a host language such as XSLT or XQuery. XPath has a natural subset that can be used for matching, that is, testing whether or not a node matches a pattern.

XQuery Version 1.0 contains XPath Version 2.0 as a subset. Any expression that is syntactically valid and executes successfully in both XPath 2.0 and XQuery 1.0 will return the same result in both languages.

XPath also depends on and is closely related to the following specifications:

- The XPath data model defines the information in an XML document that is available to an XPath processor. The data model is defined in XQuery 1.0 and XPath 2.0 Data Model.
- The static and dynamic semantics of XPath are formally defined in XQuery 1.0 Formal Semantics. This is done by mapping the full XPath language into a "core" subset for which the semantics is defined. This document is useful for implementors and others who require a rigorous definition of XPath.
- The library of functions and operators supported by XPath is defined in XQuery 1.0 and XPath 2.0 Functions and Operators.

XPath 2.0 Expressions

The basic building block of XPath is the expression. The language provides several kinds of expressions which may be constructed from keywords, symbols, and operands. In general, the operands of an expression are other expressions.

XPath is a functional language which allows various kinds of expressions to be nested with full generality. It is also a strongly-typed language in which the operands of various expressions, operators, and functions must conform to designated types.

Like XML, XPath is a case-sensitive language. All keywords in XPath use lower-case characters.

Expr

```
 ::=
  OrExpr
  | AndExpr
  | ForExpr
  | QuantifiedExpr
  | IfExpr
  | GeneralComp
  | ValueComp
  | NodeComp
  | OrderComp
  | InstanceofExpr
  | RangeExpr
  | AdditiveExpr
  | MultiplicativeExpr
  | UnionExpr
  | IntersectExceptExpr
  | UnaryExpr
  | CastExpr
  | PathExpr
```

Introducing the W3C Namespaces in XML Recommendation

Software modules must recognize tags and attributes which they are designed to process, even in the face of "collisions" occurring when markup intended for some other software package uses the same element type or attribute name.

Document constructs should have universal names, whose scope extends beyond their containing document. The W3C Namespaces in XML Recommendation describes the mechanism, XML namespaces, which accomplishes this.

See Also: <http://www.w3.org/TR/REC-xml-names/>

What Is a Namespace?

An XML namespace is a collection of names, identified by a URI reference [RFC2396], which are used in XML documents as element types and attribute names. XML namespaces differ from the "namespaces" conventionally used in computing disciplines in that the XML version has internal structure and is not, mathematically speaking, a set. These issues are discussed in the W3C Namespace Recommendation, appendix, "A. The Internal Structure of XML Namespaces".

URI References

URI references which identify namespaces are considered identical when they are exactly the same character-for-character. Note that URI references which are not identical in this sense may in fact be functionally equivalent. Examples include URI references which differ only in case, or which are in external entities which have different effective base URIs.

Names from XML namespaces may appear as qualified names, which contain a single colon, separating the name into a namespace prefix and a local part.

The prefix, which is mapped to a URI reference, selects a namespace. The combination of the universally managed URI namespace and the document's own namespace produces identifiers that are universally unique. Mechanisms are provided for prefix scoping and defaulting.

URI references can contain characters not allowed in names, so cannot be used directly as namespace prefixes. Therefore, the namespace prefix serves as a proxy for a URI reference. An attribute-based syntax described in the following section is used to declare the association of the namespace prefix with a URI reference; software which supports this namespace proposal must recognize and act on these declarations and prefixes.

Notation and Usage

Many of the nonterminals in the productions in this specification are defined not here but in the W3C XML Recommendation. When nonterminals defined here have the same names as nonterminals defined in the W3C XML Recommendation, the productions here in all cases match a subset of the strings matched by the corresponding ones there.

In this document's productions, the NSC is a "Namespace Constraint", one of the rules that documents conforming to this specification must follow.

All Internet domain names used in examples, with the exception of w3.org, are selected at random and should not be taken as having any import.

Declaring Namespaces

A namespace is declared using a family of reserved attributes. Such an attribute's name must either be `xmlns` or have `xmlns:` as a prefix. These attributes, like any other XML attributes, can be provided directly or by default.

Attribute Names for Namespace Declaration

```
[1] NSAttName ::=   PrefixedAttName
                  | DefaultAttName
[2] PrefixedAttName ::= 'xmlns:' NCName

[NSC: Leading "XML" ]
[3] DefaultAttName ::= 'xmlns'
[4] NCName ::= (Letter | '_' ) (NCNameChar)*

/* An XML Name, minus the ":" */
[5] NCNameChar ::= Letter | Digit | '.' | '-' | '_' | CombiningChar
                  | Extender
```

The attribute's value, a URI reference, is the namespace name identifying the namespace. The namespace name, to serve its intended purpose, should have the characteristics of uniqueness and persistence. It is not a goal that it be directly usable for retrieval of a schema (if any exists). An example of a syntax that is designed with these goals in mind is that for Uniform Resource Names [RFC2141]. However, it should be noted that ordinary URLs can be managed in such a way as to achieve these same goals.

When the Attribute Name Matches the `PrefixedAttName`

If the attribute name matches `PrefixedAttName`, then the `NCName` gives the namespace prefix, used to associate element and attribute names with the namespace name in the attribute value in the scope of the element to which the declaration is attached. In such declarations, the namespace name may not be empty.

When the Attribute Name Matches the `DefaultAttName`

If the attribute name matches `DefaultAttName`, then the namespace name in the attribute value is that of the default namespace in the scope of the element to which the declaration is attached. In such a default declaration, the attribute value may be empty. Default namespaces and overriding of declarations are discussed in section ["Applying Namespaces to Elements and Attributes"](#) on page C-23 of the W3C Namespace Recommendation.

The following example namespace declaration associates the namespace prefix `edi` with the namespace name `http://ecommerce.org/schema`:

```
<x xmlns:edi='http://ecommerce.org/schema'>
  <!-- the "edi" prefix is bound to http://ecommerce.org/schema
        for the "x" element and contents -->
</x>
```

Namespace Constraint: Leading "XML"

Prefixes beginning with the three-letter sequence `x`, `m`, `l`, in any case combination, are reserved for use by XML and XML-related specifications.

Qualified Names

In XML documents conforming to the W3C Namespace Recommendation, some names (constructs corresponding to the nonterminal `Name`) may be given as qualified names, defined as follows:

Qualified Name Syntax

```
[6] QName ::= (Prefix ':')? LocalPart
[7] Prefix ::= NCName
[8] LocalPart ::= NCName
```

What is the Prefix?

The Prefix provides the namespace prefix part of the qualified name, and must be associated with a namespace URI reference in a namespace declaration.

The LocalPart provides the local part of the qualified name. Note that the prefix functions only as a placeholder for a namespace name. Applications should use the namespace name, not the prefix, in constructing names whose scope extends beyond the containing document.

Using Qualified Names

In XML documents conforming to the W3C Namespace Recommendation, element types are given as qualified names, as follows:

Element Types

```
[9] STag ::= '<' QName (S Attribute)* S? '>' [NSC: Prefix Declared ]
[10] ETag ::= '</' QName S? '>' [NSC: Prefix Declared ]
[11] EmptyElemTag ::= '<' QName (S Attribute)* S? '/>' [NSC: Prefix Declared ]
```

The following is an example of a qualified name serving as an element type:

```
<x xmlns:edi='http://ecommerce.org/schema'>
  <!-- the 'price' element's namespace is http://ecommerce.org/schema -->
  <edi:price units='Euro'>32.18</edi:price>
</x>
```

Attributes are either namespace declarations or their names are given as qualified names:

Attribute

```
[12] Attribute ::= NSAttName Eq AttValue | QName Eq AttValue [NSC:Prefix Declared]
```

The following is an example of a qualified name serving as an attribute name:

```
<x xmlns:edi='http://ecommerce.org/schema'>
  <!-- the 'taxClass' attribute's namespace is http://ecommerce.org/schema -->
  <lineItem edi:taxClass="exempt">Baby food</lineItem>
</x>
```

Namespace Constraint: Prefix Declared

The namespace prefix, unless it is `xml` or `xmlns`, must have been declared in a namespace declaration attribute in either the start-tag of the element where the prefix is used or in an ancestor element, that is, an element in whose content the prefixed markup occurs:

The prefix `xml` is by definition bound to the namespace name `http://www.w3.org/XML/1998/namespace`.

The prefix `xmlns` is used only for namespace bindings and is not itself bound to any namespace name.

This constraint may lead to operational difficulties in the case where the namespace declaration attribute is provided, not directly in the XML document entity, but through a default attribute declared in an external entity. Such declarations may not be read by software which is based on a non-validating XML processor.

Many XML applications, presumably including namespace-sensitive ones, fail to require validating processors. For correct operation with such applications, namespace declarations must be provided either directly or through default attributes declared in the internal subset of the DTD.

Element names and attribute types are also given as qualified names when they appear in declarations in the DTD:

Qualified Names in Declarations

```
[13] doctypedecl ::= '<!DOCTYPE' S QName (S ExternalID)? S? ('[' (markupdecl |
    PEReference | S)* ']' S?)? '>'
[14] elementdecl ::= '<!ELEMENT' S QName S contentSpec S? '>'
[15] cp           ::= (QName | choice | seq) ('?' | '*' | '+')?
[16] Mixed       ::= '(' S? '#PCDATA' (S? '|' S? QName)* S? ')' *
    | '(' S? '#PCDATA' S? ')'
[17] AttlistDecl ::= '<!ATTLIST' S QName AttDef* S? '>'
[18] AttDef      ::= S (QName | NSAttName) S AttType S DefaultDecl
```

Applying Namespaces to Elements and Attributes

Namespace Scoping

The namespace declaration is considered to apply to the element where it is specified and to all elements within the content of that element, unless overridden by another namespace declaration with the same NSAttName part:

```
<?xml version="1.0"?>
  <!-- all elements here are explicitly in the HTML namespace -->
  <html:html xmlns:html='http://www.w3.org/TR/REC-html40'>
    <html:head><html:title>Frobnostication</html:title></html:head>
    <html:body><html:p>Moved to
      <html:a href='http://frob.com'>here.</html:a></html:p></html:body>
  </html:html>
```

Multiple namespace prefixes can be declared as attributes of a single element, as shown in this example:

```
<?xml version="1.0"?>
  <!-- both namespace prefixes are available throughout -->
  <bk:book xmlns:bk='urn:loc.gov:books'
    xmlns:isbn='urn:ISBN:0-395-36341-6'>
    <bk:title>Cheaper by the Dozen</bk:title>
    <isbn:number>1568491379</isbn:number>
  </bk:book>
```

Namespace Defaulting

A default namespace is considered to apply to the element where it is declared (if that element has no namespace prefix), and to all elements with no prefix within the content of that element. If the URI reference in a default namespace declaration is empty, then unprefixes elements in the scope of the declaration are not considered to be in any namespace. Note that default namespaces do not apply directly to attributes.

```
<?xml version="1.0"?>
  <!-- elements are in the HTML namespace, in this case by default -->
  <html xmlns='http://www.w3.org/TR/REC-html40'>
    <head><title>Frobnostication</title></head>
    <body><p>Moved to
      <a href='http://frob.com'>here</a>.</p></body>
  </html>
```

```
<?xml version="1.0"?>
  <!-- unprefixes element types are from "books" -->
  <book xmlns='urn:loc.gov:books'
        xmlns:isbn='urn:ISBN:0-395-36341-6'>
    <title>Cheaper by the Dozen</title>
    <isbn:number>1568491379</isbn:number>
  </book>
```

A larger example of namespace scoping:

```
<?xml version="1.0"?>
  <!-- initially, the default namespace is "books" -->
  <book xmlns='urn:loc.gov:books'
        xmlns:isbn='urn:ISBN:0-395-36341-6'>
    <title>Cheaper by the Dozen</title>
    <isbn:number>1568491379</isbn:number>
    <notes>
      <!-- make HTML the default namespace for some commentary -->
      <p xmlns='urn:w3-org-ns:HTML'>
        This is a <i>funny</i> book!
      </p>
    </notes>
  </book>
```

The default namespace can be set to the empty string. This has the same effect, within the scope of the declaration, of there being no default namespace.

```
<?xml version='1.0'?>
  <Beers>
```

```

<!-- the default namespace is now that of HTML -->
<table xmlns='http://www.w3.org/TR/REC-html40'>
  <thead><tr><th><td>Name</td><td>Origin</td><td>Description</td></tr>
  <tbody><tr>
    <!-- no default namespace inside table cells -->
    <td><brandName xmlns="">Huntsman</brandName></td>
    <td><origin xmlns="">Bath, UK</origin></td>
    <td>
      <details xmlns=""><class>Bitter</class><hop>Fuggles</hop>
        <pro>Wonderful hop, light alcohol, good summer beer</pro>
        <con>Fragile; excessive variance pub to pub</con>
      </details>
    </td>
  </tr>
</tbody>
</table>
</Beers>

```

Uniqueness of Attributes

In XML documents conforming to this specification, no tag may contain two attributes which:

- Have identical names, or
- Have qualified names with the same local part and with prefixes which have been bound to namespace names that are identical.

For example, each of the bad start-tags is illegal in the following:

```

<!-- http://www.w3.org is bound to n1 and n2 -->
<x xmlns:n1="http://www.w3.org"
  xmlns:n2="http://www.w3.org" >
  <bad a="1"      a="2" />
  <bad n1:a="1"  n2:a="2" />
</x>

```

However, each of the following is legal, the second because the default namespace does not apply to attribute names:

```

<!-- http://www.w3.org is bound to n1 and is the default -->
<x xmlns:n1="http://www.w3.org"
  xmlns="http://www.w3.org" >
  <good a="1"      b="2" />
  <good a="1"      n1:a="2" />
</x>

```

Conformance of XML Documents

In XML documents which conform to the W3C Namespace Recommendation, element types and attribute names must match the production for `QName` and must satisfy the "Namespace Constraints".

An XML document conforms to this specification if all other tokens in the document which are required, for XML conformance, to match the XML production for `Name`, match this specification's production for `NCName`.

The effect of conformance is that in such a document:

- All element types and attribute names contain either zero or one colon.
- No entity names, PI targets, or notation names contain any colons.

Strictly speaking, attribute values declared to be of types `ID`, `IDREF(S)`, `ENTITY(IES)`, and `NOTATION` are also Names, and thus should be colon-free.

However, the declared type of attribute values is only available to processors which read markup declarations, for example validating processors. Thus, unless the use of a validating processor has been specified, there can be no assurance that the contents of attribute values have been checked for conformance to this specification.

The following W3C Namespace Recommendation Appendixes are not included in this primer:

- A. The Internal Structure of XML Namespaces (Non-Normative)
 - A.1 The Insufficiency of the Traditional Namespace
 - A.2 XML Namespace Partitions
 - A.3 Expanded Element Types and Attribute Names
 - A.4 Unique Expanded Attribute Names

Introducing the W3C XML Information Set

The W3C XML Information Set specification defines an abstract data set called the XML Information Set (Infoset). It provides a consistent set of definitions for use in other specifications that need to refer to the information in a well-formed XML document.

The primary criterion for inclusion of an information item or property has been that of expected usefulness in future specifications. It does not constitute a minimum set of information that must be returned by an XML processor.

An XML document has an information set if it is well-formed and satisfies the namespace constraints described in the following section.

There is no requirement for an XML document to be valid in order to have an information set.

See Also: <http://www.w3.org/TR/xml-infoset/>

Information sets may be created by methods (not described in this specification) other than parsing an XML document. See "[Synthetic Infosets](#)" on page C-29.

An XML document's information set consists of a number of information items; the information set for any well-formed XML document will contain at least a document information item and several others. An information item is an abstract description of some part of an XML document: each information item has a set of associated named properties. In this specification, the property names are shown in square brackets, [thus]. The types of information item are listed in section 2.

The XML Information Set does not require or favor a specific interface or class of interfaces. This specification presents the information set as a modified tree for the sake of clarity and simplicity, but there is no requirement that the XML Information Set be made available through a tree structure; other types of interfaces, including (but not limited to) event-based and query-based interfaces, are also capable of providing information conforming to the XML Information Set.

The terms "information set" and "information item" are similar in meaning to the generic terms "tree" and "node", as they are used in computing. However, the former terms are used in this specification to reduce possible confusion with other specific data models. Information items do not map one-to-one with the nodes of the DOM or the "tree" and "nodes" of the XPath data model.

In this specification, the words "must", "should", and "may" assume the meanings specified in [RFC2119], except that the words do not appear in uppercase.

Namespaces

XML 1.0 documents that do not conform to the W3C Namespace Recommendation, though technically well-formed, are not considered to have meaningful information sets. That is, this specification does not define an information set for documents that have element or attribute names containing colons that are used in other ways than as prescribed by the W3C Namespace Recommendation.

Also, the XML Infoset specification does not define an information set for documents which use relative URI references in namespace declarations. This is in

accordance with the decision of the W3C XML Plenary Interest Group described in Relative Namespace URI References in the W3C Namespace Recommendation.

The value of a namespace name property is the normalized value of the corresponding namespace attribute; no additional URI escaping is applied to it by the processor.

Entities

An information set describes its XML document with entity references already expanded, that is, represented by the information items corresponding to their replacement text. However, there are various circumstances in which a processor may not perform this expansion. An entity may not be declared, or may not be retrievable. A non-validating processor may choose not to read all declarations, and even if it does, may not expand all external entities. In these cases an unexpanded entity reference information item is used to represent the entity reference.

End-of-Line Handling

The values of all properties in the Infoset take account of the end-of-line normalization described in the XML Recommendation, 2.11 "End-of-Line Handling".

Base URIs

Several information items have a base URI or declaration base URI property. These are computed according to XML Base. Note that retrieval of a resource may involve redirection at the parser level (for example, in an entity resolver) or at a lower level; in this case the base URI is the final URI used to retrieve the resource after all redirection.

The value of these properties does not reflect any URI escaping that may be required for retrieval of the resource, but it may include escaped characters if these were specified in the document, or returned by a server in the case of redirection.

In some cases (such as a document read from a string or a pipe) the rules in XML Base may result in a base URI being application dependent. In these cases this specification does not define the value of the base URI or declaration base URI property.

When resolving relative URIs the base URI property should be used in preference to the values of `xml:base` attributes; they may be inconsistent in the case of Synthetic Infosets.

Unknown and No Value

Some properties may sometimes have the value unknown or no value, and it is said that a property value is unknown or that a property has no value respectively. These values are distinct from each other and from all other values. In particular they are distinct from the empty string, the empty set, and the empty list, each of which simply has no members. This specification does not use the term null since in some communities it has particular connotations which may not match those intended here.

Synthetic Infosets

This specification describes the information set resulting from parsing an XML document. Information sets may be constructed by other means, for example by use of an API such as the DOM or by transforming an existing information set.

An information set corresponding to a real document will necessarily be consistent in various ways; for example the in-scope namespaces property of an element will be consistent with the [namespace attributes] properties of the element and its ancestors. This may not be true of an information set constructed by other means; in such a case there will be no XML document corresponding to the information set, and to serialize it will require resolution of the inconsistencies (for example, by outputting namespace declarations that correspond to the namespaces in scope).

XSLT Primer

This appendix describes introductory information about the W3C XSL and XSLT Recommendation. It contains the following sections:

- [Introducing XSL](#)
- [XSL Transformation \(XSLT\)](#)
- [XML Path Language \(XPath\)](#)
- [CSS Versus XSL](#)
- [XSL Stylesheet Example, PurchaseOrder.xsl](#)

Introducing XSL

XML documents have structure but no format. Extensible Stylesheet Language (XSL) adds formatting to XML documents. It provides a way to display XML semantics and can map XML elements into other formatting languages such as HTML.

See Also:

- *Oracle9i XML Case Studies and Applications* in particular, the chapters that describe customizing content, Oracle9i Wireless Edition, and customizing presentation with XML and XSL.
- <http://www.oasis-open.org/cover/xsl.html>
- <http://www.mulberrytech.com/xsl/xsl-list/>
- http://www.builder.com/Authoring/XmlSpot/?tag=st.cn.sr1.ssr.bl_xml
- <http://www.zvon.org/HTMLOnly/XSLTutorial/Books/Book1/index.html>
- [Chapter 6, "Transforming and Validating XMLType Data"](#)

The W3C XSL Transformation Recommendation Version 1.0

This specification defines the syntax and semantics of XSLT, which is a language for transforming XML documents into other XML documents.

XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL.

See Also: <http://www.w3.org/TR/xslt>

This specification defines the syntax and semantics of the XSLT language. A transformation in the XSLT language is expressed as a well-formed XML document

conforming to the Namespaces in XML Recommendation, which may include both elements that are defined by XSLT and elements that are not defined by XSLT.

XSLT-defined elements are distinguished by belonging to a specific XML namespace (see [2.1 XSLT Namespace]), which is referred to in this specification as the XSLT namespace. Thus this specification is a definition of the syntax and semantics of the XSLT namespace.

A transformation expressed in XSLT describes rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create part of the result tree. The result tree is separate from the source tree. The structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added.

A transformation expressed in XSLT is called a stylesheet. This is because, in the case when XSLT is transforming into the XSL formatting vocabulary, the transformation functions as a stylesheet.

This document does not specify how an XSLT stylesheet is associated with an XML document. It is recommended that XSL processors support the mechanism described in. When this or any other mechanism yields a sequence of more than one XSLT stylesheet to be applied simultaneously to a XML document, then the effect should be the same as applying a single stylesheet that imports each member of the sequence in order.

A stylesheet contains a set of template rules. A template rule has two parts: a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

The W3C is developing the XSL specification as part of its Style Sheets Activity. XSL has document manipulation capabilities beyond styling. It is a stylesheet language for XML.

The July 1999 W3C XSL specification, was split into two separate documents:

- XSL syntax and semantics
- How to use XSL to apply style sheets to transform one document into another

The formatting objects used in XSL are based on prior work on Cascading Style Sheets (CSS) and the Document Style Semantics & Specification Language (DSSSL). XSL is designed to be easier to use than DSSSL.

Capabilities provided by XSL as defined in the proposal enable the following functionality:

- Formatting of source elements based on ancestry and descendency, position, and uniqueness
- The creation of formatting constructs including generated text and graphics
- The definition of reusable formatting macros
- Writing-direction independent stylesheets
- An extensible set of formatting objects.

See Also: <http://www.w3.org/Style/XSL/>

XSL Specification Proposal

The XSL specification defines XSL as a language for expressing stylesheets. Given a class of arbitrarily structured XML documents or data files, designers use an XSL stylesheet to express their intentions about how that structured content should be presented; that is, how the source content should be styled, laid out, and paginated in a presentation medium, such as a window in a Web browser or a hand-held device, or a set of physical pages in a catalog, report, pamphlet, or book. Formatting is enabled by including formatting semantics in the result tree.

Formatting semantics are expressed in terms of a catalog of classes of formatting objects. The nodes of the result tree are formatting objects. The classes of formatting objects denote typographic abstractions such as page, paragraph, table, and so forth.

Finer control over the presentation of these abstractions is provided by a set of formatting properties, such as those controlling indents, word and letter spacing, and widow, orphan, and hyphenation control. In XSL, the classes of formatting objects and formatting properties provide the vocabulary for expressing presentation intent.

An implementation is not mandated to provide these as separate processes. Furthermore, implementations are free to process the source document in any way that produces the same result as if it were processed using the conceptual XSL processing model.

Namespaces in XML

A namespace is a unique identifier or name. This is needed because XML documents can be authored separately with different DTDs or XML Schemas.

Namespaces prevent conflicts in markup tags by identifying which DTD or XML Schema a tag comes from. Namespaces link an XML element to a specific DTD or XML Schema.

Before you can use a namespace marker such as `rml:`, `xhtml:`, or `xsl:`, you must identify it using the namespace indicator, `xmlns` as shown in the next paragraph.

See Also: <http://w3.org/TR/REC-xml-names>

XSL Stylesheet Architecture

The XSL stylesheets must include the following syntax:

- Start tag stating the stylesheet, such as `<xsl:stylesheet2>`
- Namespace indicator, such as `xmlns:xsl="http://www.w3.org/TR/WD-xsl"` for an XSL namespace indicator and `xmlns:fo="http://www.w3.org/TR/WD-xsl/FO"` for a formatting object namespace indicator
- Template rules including font families and weight, colors, and breaks. The templates have instructions that control the element and element values
- End of stylesheet declaration, `</xsl:stylesheet2>`

XSL Transformation (XSLT)

XSLT is designed to be used as part of XSL. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

Meanwhile the second part is concerned with the XSL formatting objects, their attributes, and how they can be combined.

See Also: [Chapter 6, "Transforming and Validating XMLType Data"](#)

XSLT 1.1 Specification

The W3C Working Group on XSL has released a document describing the requirements for the XSLT 1.1 specification. The primary goal of the XSLT 1.1 specification is to improve stylesheet portability. The new draft is available at <http://www.w3.org/TR/xslt11req>

In addition the XSLT 1.0 extension mechanism provides for an additional built-in transformation functionality.

Benefits of the extensions come at the price of portability. Since XSLT 1.0 provides no details or guidance on the implementation of extensions, today any user-written or built-in extensions are inevitably tied to a single XSLT processor.

Goal 1. Improve Stylesheet Portability

The primary goal of the XSLT 1.1 specification is to improve stylesheet portability. This goal will be achieved by standardizing the mechanism for implementing extension functions, and by including in the core XSLT specification two of the built-in extensions that many existing vendors XSLT processors have added due to user demand:

- Support for multiple output documents from a transformation
- Support for converting a result tree fragment to a nodeset for further processing
By standardizing these extension-related aspects which multiple vendor implementations already provide, the ability to create stylesheets that work across multiple XSLT processors should improve dramatically.

Goal 2. Support the New XML Specification

A secondary goal of the XSLT 1.1 specification is to support the new XML base specification.

The XSLT 1.1 specification proposal provides the requirements that will achieve these goals. The working group has decided to limit the scope of XSLT 1.1 to the standardization of features already implemented in several XSLT 1.0 processors, and concentrate first on standardizing the implementation of extension functions.

Standardization of extension elements and support for new XML Schema data type aware facilities are planned for XSLT 2.0.

XML Path Language (XPath)

A separate, related specification is published as the XML Path Language (XPath) Version 1.0. XPath is a language for addressing parts of an XML document, essential for cases where you want to specify exactly which parts of a document are to be transformed by XSL. For example, XPath lets you select all paragraphs belonging to the chapter element, or select the elements called special notes. XPath is designed to be used by both XSLT and XPointer. XPath is the result of an effort to provide a

common syntax and semantics for functionality shared between XSL transformations and XPointer.

See Also: [Appendix C, "XPath and Namespace Primer"](#)

CSS Versus XSL

W3C is working to ensure that interoperable implementations of the formatting model is available.

Cascading Stylesheets (CSS)

Cascading Stylesheets (CSS) can be used to style HTML documents. CSS were developed by the W3C Style Working Group. CSS2 is a style sheet language that allows authors and users to attach styles (for example, fonts, spacing, or aural cues) to structured documents, such as HTML documents and XML applications.

By separating the presentation style of documents from the content of documents, CSS2 simplifies Web authoring and site maintenance.

XSL

XSL, on the other hand, is able to transform documents. For example, XSL can be used to transform XML data into HTML/CSS documents on the Web server. This way, the two languages complement each other and can be used together. Both languages can be used to style XML documents. CSS and XSL will use the same underlying formatting model and designers will therefore have access to the same formatting features in both languages.

The model used by XSL for rendering documents on the screen builds on years of work on a complex ISO-standard style language called DSSSL. Aimed mainly at complex documentation projects, XSL also has many uses in automatic generation of tables of contents, indexes, reports, and other more complex publishing tasks.

XSL Stylesheet Example, PurchaseOrder.xsl

The following example, `PurchaseOrder.xsl`, is an example of an XSL stylesheet. the example stylesheet is used in the examples in [Chapter 3, "Using Oracle XML DB"](#).

```
<?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="#FFFFFF" 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>
      </xsl:for-each select="PurchaseOrder"/>
      <xsl:for-each select="PurchaseOrder">
        <table border="0" width="100%" bgcolor="#000000">
          <tbody>
            <tr>
              <td width="296">
                <P>
                  <B>
```

```

<FONT SIZE="+1" COLOR="#FF0000"
  FACE="Arial, Helvetica, sans-serif">Internal
</FONT>
</B>
</P>
<table border="0" width="98%" BGCOLOR="#000099">
<tbody>
<tr>
<td>
<td WIDTH="49%">
<B>
<FONT COLOR="#FFFF00">Actions</FONT>
</B>
</td>
<td WIDTH="51%">
<xsl:for-each select="Actions">
<xsl:for-each select="Action">
<table border="1" WIDTH="143">
<xsl:if test="position()=1">
<thead>
<tr>
<td HEIGHT="21">
<FONT
  COLOR="#FFFF00">User</FONT>
</td>
<td HEIGHT="21">
<FONT
  COLOR="#FFFF00">Date</FONT>
</td>
</tr>
</thead>
</xsl:if>
<tbody>
<tr>
<td>
<xsl:for-each select="User">
<xsl:apply-templates/>
</xsl:for-each>
</td>
<td>
<xsl:for-each select="Date">
<xsl:apply-templates/>
</xsl:for-each>
</td>
</tr>
</tbody>
</tbody>

```

```
        </table>
        </xsl:for-each>
    </xsl:for-each>
</td>
</tr>
<tr>
    <td WIDTH="49%">
        <B>
            <FONT COLOR="#FFFF00">Requestor</FONT>
        </B>
    </td>
    <td WIDTH="51%">
        <xsl:for-each select="Requestor">
            <xsl:apply-templates/>
        </xsl:for-each>
    </td>
</tr>
<tr>
    <td WIDTH="49%">
        <B>
            <FONT COLOR="#FFFF00">User</FONT>
        </B>
    </td>
    <td WIDTH="51%">
        <xsl:for-each select="User">
            <xsl:apply-templates/>
        </xsl:for-each>
    </td>
</tr>
<tr>
    <td WIDTH="49%">
        <B>
            <FONT COLOR="#FFFF00">Cost Center</FONT>
        </B>
    </td>
    <td WIDTH="51%">
        <xsl:for-each select="CostCenter">
            <xsl:apply-templates/>
        </xsl:for-each>
    </td>
</tr>
</tbody>
</table>
</td>
<td width="93"/>
```

```

<td valign="top" WIDTH="340">
  <B>
    <FONT COLOR="#FF0000">
      <FONT SIZE="+1">Ship To</FONT>
    </FONT>
  </B>
  <xsl:for-each select="ShippingInstructions">
    <xsl:if test="position()=1"/>
  </xsl:for-each>
  <xsl:for-each select="ShippingInstructions">
    <xsl:if test="position()=1">
      <table border="0" BGCOLOR="#999900">
        <tbody>
          <tr>
            <td WIDTH="126" HEIGHT="24">
              <B>Name</B>
            </td>
            <xsl:for-each
              select="./ShippingInstructions">
              <td WIDTH="218" HEIGHT="24">
                <xsl:for-each select="name">
                  <xsl:apply-templates/>
                </xsl:for-each>
              </td>
            </xsl:for-each>
          </tr>
          <tr>
            <td WIDTH="126" HEIGHT="34">
              <B>Address</B>
            </td>
            <xsl:for-each
              select="./ShippingInstructions">
              <td WIDTH="218" HEIGHT="34">
                <xsl:for-each select="address">
                  <span style="white-space:pre">
                    <xsl:apply-templates/>
                  </span>
                </xsl:for-each>
              </td>
            </xsl:for-each>
          </tr>
          <tr>
            <td WIDTH="126" HEIGHT="32">
              <B>Telephone</B>
            </td>

```

```

        <xsl:for-each
            select="../ShippingInstructions">
            <td WIDTH="218" HEIGHT="32">
                <xsl:for-each select="telephone">
                    <xsl:apply-templates/>
                </xsl:for-each>
            </td>
        </xsl:for-each>
    </tr>
</tbody>
</table>
</xsl:if>
</xsl:for-each>
</td>
</tr>
</tbody>
</table>
<br/>
<B>
    <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
</B>
<br/>
<br/>
<table border="0">
    <xsl:for-each select="LineItems">
        <xsl:for-each select="LineItem">
            <xsl:if test="position()=1">
                <thead>
                    <tr bgcolor="#C0C0C0">
                        <td>
                            <FONT COLOR="#FF0000">
                                <B>ItemNumber</B>
                            </FONT>
                        </td>
                        <td>
                            <FONT COLOR="#FF0000">
                                <B>Description</B>
                            </FONT>
                        </td>
                        <td>
                            <FONT COLOR="#FF0000">
                                <B>PartId</B>
                            </FONT>
                        </td>
                        <td>

```

```

        <FONT COLOR="#FF0000">
            <B>Quantity</B>
        </FONT>
    </td>
    <td>
        <FONT COLOR="#FF0000">
            <B>Unit Price</B>
        </FONT>
    </td>
    <td>
        <FONT COLOR="#FF0000">
            <B>Total Price</B>
        </FONT>
    </td>
</tr>
</thead>
</xsl:if>
<tbody>
<tr bgcolor="#DADADA">
    <td>
        <FONT COLOR="#000000">
            <xsl:for-each select="@ItemNumber">
                <xsl:value-of select="."/>
            </xsl:for-each>
        </FONT>
    </td>
    <td>
        <FONT COLOR="#000000">
            <xsl:for-each select="Description">
                <xsl:apply-templates/>
            </xsl:for-each>
        </FONT>
    </td>
    <td>
        <FONT COLOR="#000000">
            <xsl:for-each select="Part">
                <xsl:for-each select="@Id">
                    <xsl:value-of select="."/>
                </xsl:for-each>
            </xsl:for-each>
        </FONT>
    </td>
    <td>
        <FONT COLOR="#000000">
            <xsl:for-each select="Part">

```

```
        <xsl:for-each select="@Quantity">
            <xsl:value-of select="."/>
        </xsl:for-each>
    </xsl:for-each>
</FONT>
</td>
<td>
    <FONT COLOR="#000000">
        <xsl:for-each select="Part">
            <xsl:for-each select="@UnitPrice">
                <xsl:value-of select="."/>
            </xsl:for-each>
        </xsl:for-each>
    </FONT>
</td>
<td>
    <FONT FACE="Arial, Helvetica, sans-serif"
        COLOR="#000000">
        <xsl:for-each select="Part">
            <xsl:value-of select="@Quantity*@UnitPrice"/>
        </xsl:for-each>
    </FONT>
</td>
</tr>
</tbody>
</xsl:for-each>
</xsl:for-each>
</table>
</xsl:for-each>
</FONT>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Java DOM and Java Bean API for XMLType, Resource API for Java/JNDI: Quick Reference

This appendix contains a quick reference for the following Oracle XML DB Java APIs:

- [Java DOM API For XMLType](#)
- [Java Bean API for XMLType](#)
- [Oracle XML DB Resource API for Java/JNDI](#)

Java DOM API For XMLType

Packages `oracle.xml.db` and `oracle.xml.db.dom` implements the Java DOM API for XMLType. Java DOM API for XMLType implements the W3C DOM Recommendation Level 1.0 and Level 2.0 Core and also provides Oracle-specific extensions.

[Table E-1](#) lists the Java DOM API for XMLType (`oracle.xml.db.dom` and `oracle.xml.db`) classes. Note that class XMLType is in package `oracle.xml.db` and not `oracle.xml.db.dom`.

See Also:

- [Oracle9i XML API Reference - XDK and Oracle XML DB](#)
- [Chapter 9, "Java and Java Bean APIs for XMLType"](#)

Table E-1 Java DOM API for XMLType (mostly oracle.xml.db.dom) Classes

Java DOM API for XMLType	Description
XDBAttribute	Implements the W3C DOM Node interface for interacting with XOBs.
XDBCData	Implements <code>org.w3c.dom.CData</code> , the W3C text interface.
XDBCharData	Implements <code>org.w3c.dom.CharData</code> , the W3C CharacterData interface.
XDBComment	Implements the <code>org.w3c.dom.Comment</code> interface.
XDBDocument	Implements the <code>org.w3c.dom.Document</code> interface. Methods: XDBDocument() constructor: XDBDocument();Creates new Document. Can be used in server only. XDBDocument(byte[] source);Populates Document from source. Can be used in server only. XDBDocument(Connection conn);Opens connection for caching Document source. XDBDocument(Connection conn, byte[] source); Connection for caching bytes for Document source. XDBDocument(Connection conn, String source);Opens connection for caching string containing XML text. XDBDocument(String source);The string containing XML text. Can be used in server only. Parameters: source - Contains XML text., conn -Connection to be used.

Table E-1 Java DOM API for XMLType (mostly oracle.xdb.dom) Classes (Cont.)

Java DOM API for XMLType	Description
XDBDomImplementation	Implements <code>org.w3c.dom.DomImplementation</code> . Methods: <code>XDBDomImplementation()</code> - Opens a JDBC connection to the server.
XDBElement	Implements <code>org.w3c.dom.Element</code> .
XDBEntity	Implements <code>org.w3c.dom.Entity</code> .
XDBNodeMap	Implements <code>org.w3c.dom.NamedNodeMap</code> .
XDBNode	Implements <code>org.w3c.dom.Node</code> , the W3C DOM Node interface for interacting with XOBs. Methods: <code>write()</code> -Writes XML for this Node and all subnodes to an <code>OutputStream</code> . If the <code>OutputStream</code> is <code>ServletOutputStream</code> , the servlet output is committed and data is written using native streaming. <code>public void write(OutputStream s, String charEncoding, short indent);</code> Parameters: <code>s</code> - stream to write the output to <code>Contains XML text</code> <code>charEncoding</code> - IANA char code (for example, "ISO-8859") <code>indent</code> - number of characters to indent nested elements
XDBNodeList	Implements <code>org.w3c.dom.NodeList</code> .
XDBNotation	Implements <code>org.w3c.dom.Notation</code> .
XDBProcInst	Implements <code>org.w3c.dom.ProcInst</code> , the W3C DOM <code>ProcessingInstruction</code> interface.
XDBText	Implements <code>org.w3c.dom.Text</code> .

Table E-1 Java DOM API for XMLType (mostly oracle.xdb.dom) Classes (Cont.)

Java DOM API for XMLType	Description
XMLType (package oracle.xdb)	<p>Implements Java methods for the SQL type <code>SYS.XMLTYPE</code>.</p> <p>Methods:</p> <p><code>createXML()</code> - Creates an <code>XMLType</code>. Use this method when accessing data through JDBC.</p> <p><code>getStringVal()</code> - Retrieves string value containing the XML data from the <code>XMLType</code></p> <p><code>getClobVal()</code> - Retrieves the CLOB value containing the XML data from the <code>XMLType</code></p> <p><code>extract()</code> - Extracts the given set of nodes from the <code>XMLType</code></p> <p><code>existsNode()</code> - Checks for the existence of the given set of nodes in the <code>XMLType</code></p> <p><code>transform()</code> - Transforms the <code>XMLType</code> using the given XSL document</p> <p><code>isFragment()</code> - Checks if the <code>XMLType</code> is a regular document or a document fragment</p> <p><code>getDOM()</code> - Retrieves the DOM document associated with the <code>XMLType</code>.</p>
<code>createXML()</code>	<p>Creates an <code>XMLType</code>. Throws <code>java.sql.SQLException</code> if the <code>XMLType</code> could not be created:</p> <p><code>public static XMLType createXML(OPAQUE opq);</code> Creates and returns an <code>XMLType</code> given the opaque type containing the <code>XMLType</code> bytes.</p> <p><code>public static XMLType createXML(Connection conn, String xmlval);</code> Creates and returns an <code>XMLType</code> given the string containing the XML data.</p> <p><code>public static XMLType createXML(Connection conn, CLOB xmlval);</code> Creates and returns an <code>XMLType</code> given a CLOB containing the XML data.</p> <p><code>public static XMLType createXML(Connection conn, Document domdoc);</code> Creates and returns an <code>XMLType</code> given an instance of the DOM document.</p> <p>Parameters:</p> <p><code>opq</code> - opaque object from which the <code>XMLType</code> is to be constructed</p> <p><code>conn</code> - connection object to be used, <code>xmlval</code> - contains the XML data</p> <p><code>domdoc</code> - the DOM Document which represents the DOM tree,</p>
<code>getStringVal()</code>	<p>Retrieves the string value containing the XML data from the <code>XMLType</code>. Throws <code>java.sql.SQLException</code>.</p> <p><code>public String getStringVal();</code></p>
<code>getClobVal()</code>	<p>Retrieves the CLOB value containing the XML data from the <code>XMLType</code>. Throws <code>java.sql.SQLException</code></p> <p><code>public CLOB getClobVal();</code></p>

Table E-1 Java DOM API for XMLType (mostly oracle.xdb.dom) Classes (Cont.)

Java DOM API for XMLType	Description
extract()	<p>Extracts and returns the given set of nodes from the XMLType. The set of nodes is specified by the XPath expression. The original XMLType remains unchanged. Works only in the thick case. If no nodes match the specified expression, returns NULL. Throws <code>java.sql.SQLException</code></p> <pre>public XMLType extract(String xpath, String nsmap);</pre> <p>Parameters:</p> <p>xpath - xpath expression which specifies for which nodes to search</p> <p>nsmap - map of namespaces which resolves the prefixes in the xpath expression; format is "xmlns=a.com xmlns:b=b.com"</p>
existsNode()	<p>Checks for existence of given set of nodes in the XMLType. This set of nodes is specified by the xpath expression. Returns TRUE if specified nodes exist in the XMLType; otherwise, returns FALSE. Throws <code>java.sql.SQLException</code></p> <pre>public boolean existsNode(String xpath, String nsmap);</pre> <p>Parameters:</p> <p>xpath - xpath expression that specifies for which nodes to search</p> <p>nsmap - map of namespaces that resolves prefixes in the xpath expression;format is "xmlns=a.com xmlns:b=b.com",</p>
transform()	<p>Transforms and returns the XMLType using the given XSL document. The new (transformed) XML document is returned. Throws <code>java.sql.SQLException</code>.</p> <pre>public XMLType transform(XMLType xsldoc, String parammap);</pre> <p>Parameters:</p> <p>xsldoc - The XSL document to be applied to the XMLType</p> <p>parammap - top level parameters to be passed to the XSL transformation. Use format "a=b c=d e=f". Can be NULL.</p>
isFragment()	<p>Checks if the XMLType is a regular document or document fragment. Returns TRUE if doc is a fragment; otherwise, returns FALSE. Throws <code>java.sql.SQLException</code>.</p> <pre>public boolean isFragment();</pre>
getDOM()	<p>Retrieves the DOM document associated with the XMLType. This document is the <code>org.w3c.dom.Document</code>. The caller can perform all DOM operations on the Document. If the document is binary, <code>getDOM</code> returns NULL. Throws <code>java.sql.SQLException</code>.</p> <pre>public org.w3c.dom.Document getDOM();</pre>

Java Bean API for XMLType

The package `oracle.xml.db.bean` implements the Java Bean API for `XMLType`. It maps between XML schema and SQL and Java entities and datatypes, and Java Bean classes.

[Table E-2](#) lists the mapping from XML schema entities to Java Bean classes and methods.

See Also: [Chapter 9, "Java and Java Bean APIs for XMLType"](#)

Table E-2 Mapping From XML Schema Entities to Java Bean Classes

XML Schema Entity	Java Bean Class	Description
Attribute	<code>Get/setAttribute{attrname}</code>	Gets/sets the attribute for the document child element and its specified attribute.
Complextype	<code>Get/set{complextype classname}</code> .	For complex children, separate bean classes get generated. Extras (processing instructions or comments). DOM classes for processing instructions and comments are returned.
Scalar data	<code>Get/set{scalar type name}</code>	Children with <code>maxOccurs > 1</code> Get/set List of type of child.

[Table E-3](#) lists the mapping used by the Java Bean API for `XMLType` between XML Schema, SQL, and Java datatypes.

Table E-3 Mapping From XML Schema to SQL and Java Datatypes

XML Schema Datatype	SQL Datatype	Java Bean Datatype
Boolean	boolean	boolean
String	URI reference	ID
IDREF	ENTITY	NOTATION
Language	NCName	Name
<code>java.lang.String</code>	String	DECIMAL
INTEGER	LONG	SHORT
INT	POSITIVEINTEGER	NONPOSITIVEINTEGER
<code>oracle.sql.Number</code>	int	FLOAT

Table E-3 Mapping From XML Schema to SQL and Java Datatypes (Cont.)

XML Schema Datatype	SQL Datatype	Java Bean Datatype
DOUBLE	oracle.sql.Number	float
TIMEDURATION	TIMEPERIOD	RECURRINGDURATION
DATE	TIME	MONTH, YEAR
RECURRINGDATE	java.sql.Timestamp	Time
REF	oracle.sql.Ref	Ref
BINARY	oracle.sql.RAW	Byte[]
QNAME	java.lang.String	String

Oracle XML DB Resource API for Java/JNDI

Oracle XML DB Resource API for Java/JNDI is implemented using package `oracle.xml.db.spi` classes that render the service provider interface (SPI) drivers. These provide the application with common access to Java Naming and Directory Interface (JNDI).

Classes in `oracle.xml.db.spi` implement core JNDI SPI interfaces and WebDAV support for Oracle XML DB. [Table E-4](#) lists the `oracle.xml.db.spi` classes.

See Also: [Chapter 17, "Oracle XML DB Resource API for Java/JNDI"](#)

Table E-4 Oracle XML DB Resource API for Java/JNDI (oracle.xdb.spi)

oracle.xdb.spi Class	Description
XDBContext Class	<p>Implements the Java naming and context interface for Oracle XML DB, which extends <code>javax.naming.context</code>. In this release there is no federation support, in other words, it is completely unaware of the existence of other namespaces.</p> <p>Methods:</p> <p>XDBContext() - Class XDBContext constructor.</p> <ul style="list-style-type: none"> ▪ <code>public XDBContext(Hashtable env);</code> Creates an instance of XDBContext class given the environment. ▪ <code>public XDBContext(Hashtable env, String path);</code> Creates an instance of XDBContext class given the environment and path. <p>Parameters: <code>env</code> - Environment to describe properties of context, <code>path</code> - Initial path for the context.</p>
XDBContextFactory Class	<p>Implements <code>javax.naming.context</code>.</p> <p>Methods:</p> <p>XDBContextFactory() - Constructor for class XDBContextFactory. <code>public XDBContextFactory();</code></p>
XDBNameParser Class	Implements <code>javax.naming.NameParser</code>
XDBNamingEnumeration Class	Implements <code>javax.naming.NamingEnumeration</code>

Table E-4 Oracle XML DB Resource API for Java/JNDI (oracle.xdb.spi) (Cont.)

oracle.xdb.spi Class	Description
XDBResource Class	<p>Implements the core features for Oracle XML DB JNDI service provider interface (SPI). This release has no federation support, and is unaware of the existence of other namespaces. public class XDBResource extends java.lang.Object.</p> <p>Methods:</p> <p>XDBResource() - Creates a new instance of XDBResource</p> <p>getAuthor() - Returns author of the resource</p> <p>getComment() - Returns the DAV comment of the resource</p> <p>getContent() - Returns the content of the resource</p> <p>getContentType() - Returns the content type of the resource</p> <p>getCreateDate() - Returns the create date of the resource</p> <p>getDisplayName() - Returns the display name of the resource</p> <p>getLanguage() - Returns the language of the resource</p> <p>getLastModDate() - Returns the last modification date of the resource</p> <p>getOwnerId() - Returns the owner ID of the resource</p> <p>setACL() - Sets the ACL on the resource</p> <p>setAuthor() - Sets the author of the resource</p> <p>setComment() - Sets the DAV comment of the resource</p> <p>setContent() - Sets the content of the resource</p> <p>setContentType() - Sets the content type of the resource</p> <p>setCreateDate() - Sets the creation date of the resource</p> <p>setDisplayName() - Sets the display name of the resource</p> <p>setLanguage() - Sets the language of the resource</p> <p>setLastModDate() - Sets the last modification date of the resource</p> <p>setOwnerId() - Sets the owner ID of the resource</p>
XDBResource()	<p>Creates a new instance of XDBResource. public Creates a new instance of XDBResource given the environment.</p> <p>public XDBResource(Hashtable env, String path); Creates a new instance of XDBResource given the environment and path.</p> <p>Parameters: env - Environment passed in, path - Path to the resource</p>

Table E-4 Oracle XML DB Resource API for Java/JNDI (oracle.xdb.spi) (Cont.)

oracle.xdb.spi Class	Description
getAuthor()	Retrieves the author of the resource. public String getAuthor();
getComment()	Retrieves the DAV (Web Distributed <u>A</u> uthoring and <u>V</u> ersioning) comment of the resource. public String getComment();
getContent()	Returns the content of the resource. public Object getContent();
getContentType()	Returns the content type of the resource. public String getContentType();
getCreateDate()	Returns the creation date of the resource. public Date getCreateDate();
getDisplayName()	Returns the display name of the resource. public String getDisplayName();
getLanguage()	Returns the Language of the resource. public String getLanguage();
getLastModDate()	Returns the last modification date of the resource. public Date getLastModDate();
getOwnerId()	Returns the owner id of the resource. The value expected by this method is the user id value for the database user as provided by the catalog views such as ALL_USERS, and so on. public long getOwnerId();
setACL()	Sets the ACL on the resource. public void setACL(String aclpath); Parameters: aclpath - The path to the ACL resource.
setAuthor()	Sets the author of the resource. public void setAuthor(String authname); Parameter: authname - Author of the resource.
setComment()	Sets the DAV (Web Distributed <u>A</u> uthoring and <u>V</u> ersioning) comment of the resource. public void setComment(String davcom); Parameter: davcom - DAV comment of the resource.
setContent()	Sets the content of the resource. public void setContent(Object xmlobj); Parameter: xmlobj - Content of the resource.

Table E-4 Oracle XML DB Resource API for Java/JNDI (oracle.xdb.spi) (Cont.)

oracle.xdb.spi Class	Description
setContentType()	Sets the content type of the resource. public void setContentType(String conttype); Parameter: conttype - Content type of the resource.
setCreateDate()	Sets the creation date of the resource. public void setCreateDate(Date createDate); Parameter: createDate - Creation date of the resource.
setDisplayName()	Sets the display name of the resource. public void setDisplayName(String dname); Parameter: dname - Display name of the resource.
setLanguage()	Sets the language of the resource. public void setLanguage(String lang); Parameter: lang - Language of the resource.
setLastModDate()	Sets the last modification date of the resource. public void - setLastModDate(Date d); Parameter: d - Last modification date of the resource.
setOwnerId()	Sets the owner id of the resource. The owner id value expected by this method is the user id value for the database user as provided by the catalog views such as ALL_USERS, and so on. public void setOwnerId(long ownerId); Parameters: ownerId - Owner id of the resource.

Oracle XML DB XMLType API, PL/SQL and Resource PL/SQL APIs: Quick Reference

This appendix provides a summary of the following Oracle XML DB SQL and PL/SQL APIs:

- XMLType API
- PL/SQL DOM API for XMLType (DBMS_XMLDOM)
- PL/SQL Parser for XMLType (DBMS_XMLPARSER)
- PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)
- DBMS_XMLSCHEMA
- Oracle XML DB XML Schema Catalog Views
- Resource API for PL/SQL (DBMS_XDB)
- RESOURCE_VIEW, PATH_VIEW
- DBMS_XDB_VERSION
- DBMS_XDBT

XMLType API

`XMLType` is a system-defined opaque type for handling XML data. `XMLType` has predefined member functions to extract XML nodes and fragments. You can create columns of `XMLType` and insert XML documents into them. You can also generate XML documents as `XMLType` instances dynamically using SQL functions, `SYS_XMLGEN` and `SYS_XMLAGG`, the PL/SQL package `DBMS_XMLGEN`, and the `SQLX` functions.

[Table F-1](#) lists the `XMLType` API functions.

See Also:

- *Oracle9i XML API Reference - XDK and Oracle XML DB*
- [Chapter 4, "Using XMLType"](#)

Table F-1 XMLType API

Function	Description
<p>XMLType()</p> <p>constructor function XMLType(xmlData IN clob, schema IN varchar2 := NULL, validated IN number := 0, wellformed IN Number := 0) return self as result</p> <p>constructor function XMLType(xmlData IN varchar2, schema IN varchar2 := NULL, validated IN number := 0, wellformed IN number := 0) return self as result</p> <p>constructor function XMLType (xmlData IN "<ADT_1>", schema IN varchar2 := NULL, element IN varchar2 := NULL, validated IN number := 0) return self as result</p> <p>onstructor function XMLType(xmlData IN SYS_REFCURSOR, schema in varchar2 := NULL, element in varchar2 := NULL, validated in number := 0) return self as result</p>	<p>Constructor that constructs an instance of the XMLType datatype. The constructor can take in the XML as a CLOB, VARCHAR2 or take in a object type.</p> <p>Parameters:</p> <p>xmlData - data in the form of a CLOB, REF cursor, VARCHAR2 or object type.</p> <p>schema - optional schema URL used to make the input conform to the given schema.</p> <p>validated - flag to indicate that the instance is valid according to the given XMLSchema. (default 0)</p> <p>wellformed - flag to indicate that the input is wellformed. If set, then the database would not do well formed check on the input instance. (default 0)</p> <p>element - optional element name in the case of the ADT_1 or REF CURSOR constructors. (default null)</p>
--	--
<p>createXML()</p> <p>STATIC FUNCTION createXML(xmlval IN varchar2) RETURN XMLType deterministic</p> <p>STATIC FUNCTION createXML(xmlval IN clob) RETURN XMLType</p> <p>STATIC FUNCTION createXML (xmlData IN clob, schema IN varchar2, validated IN number := 0, wellformed IN number := 0) RETURN XMLType deterministic</p> <p>STATIC FUNCTION createXML (xmlData IN varchar2, schema IN varchar2, validated IN number := 0, wellformed IN number := 0) RETURN XMLType deterministic</p> <p>STATIC FUNCTION createXML (xmlData IN "<ADT_1>", schema IN varchar2 := NULL, element IN varchar2 := NULL, validated IN NUMBER := 0) RETURN XMLType deterministic</p> <p>STATIC FUNCTION createXML (xmlData IN SYS_REFCURSOR, schema in varchar2 := NULL, element in varchar2 := NULL, validated in number := 0) RETURN XMLType deterministic</p>	<p>Static function for creating and returning an XMLType instance. The string and clob parameters used to pass in the data must contain well-formed and valid XML documents. The options are described in the following table.</p> <p>Parameters:</p> <p>xmlData - Actual data in the form of a CLOB, REF cursor, VARCHAR2 or object type.</p> <p>schema - optional Schema URL to be used to make the input conform to the given schema.</p> <p>validated - flag to indicate that the instance is valid according to the given XMLSchema. (default 0)</p> <p>wellformed - flag to indicate that the input is wellformed. If set, then the database would not do well formed check on the input instance. (default 0)</p> <p>element - optional element name in the case of the ADT_1 or REF CURSOR constructors. (default null)</p>

Table F-1 XMLType API (Cont.)

Function	Description
<p>existsNode()</p> <p>MEMBER FUNCTION existsNode(xpath IN varchar2) RETURN number deterministic</p> <p>MEMBER FUNCTION existsNode(xpath in varchar2, nsmmap in varchar2) RETURN number deterministic</p>	<p>Takes an XMLType instance and a XPath and returns 1 or 0 indicating if applying the XPath returns a non-empty set of nodes. If the XPath string is NULL or the document is empty, then a value of 0 is returned, otherwise returns 1.</p> <p>Parameters:</p> <p>xpath - XPath expression to test.</p> <p>nsmmap - optional namespace mapping.</p>
<p>extract()</p> <p>MEMBER FUNCTION extract(xpath IN varchar2) RETURN XMLType deterministic</p> <p>MEMBER FUNCTION extract(xpath IN varchar2, nsmmap IN varchar2) RETURN XMLType deterministic</p>	<p>Extracts an XMLType fragment and returns an XMLType instance containing the result node(s). If the XPath does not result in any nodes, it returns NULL.</p> <p>Parameters:</p> <p>xpath - XPath expression to apply.</p> <p>nsmmap - optional prefix to namespace mapping information.</p>
<p>isFragment()</p> <p>MEMBER FUNCTION isFragment() RETURN number deterministic</p>	<p>Determines if the XMLType instance corresponds to a well-formed document, or a fragment. Returns 1 or 0 indicating if the XMLType instance contains a fragment or a well-formed document. Returns 1 or 0 indicating if the XMLType instance contains a fragment or a well-formed document..</p>
<p>getClobVal()</p> <p>MEMBER FUNCTION getClobVal() RETURN clob deterministic</p>	<p>Returns a CLOB containing the serialized XML representation; if the return is a temporary CLOB, it must be freed after use.</p>
<p>getNumVal()</p> <p>MEMBER FUNCTION getNumVal() RETURN number deterministic</p>	<p>Returns a numeric value, formatted from the text value pointed to by the XMLType instance. The XMLType must point to a valid text node that contains a numerical value.</p>
<p>getStringVal()</p> <p>MEMBER FUNCTION getStringVal() RETURN varchar2 deterministic</p>	<p>Returns the document as a string containing the serialized XML representation, or for text nodes, the text itself. If the XML document is bigger than the maximum size of the VARCHAR2, 4000, then an error is raised at run time.</p>
<p>transform()</p> <p>MEMBER FUNCTION transform(xsl IN XMLType, parammap in varchar2 := NULL) RETURN XMLType deterministic</p>	<p>Transforms XML data using the XSL stylesheet argument and the top-level parameters passed as a string of name=value pairs. If any argument other than the parammap is NULL, a NULL is returned.</p> <p>Parameter</p> <p>xsl - XSL stylesheet describing the transformation</p> <p>parammap - top level parameters to the XSL - string of name=value pairs</p>

Table F-1 XMLType API (Cont.)

Function	Description
<p>toObject()</p> <p>MEMBER PROCEDURE toObject(SELF in XMLType, object OUT "<ADT_1>", schema in varchar2 := NULL, element in varchar2 := NULL)</p>	<p>Converts XML data into an instance of a user defined type, using the optional schema and top-level element arguments.</p> <p>Parameters:</p> <p>SELF - instance to be converted. Implicit if used as a member procedure.</p> <p>object - converted object instance of the required type may be passed in to this function.</p> <p>schema - schema URL. Mapping of the XMLType instance to the converted object instance can be specified using a schema.</p> <p>element - top-level element name. This specifies the top-level element name in the XMLSchema document to map the XMLType instance.</p>
<p>isSchemaBased()</p> <p>MEMBER FUNCTION isSchemaBased return number deterministic</p>	<p>Determines if the XMLType instance is schema-based. Returns 1 or 0 depending on whether the XMLType instance is schema-based or not.</p>
<p>getSchemaURL()</p> <p>MEMBER FUNCTION getSchemaURL return varchar2 deterministic</p>	<p>Returns the XML schema URL corresponding to the XMLType instance, if the XMLType instance is a schema-based document. Otherwise returns NULL.</p>
<p>getRootElement()</p> <p>MEMBER FUNCTION getRootElement return varchar2 deterministic</p>	<p>Gets the root element of the XMLType instance. Returns NULL if the instance is a fragment.</p>
<p>createSchemaBasedXML()</p> <p>MEMBER FUNCTION createSchemaBasedXML(schema IN varchar2 := NULL) return sys.XMLType deterministic</p>	<p>Creates a schema-based XMLType instance from a non-schema-based XML and a schemaURL.</p> <p>Parameter:</p> <p>schema - schema URL If NULL, then the XMLType instance must contain a schema URL.</p>
<p>createNonSchemaBasedXML()</p> <p>MEMBER FUNCTION createNonSchemaBasedXML return XMLType deterministic</p>	<p>Creates a non-schema-based XML document from an XML schema-based instance.</p>
<p>getNamespace()</p> <p>MEMBER FUNCTION getNamespace return varchar2 deterministic</p>	<p>Returns the namespace of the top level element in the instance. NULL if the input is a fragment or is a non-schema-based instance.</p>
<p>schemaValidate()</p> <p>MEMBER PROCEDURE schemaValidate</p>	<p>Validates the XML instance against its schema if it has not already validated. For non-schema based documents an error is raised. If validation fails an error is raised; else, the document's status is changed to validated.</p>

Table F-1 XMLType API (Cont.)

Function	Description
isSchemaValidated() MEMBER FUNCTION isSchemaValidated return NUMBER deterministic	Returns the validation status of the XMLType instance if it has been validated against its schema. Returns 1 if validated against the schema, 0 otherwise.
setSchemaValidated() MEMBER PROCEDURE setSchemaValidated(flag IN BINARY_INTEGER := 1)	Sets the VALIDATION state of the input XML instance to avoid schema validation. Parameter: flag - 0 = NOT VALIDATED; 1 = VALIDATED; Default value is 1.
isSchemaValid() member function isSchemaValid(schurl IN VARCHAR2 := NULL, elem IN VARCHAR2 := NULL) return NUMBER deterministic	Checks if the input instance conforms to a specified schema. Does not change validation status of the XML instance. If an XML schema URL is not specified and the XML document is schema-based, conformance is checked against the XMLType instance's own schema. Parameter: schurl - URL of the XML Schema against which to check conformance. elem - Element of a specified schema, against which to validate. Useful when you have an XML Schema that defines more than one top level element, and you need to check conformance against a specific elements.

PL/SQL DOM API for XMLType (DBMS_XMLDOM)

Table F-2 lists the PL/SQL DOM API for XMLType (DBMS_XMLDOM) functions and procedures. These are grouped according to the W3C DOM Recommendation.

See Also: [Chapter 8, "PL/SQL API for XMLType"](#)

Table F-2 DBMS_XMLDOM Functions and Procedures

Functions and Procedures	Description
DOM Node	Implements the DOM Node interface.
isNull()	Tests if the node is NULL.
makeAttr()	Casts the node to an Attribute.
makeCDataSection()	Casts the node to a CDataSection.
makeCharacterData()	Casts the node to CharacterData.
makeComment()	Casts the node to a Comment.
makeDocumentFragment()	Casts the node to a DocumentFragment.

Table F–2 DBMS_XMLDOM Functions and Procedures (Cont.)

Functions and Procedures	Description
makeDocumentType()	Casts the node to a Document Type.
makeElement()	Casts the node to an Element.
makeEntity()	Casts the node to an Entity.
makeEntityReference()	Casts the node to an EntityReference.
makeNotation()	Casts the node to a Notation.
makeProcessingInstruction()	Casts the node to a DOMProcessingInstruction.
makeText()	Casts the node to a DOMText.
makeDocument()	Casts the node to a DOMDocument.
writeToFile()	Writes the contents of the node to a file.
writeToBuffer()	Writes the contents of the node to a buffer.
writeToClob()	Writes the contents of the node to a clob.
getNodeName()	Retrieves the Name of the node.
getNodeValue()	Retrieves the Value of the node.
setNodeValue()	Sets the Value of the node.
getNodeType()	Retrieves the Type of the node.
getParentNode()	Retrieves the parent of the node.
getChildNodes()	Retrieves the children of the node.
getFirstChild()	Retrieves the first child of the node.
getLastChild()	Retrieves the last child of the node.
getPreviousSibling()	Retrieves the previous sibling of the node.
getNextSibling()	Retrieves the next sibling of the node.
getAttributes()	Retrieves the attributes of the node.
getOwnerDocument()	Retrieves the owner document of the node.
insertBefore()	Inserts a child before the reference child.
replaceChild()	Replaces the old child with a new child.
removeChild()	Removes a specified child from a node.
appendChild()	Appends a new child to the node.

Table F–2 DBMS_XMLDOM Functions and Procedures (Cont.)

Functions and Procedures	Description
hasChildNodes()	Tests if the node has child nodes.
cloneNode()	Clones the node.
DOM Named Node Map	Implements the DOM NamedNodeMap interface.
isNull()	Tests if the NamedNodeMap is NULL .
getNamedItem()	Retrieves the item specified by the name.
setNamedItem()	Sets the item in the map specified by the name.
removeNamedItem()	Removes the item specified by name.
item()	Retrieves the item given the index in the map.
getLength()	Retrieves the number of items in the map.
DOM Node List	Implements the DOM NodeList interface.
isNull()	Tests if the Nodelist is NULL .
item()	Retrieves the item given the index in the nodelist.
getLength()	Retrieves the number of items in the list.
DOM Attr	Implements the DOM Attribute interface.
isNull()	Tests if the Attribute Node is NULL .
makeNode()	Casts the Attribute to a node.
getQualifiedName()	Retrieves the Qualified Name of the attribute.
getNamespace()	Retrieves the NS URI of the attribute.
getLocalName()	Retrieves the local name of the attribute.
getExpandedName()	Retrieves the expanded name of the attribute.
getName()	Retrieves the name of the attribute.
getSpecified()	Tests if attribute was specified in the owning element.
getValue()	Retrieves the value of the attribute.
setValue()	Sets the value of the attribute.
DOM CData Section	Implements the DOM CDataSection interface.
isNull()	Tests if the CDataSection is NULL .
makeNode()	Casts the CDataSection to a node.

Table F–2 DBMS_XMLDOM Functions and Procedures (Cont.)

Functions and Procedures	Description
DOM Character Data	Implements the DOM Character Data interface.
isNull()	Tests if the <code>CharacterData</code> is <code>NULL</code> .
makeNode()	Casts the <code>CharacterData</code> to a node.
getData()	Retrieves the data of the node.
setData()	Sets the data to the node.
getLength()	Retrieves the length of the data.
substringData()	Retrieves the substring of the data.
appendData()	Appends the given data to the node data.
insertData()	Inserts the data in the node at the given <code>offsets</code> .
deleteData()	Deletes the data from the given <code>offsets</code> .
replaceData()	Replaces the data from the given <code>offsets</code> .
DOM Comment	Implements the DOM Comment interface.
isNull()	Tests if the comment is <code>NULL</code> .
makeNode()	Casts the <code>Comment</code> to a node.
DOM Implementation	Implements the DOM DOMImplementation interface.
isNull()	Tests if the <code>DOMImplementation</code> node is <code>NULL</code> .
hasFeature()	Tests if the <code>DOMImplementation</code> implements a given feature.
DOM Document Fragment	Implements the DOM DocumentFragment interface.
isNull()	Tests if the <code>DocumentFragment</code> is <code>NULL</code> .
makeNode()	Casts the <code>DocumentFragment</code> to a node.
DOM Document Type	Implements the DOM Document Type interface.
isNull()	Tests if the document type is <code>NULL</code> .
makeNode()	Casts the document type to a node.
findEntity()	Finds the specified entity in the document type.
findNotation()	Finds the specified notation in the document type.

Table F–2 DBMS_XMLDOM Functions and Procedures (Cont.)

Functions and Procedures	Description
getPublicId()	Retrieves the public ID of the document type.
getSystemId()	Retrieves the system ID of the document type.
writeExternalDTDToFile()	Writes the document type definition to a file.
writeExternalDTDToBuffer()	Writes the document type definition to a buffer.
writeExternalDTDToClob()	Writes the document type definition to a CLOB.
getName()	Retrieves the name of the Document type.
getEntities()	Retrieves the nodemap of entities in the Document type.
getNotations()	Retrieves the nodemap of the notations in the Document type.
DOM Element	Implements the DOM Element interface.
isNull()	Tests if the Element is NULL .
makeNode()	Casts the Element to a node.
getQualifiedName()	Retrieves the qualified name of the element.
getNamespace()	Retrieves the NS URI of the element.
getLocalName()	Retrieves the local name of the element.
getExpandedName()	Retrieves the expanded name of the element.
getChildrenByTagName()	Retrieves the children of the element by tag name.
getElementsByTagName()	Retrieves the elements in the subtree by tagname.
resolveNamespacePrefix()	Resolve the prefix to a namespace uri.
getTagName()	Retrieves the Tag name of the element.
getAttribute()	Retrieves the attribute node specified by the name.
setAttribute()	Sets the attribute specified by the name.
removeAttribute()	Removes the attribute specified by the name.
getAttributeNode()	Retrieves the attribute node specified by the name.
setAttributeNode()	Sets the attribute node in the element.
removeAttributeNode()	Removes the attribute node in the element.
normalize()	Normalizes the text children of the element.

Table F–2 DBMS_XMLDOM Functions and Procedures (Cont.)

Functions and Procedures	Description
DOM Entity	Implements the DOM Entity interface.
isNull()	Tests if the Entity is NULL.
makeNode()	Casts the Entity to a node.
getPublicId()	Retrieves the public Id of the entity.
getSystemId()	Retrieves the system Id of the entity.
getNotationName()	Retrieves the notation name of the entity.
DOM Entity Reference	Implements the DOM EntityReference interface.
isNull()	Tests if the Entity Reference is NULL.
makeNode()	Casts the Entity Reference to NULL.
DOM Notation	Implements the DOM Notation interface.
isNull()	Tests if the Notation is NULL.
makeNode()	Casts the notation to a node.
getPublicId()	Retrieves the public Id of the notation.
getSystemId()	Retrieves the system Id of the notation.
DOM Processing Instruction	Implements the DOM Processing instruction interface.
isNull()	Tests if the Processing Instruction is NULL.
makeNode()	Casts the Processing Instruction to a node.
getData()	Retrieves the data of the processing instruction.
getTarget()	Retrieves the target of the processing instruction.
setData()	Sets the data of the processing instruction.
DOM Text	Implements the DOM Text interface.
isNull()	Tests if the text is NULL.
makeNode()	Casts the text to a node.
splitText()	Splits the contents of the text node into 2 text nodes.
DOM Document	Implements the DOM Document interface.
isNull()	Tests if the document is NULL.

Table F-2 DBMS_XMLDOM Functions and Procedures (Cont.)

Functions and Procedures	Description
makeNode()	Casts the document to a node.
newDOMDocument()	Creates a new Document.
freeDocument()	Frees the document.
getVersion()	Retrieves the version of the document.
setVersion()	Sets the version of the document.
getCharSet()	Retrieves the Character set of the document.
setCharSet()	Sets the Character set of the document.
getStandalone()	Retrieves if the document is specified as standalone.
setStandalone()	Sets the document standalone.
writeToFile()	Writes the document to a file.
writeToBuffer()	Writes the document to a buffer.
writeToClob()	Writes the document to a CLOB..
writeExternalDTDToFile()	Writes the DTD of the document to a file.
writeExternalDTDToBuffer()	Writes the DTD of the document to a buffer.
writeExternalDTDToClob()	Writes the DTD of the document to a CLOB..
getDoctype()	Retrieves the DTD of the document.
getImplementation()	Retrieves the DOM implementation.
getDocumentElement()	Retrieves the root element of the document.
createElement()	Creates a new Element.
createDocumentFragment()	Creates a new Document Fragment.
createTextNode()	Creates a Text node.
createComment()	Creates a Comment node.
createCDATASection()	Creates a CDATAsection node.
createProcessingInstruction()	Creates a Processing Instruction.
createAttribute()	Creates an Attribute.
createEntityReference()	Creates an Entity reference.
getElementsByTagName()	Retrieves the elements in the by tag name.

PL/SQL Parser for XMLType (DBMS_XMLPARSER)

You can access the content and structure of XML documents through the PL/SQL Parser for XMLType (DBMS_XMLPARSER).

[Table F-3](#) lists the PL/SQL Parser for XMLType (DBMS_XMLPARSER) functions and procedures.

See Also: [Chapter 8, "PL/SQL API for XMLType"](#)

Table F-3 *DBMS_XMLPARSER Functions and Procedures*

Functions/Procedures	Description
parse()	Parses XML stored in the given url/file.
newParser()	Returns a new parser instance
parseBuffer()	Parses XML stored in the given buffer
parseClob()	Parses XML stored in the given clob
parseDTD()	Parses DTD stored in the given url/file
parseDTDBuffer()	Parses DTD stored in the given buffer
parseDTDClob()	Parses DTD stored in the given clob
setBaseDir()	Sets base directory used to resolve relative URLs.
showWarnings()	Turns warnings on or off.
setErrorLog()	Sets errors to be sent to the specified file
setPreserveWhitespace()	Sets white space preserve mode
setValidationMode()	Sets validation mode.
getValidationMode()	Returns validation mode.
setDoctype()	Sets DTD.
getDoctype()	Gets DTD Parser.
getDocument()	Gets DOM document.
freeParser()	Frees a parser object.
getReleaseVersion()	Returns the release version of Oracle XML Parser for PL/SQL.

PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)

This PL/SQL implementation of the XSL processor follows the W3C XSLT Working Draft (Rev WD-xslt-19990813).

[Table F-4](#) summarizes the PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) functions and procedures.

See Also: [Chapter 8, "PL/SQL API for XMLType"](#)

Table F-4 *PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) Functions*

Functions and Procedures	Description
<code>newProcessor()</code>	Returns a new processor instance.
<code>processXSL()</code>	Transforms an input XML document.
<code>showWarnings()</code>	Turns warnings on or off.
<code>setErrorLog()</code>	Sets errors to be sent to the specified file.
<code>newStylesheet()</code>	Creates a new stylesheet using the given input and reference URLs.
<code>transformNode()</code>	Transforms a node in a DOM tree using the given stylesheet.
<code>selectNodes()</code>	Selects nodes from a DOM tree that match the given pattern.
<code>selectSingleNode()</code>	Selects the first node from the tree that matches the given pattern.
<code>valueOf()</code>	Retrieves the value of the first node from the tree that matches the given pattern
<code>setParam()</code>	Sets a top-level parameter in the stylesheet
<code>removeParam()</code>	Removes a top-level stylesheet parameter
<code>resetParams()</code>	Resets the top-level stylesheet parameters
<code>freeStylesheet()</code>	Frees a stylesheet object
<code>freeProcessor()</code>	Frees a processor object

DBMS_XMLSCHEMA

This package is created by `dbmsxsch.sql` during the Oracle XML DB installation. It provides procedures for registering and deleting your XML schemas. [Table F-5](#) summarizes the DBMS_XMLSCHEMA functions and procedures.

See Also: [Chapter 5, "Structured Mapping of XMLType"](#)

Table F-5 DBMS_XMLSCHEMA Functions and Procedures

Constant	Description
<pre> registerSchema() procedure registerSchema(schemaURL IN VARCHAR2, schemaDoc IN VARCHAR2, local IN BOOLEAN := TRUE, genTypes IN BOOLEAN := TRUE, genbean IN BOOLEAN := FALSE, genTables IN BOOLEAN := TRUE, force IN BOOLEAN := FALSE, owner IN VARCHAR2 := null); procedure registerSchema(schemaURL IN VARCHAR2, schemaDoc IN CLOB, local IN BOOLEAN := TRUE, genTypes IN BOOLEAN := TRUE, genbean IN BOOLEAN := FALSE, force IN BOOLEAN := FALSE, owner IN VARCHAR2 := null); procedure registerSchema(schemaURL IN VARCHAR2, schemaDoc IN BFILE, local IN BOOLEAN := TRUE, genTypes IN BOOLEAN := TRUE, genbean IN BOOLEAN := FALSE, force IN BOOLEAN := FALSE, owner IN VARCHAR2 := null); procedure registerSchema(schemaURL IN VARCHAR2, schemaDoc IN SYS.XMLType, local IN BOOLEAN := TRUE, genTypes IN BOOLEAN := TRUE, genbean IN BOOLEAN := FALSE, force IN BOOLEAN := FALSE, owner IN VARCHAR2 := null); procedure registerSchema(schemaURL IN VARCHAR2, schemaDoc IN SYS.URIType, local IN BOOLEAN := TRUE, genTypes IN BOOLEAN := TRUE, genbean IN BOOLEAN := FALSE, force IN BOOLEAN := FALSE, owner IN VARCHAR2 := null); </pre>	<p>Registers the specified XML schema for use by Oracle XML DB. This schema can then be used to store documents that conform to it.</p> <p>Parameters:</p> <p>schemaURL - URL that uniquely identifies the schema document. This value is used to derive the path name of the schema document within the XDB hierarchy.</p> <p>schemaDoc - a valid XML schema document</p> <p>local - is this a local or global schema? By default, all schemas are registered as local schemas i.e. under /sys/schemas/<username/... If a schema is registered as global, it is added under /sys/schemas/PUBLIC/.... You need write privileges on the above directory to be able to register a schema as global.</p> <p>genTypes - should the schema compiler generate object types? By default, TRUE</p> <p>genbean - should the schema compiler generate Java beans? By default, FALSE.</p> <p>genTables - sShould the schema compiler generate default tables? By default, TRUE</p> <p>force - if this parameter is set to TRUE, the schema registration will not raise errors. Instead, it creates an invalid XML schema object in case of any errors. By default, the value of this parameter is FALSE.</p> <p>owner - specifies the name of the database user owning the XML schema object. By default, the user registering the XML schema owns the XML schema object. Can be used to register an XML schema to be owned by a different database user.</p>

Table F-5 DBMS_XMLSCHEMA Functions and Procedures (Cont.)

Constant	Description
<pre> registerURI() procedure registerURI(schemaURL IN varchar2, schemaDocURI IN varchar2, local IN BOOLEAN := TRUE, genTypes IN BOOLEAN := TRUE, genbean IN BOOLEAN := FALSE, genTables IN BOOLEAN := TRUE, force IN BOOLEAN := FALSE, owner IN VARCHAR2 := null); </pre>	Registers an XML schema specified by a URI name.
<pre> deleteSchema() procedure deleteSchema(schemaURL IN varchar2, delete_option IN pls_ integer := DELETE_RESTRICT); </pre>	Removes the XML schema from Oracle XML DB.
<pre> generateBean() procedure generateBean(schemaURL IN varchar2); </pre>	Generates the Java Bean code corresponding to a registered XML schema.
<pre> compileSchema() procedure compileSchema(schemaURL IN varchar2); </pre>	Recompiles an already registered XML schema. Useful for bringing an invalid schema to a valid state.
<pre> generateSchema() function generateSchemas(schemaName IN varchar2, typeName IN varchar2, elementName IN varchar2 := NULL, schemaURL IN varchar2 := NULL, annotate IN BOOLEAN := TRUE, embedColl IN BOOLEAN := TRUE) return sys.XMLSequenceType; function generateSchema(schemaName IN varchar2, typeName IN varchar2, elementName IN varchar2 := NULL, recurse IN BOOLEAN := TRUE, annotate IN BOOLEAN := TRUE, embedColl IN BOOLEAN := TRUE) return sys.XMLType; </pre>	Generates XML schema(s) from an Oracle type name.

DBMS_XMLSCHEMA constants:

- DELETE_RESTRICT, CONSTANT NUMBER := 1;

- DELETE_INVALIDATE, CONSTANT NUMBER := 2;
- DELETE_CASCADE, CONSTANT NUMBER := 3;
- DELETE_CASCADE_FORCE, CONSTANT NUMBER := 4;

Oracle XML DB XML Schema Catalog Views

[Table F-6](#) lists the Oracle XML DB XML schema catalog views.

Table F-6 Oracle XML DB: XML Schema Catalog View

Schema	Description
USER_XML_SCHEMAS	Lists all registered XML Schemas owned by the user.
ALL_XML_SCHEMAS	Lists all registered XML Schemas usable by the current user.
DBA_XML_SCHEMAS	Lists all registered XML Schemas in Oracle XML DB.
DBA_XML_TABLES	Lists all XMLType tables in the system.
USER_XML_TABLES	Lists all XMLType tables owned by the current user.
ALL_XML_TABLES	Lists all XMLType tables usable by the current user.
DBA_XML_TAB_COLS	Lists all XMLType table columns in the system.
USER_XML_TAB_COLS	Lists all XMLType table columns in tables owned by the current user.
ALL_XML_TAB_COLS	Lists all XMLType table columns in tables usable by the current user.
DBA_XML_VIEWS	Lists all XMLType views in the system.
USER_XML_VIEWS	Lists all XMLType views owned by the current user.
ALL_XML_VIEWS	Lists all XMLType views usable by the current user.
DBA_XML_VIEW_COLS	Lists all XMLType view columns in the system.
USER_XML_VIEW_COLS	Lists all XMLType view columns in views owned by the current user.
ALL_XML_VIEW_COLS	Lists all XMLType view columns in views usable by the current user.

Resource API for PL/SQL (DBMS_XDB)

Resource API for PL/SQL (DBMS_XDB) PL/SQL package provides functions for the following Oracle XML DB tasks:

- Resource management of Oracle XML DB hierarchy. These functions complement the functionality provided by Resource Views.
- Oracle XML DB's Access Control List (ACL) for security management. The ACL-based security mechanism can be used for either:
 - In-hierarchy ACLs, ACLs stored through Oracle XML DB resource API
 - In-memory ACLs, that can be stored outside Oracle XML DB.

Some of these methods can be used for both Oracle XML DB resources and arbitrary database objects. `AcLCheckPrivileges()` enables database users access to Oracle XML DB ACL-based security mechanism without having to store their objects in the Oracle XML DB hierarchy.

- Oracle XML DB configuration session management.
- Rebuilding of hierarchical indexes.

[Table F-7](#) summarizes the DBMS_XDB functions and procedures.

See Also: [Chapter 16, "Oracle XML DB Resource API for PL/SQL \(DBMS_XDB\)"](#)

Table F-7 DBMS_XDB Functions and Procedures

Function/Procedure	Description
<code>getAcIDocument()</code> FUNCTION <code>getAcIDocument(</code> <code>abspath IN VARCHAR2) RETURN</code> <code>sys.xmltype;</code>	Retrieves ACL document that protects resource given its pathname.
<code>getPrivileges()</code> FUNCTION <code>getPrivileges(res_path</code> <code>IN VARCHAR2) RETURN</code> <code>sys.xmltype;</code>	Gets all privileges granted to the current user on the given XDB resource.
<code>changePrivileges()</code> FUNCTION <code>changePrivileges(res_</code> <code>path IN VARCHAR2, ace IN</code> <code>xmltype) RETURN</code> <code>pls_integer;</code>	Adds the given ACE to the given resource's ACL.

Table F-7 DBMS_XDB Functions and Procedures (Cont.)

Function/Procedure	Description
checkPrivileges() FUNCTION checkPrivileges(res_ path IN VARCHAR2, privs IN xmltype) RETURN pls_integer;	Checks access privileges granted to the current user on the specified XDB resource.
setacl() PROCEDURE setacl(res_path IN VARCHAR2, acl_path IN VARCHAR2);	Sets the ACL on the given XDB resource to be the ACL specified.
AclCheckPrivileges() FUNCTION AclCheckPrivileges(acl_path IN VARCHAR2, owner IN VARCHAR2, privs IN xmltype) RETURN pls_integer;	Checks access privileges granted to the current user by specified ACL document on a resource whose owner is specified by the 'owner' parameter.
LockResource() FUNCTION LockResource(path IN VARCHAR2, depthzero IN BOOLEAN, shared IN boolean) RETURN BOOLEAN;	Gets a WebDAV-style lock on that resource given a path to that resource.
GetLockToken() PROCEDURE GetLockToken(path IN VARCHAR2, locktoken OUT VARCHAR2);	Returns that resource's lock token for the current user given a path to a resource.
UnlockResource() FUNCTION UnlockResource(path IN VARCHAR2, deltoken IN VARCHAR2) RETURN BOOLEAN;	Unlocks the resource given a lock token and a path to the resource.

Table F-7 DBMS_XDB Functions and Procedures (Cont.)

Function/Procedure	Description
CreateResource() FUNCTION CreateResource(path IN VARCHAR2,data IN VARCHAR2) RETURN BOOLEAN; FUNCTION CreateResource(path IN VARCHAR2, data IN SYS.XMLTYPE) RETURN BOOLEAN; FUNCTION CreateResource(path IN VARCHAR2, datarow IN REF SYS.XMLTYPE) RETURN BOOLEAN; FUNCTION CreateResource(path IN VARCHAR2, data IN CLOB) RETURN BOOLEAN; FUNCTION CreateResource(path IN VARCHAR2, data IN BFILE) RETURN BOOLEAN;	Creates a new resource.
CreateFolder() FUNCTION CreateFolder(path IN VARCHAR2) RETURN BOOLEAN;	Creates a new folder resource in the hierarchy.
DeleteResource() PROCEDURE DeleteResource(path IN VARCHAR2);	Deletes a resource from the hierarchy.
Link() PROCEDURE Link(srcpath IN VARCHAR2, linkfolder IN VARCHAR2, linkname IN VARCHAR2);	Creates a link to an existing resource.
CFG_Refresh() PROCEDURE CFG_Refresh;	Refreshes the session's configuration information to the latest configuration.
CFG_Get() FUNCTION CFG_Get RETURN SYS.XMLType;	Retrieves the session's configuration information.
CFG_Update() PROCEDURE CFG_Update(xdbconfig IN SYS.XMLTYPE);	Updates the configuration information.

DBMS_XMLGEN

PL/SQL package `DBMS_XMLGEN` transforms SQL query results into a canonical XML format. It inputs an arbitrary SQL query, converts it to XML, and returns the result as a CLOB. `DBMS_XMLGEN` is similar to the `DBMS_XMLQUERY`, except that it is written in C and compiled in the kernel. This package can only be run in the database.

[Table F-8](#) summarizes the `DBMS_XMLGEN` functions and procedures.

See Also: [Chapter 10, "Generating XML Data from the Database"](#)

Table F-8 *DBMS_XMLGEN Functions and Procedures*

Function/Procedure	Description
<code>newContext()</code>	Creates a new context handle.
<code>setRowTag()</code>	Sets the name of the element enclosing each row of the result. The default tag is <code>ROW</code> .
<code>setRowSetTag ()</code>	Sets the name of the element enclosing the entire result. The default tag is <code>ROWSET</code> .
<code>getXML()</code>	Gets the XML document.
<code>getNumRowsProcessed()</code>	Gets the number of SQL rows that were processed in the last call to <code>getXML</code> .
<code>setMaxRows()</code>	Sets the maximum number of rows to be fetched each time.
<code>setSkipRows()</code>	Sets the number of rows to skip every time before generating the XML. The default is 0.
<code>setConvertSpecialChars()</code>	Sets whether special characters such as \$, which are non-XML characters, should be converted or not to their escaped representation. The default is to perform the conversion.
<code>convert()</code>	Converts the XML into the escaped or unescaped XML equivalent.
<code>useItemTagsForColl()</code>	Forces the use of the collection column name appended with the tag <code>_ITEM</code> for collection elements. The default is to set the underlying object type name for the base element of the collection.
<code>restartQUERY()</code>	Restarts the query to start fetching from the beginning.

Table F-8 DBMS_XMLGEN Functions and Procedures (Cont.)

Function/Procedure	Description
closeContext()	Closes the context and releases all resources.

RESOURCE_VIEW, PATH_VIEW

Oracle XML DB RESOURCE_VIEW and PATH_VIEW provide a mechanism for SQL-based access of data stored in the Oracle XML DB Repository. Data stored in the Oracle XML DB Repository through protocols such as FTP, WebDAV, or JNDI API can be accessed in SQL through RESOURCE and PATH VIEWS.

Oracle XML DB Resource API for PL/SQL is based on RESOURCE_VIEW, PATH_VIEW and some PL/SQL packages. It provides query and DML functionality. PATH_VIEW has one row for each unique path in the Repository, whereas RESOURCE_VIEW has one row for each resource in the Repository.

[Table F-9](#) summarizes the Oracle XML DB Resource API for PL/SQL operators.

See Also: [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)

Table F-9 RESOURCE_VIEW, PATH_VIEW Operators

Operator	Description
UNDER_PATH INTEGER UNDER_PATH(resource_column, pathname); INTEGER UNDER_PATH(resource_ column, depth, pathname); INTEGER UNDER_PATH(resource_ column, pathname, correlation) INTEGER UNDER_PATH(resource_ column, depth, pathname, correlation)	Using the Oracle XML DB hierarchical index, returns sub-paths of a particular path. Parameters: resource_column - column name or column alias of the 'resource' column in the path_view or resource_view. pathname - path name to resolve. depth - maximum depth to search; a depth of less than 0 is treated as 0. correlation - integer that can be used to correlate the UNDER_PATH operator (a primary operator) with ancillary operators (PATH and DEPTH).
EQUALS_PATH EQUALS_PATH INTEGER EQUALS_PATH(resource_column, pathname);	Finds the resource with the specified pathname.

Table F–9 RESOURCE_VIEW, PATH_VIEW Operators (Cont.)

Operator	Description
PATH PATH VARCHAR2 PATH(correlation);	Returns the relative pathname of the resource under the specified pathname argument.
DEPTH DEPTH INTEGER DEPTH(correlation);	Returns the folder depth of the resource under the specified starting path.

DBMS_XDB_VERSION

DBMS_XDB_VERSION along with DBMS_XDB implement the Oracle XML DB versioning API.

Table F–10 summarizes the DBMS_XDB_VERSION functions and procedures.

See Also: [Chapter 14, "Oracle XML DB Versioning"](#)

Table F–10 DBMS_XDB_VERSION Functions and Procedures

Function/Procedure	Description
MakeVersioned() FUNCTION MakeVersioned(pathname VARCHAR2) RETURN dbms_xdb.resid_type;	Turns a regular resource whose pathname is given into a version-controlled resource.
Checkout() PROCEDURE Checkout(pathname VARCHAR2);	Checks out a VCR before updating or deleting it.
Checkin() FUNCTION Checkin(pathname VARCHAR2) RETURN dbms_ xdb.resid_type;	Checks in a checked-out VCR and returns the resource id of the newly-created version.
Uncheckout() FUNCTION Uncheckout(pathname VARCHAR2) RETURN dbms_ xdb.resid_type;	Checks in a checked-out resource and returns the resource id of the version before the resource is checked out.
GetPredecessors() FUNCTION GetPredecessors(pathname VARCHAR2) RETURN resid_list_type;	Retrieves the list of predecessors by pathname.

Table F–10 DBMS_XDB_VERSION Functions and Procedures (Cont.)

Function/Procedure	Description
GetPredsByResId() FUNCTION GetPredsByResId(resid resid_type) RETURN resid_list_type;	Retrieves the list of predecessors by resource id.
GetResourceByResId() FUNCTION GetResourceByResId(resid resid_type) RETURN XMLType;	Obtains the resource as an XMLType, given the resource objectID.
GetSuccessors() FUNCTION GetSuccessors(pathname VARCHAR2) RETURN resid_list_type;	Retrieves the list of successors by pathname.
GetSuccsByResId() FUNCTION GetSuccsByResId(resid resid_type) RETURN resid_list_type;	Retrieves the list of successors by resource id.

DBMS_XDBT

Using DBMS_XDBT you can set up an Oracle Text `ConText` index on the Oracle XML DB Repository hierarchy. DBMS_XDBT creates default preferences and the Oracle Text index. It also sets up automatic synchronization of the `ConText` index.

DBMS_XDBT contains variables that describe the onfiguration settings for the `ConText` index. These are intended to cover the basic customizations that installations may require, but they are not a complete set.

Use DBMS_XDBT for the following tasks:

- To customize the package to set up the appropriate configuration
- To drop existing index preferences using `dropPreferences()`
- To create new index preferences using `createPreferences()`
- To create the `ConText` index using the `createIndex()`
- To set up automatic synchronization of the `ConText` index using the `configureAutoSync()`

Table F–11 summarizes the DBMS_XDBT functions and procedures.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB*

Table F-11 DBMS_XDBT Functions and Procedures

Procedure/Function	Description
<code>dropPreferences()</code>	Drops any existing preferences.
<code>createPreferences()</code>	Creates preferences required for the <code>ConText</code> index on the XML DB hierarchy.
<code>createDatastorePref()</code>	Creates a USER datastore preference for the <code>ConText</code> index.
<code>createFilterPref()</code>	Creates a filter preference for the <code>ConText</code> index.
<code>createLexerPref()</code>	Creates a lexer preference for the <code>ConText</code> index.
<code>createWordlistPref()</code>	Creates a stoplist for the <code>ConText</code> index.
<code>createStoplistPref()</code>	Creates a section group for the <code>ConText</code> index.
<code>createStoragePref()</code>	Creates a wordlist preference for the <code>ConText</code> index.
<code>createSectiongroupPref()</code>	Creates a storage preference for the <code>ConText</code> index.
<code>createIndex()</code>	Creates the <code>ConText</code> index on the XML DB hierarchy.
<code>configureAutoSync()</code>	Configures the <code>ConText</code> index for automatic maintenance (<code>SYNC</code>).

Example Setup scripts. Oracle XML DB-Supplied XML Schemas

This appendix describes a few example setup scripts for use with the examples in [Chapter 3, "Using Oracle XML DB"](#). It also includes the structure of Resource View and Path View and the Oracle XML DB supplied XML schema:

- [Example Setup Scripts](#)
- [Resource View and Path View Database and XML Schema](#)
- [XDBResource.xsd: XML Schema for Representing Oracle XML DB Resources](#)
- [acl.xsd: XML Schema for Representing Oracle XML DBACLs](#)
- [xdbconfig.xsd: XML Schema for Configuring Oracle XML DB](#)

Example Setup Scripts

The following sections include the setup scripts and sample files used for the examples in [Chapter 3, "Using Oracle XML DB"](#).

Chapter 3 Examples Set Up Script: Creating User and Directory

```
set echo on
connect / as sysdba
drop directory DIR;
drop user &1 cascade;
create user &1 identified by &2;
grant create any directory, drop any directory to &1;
grant connect, resource to &1;
connect &1/&2
create or replace function getFileContent(file bfile)
return CLOB deterministic
is
    charContent    CLOB := ' ';
    targetFile     bfile;
    warning        number;
begin
    targetFile := file;
    DBMS_LOB.fileopen(targetFile, DBMS_LOB.file_readonly);
    DBMS_LOB.loadFromFile(charContent, targetFile,
    DBMS_LOB.getLength(targetFile), 1, 1);
    DBMS_LOB.fileclose(targetFile);
    return charContent;
end;
/
show errors;
drop directory DIR;
create directory DIR as '&3';
create or replace function getDocument(filename varchar2)
return CLOB deterministic
is
    file           bfile := bfilename('DIR', filename);
    charContent    CLOB := ' ';
    targetFile     bfile;
    warning        number;
begin
    targetFile := file;
    DBMS_LOB.fileopen(targetFile, DBMS_LOB.file_readonly);
    DBMS_LOB.loadFromFile(charContent, targetFile,
    DBMS_LOB.getLength(targetFile), 1, 1);
```

```

        DBMS_LOB.fileclose(targetFile);
        return charContent;
    end;
/
show errors
declare
    result boolean;
begin
    result := dbms_xdb.createfolder('/public/&4');
end;
/
commit;
quit

```

Chapter 3 Examples Set Up Script: Granting Privileges, Creating Table...

This script grants appropriate privileges, creates tables with XMLType Columns, creates XMLType tables, Inserts, queries, and updates the tables:

```

set echo on
connect scott/tiger
grant all on emp to &1;
connect &1/&2
--
-- Table Creation Examples
--
create table EXAMPLE1
(
    KEYVALUE varchar2(10) primary key,
    XMLCOLUMN xmltype
);
create table XMLTABLE of XMLType;
--
-- Insert Example
--
describe getDocument;
insert into XMLTABLE
values (xmltype(getDocument('purchaseorder.xml')))
/
commit
/
--
-- Valid existsNode operations
--
select existsNode(value(X), '/PurchaseOrder/Reference')

```

```

from XMLTABLE X
/
select existsNode(value(X),
                  '/PurchaseOrder[Reference="ADAMS-20011127121040988PST"]')
from XMLTABLE X
/
select existsNode(value(X),
                  '/PurchaseOrder/LineItems/LineItem[2]/Part[@Id="037429135020"]')
from XMLTABLE X
/
select existsNode(value(X),
                  '/PurchaseOrder/LineItems/LineItem[Description="8 1/2"]')
from XMLTABLE X
/
--
-- Invalid existsNode() operations
--
select existsNode(value(X), '/PurchaseOrder/UserName')
from XMLTABLE X
/
select existsNode(value(X),
                  '/PurchaseOrder[Reference="ADAMS-XXXXXXXXXXXXXXXXXXXXX"]')
from XMLTABLE X
/
select existsNode(value(X),
                  '/PurchaseOrder/LineItems/LineItem[3]/Part[@Id="037429135020"]')
from XMLTABLE X
/
select existsNode(value(X),
                  '/PurchaseOrder/LineItems/LineItem[Description="Snow White"]')
from XMLTABLE X
/
--
-- existsNode() in where clause examples
--
select count(*)
from XMLTABLE x
where existsNode(value(x), '/PurchaseOrder[User="ADAMS"]') = 1
/
delete from XMLTABLE x
where existsNode(value(x), '/PurchaseOrder[User="ADAMS"]') = 1
/
commit
/
--

```

```

-- Reload the Sample Document.
--
insert into XMLTABLE
values (xmltype(getDocument('purchaseorder.xml')))
/
commit
/
--
-- Valid extractValue() operations
--
select extractValue(value(x), '/PurchaseOrder/Reference')
from XMLTABLE X
/
select extractValue(value(x),
                    '/PurchaseOrder/LineItems/LineItem[2]/Part/@Id')
from XMLTABLE X
/
--
-- Invalid extractValue() operations
--
select extractValue(value(X),
                    '/PurchaseOrder/LineItems/LineItem/Description')
from XMLTABLE X
/
select extractValue(value(X),
                    '/PurchaseOrder/LineItems/LineItem[1]')
from XMLTABLE X
/
select extractValue(value(x), '/PurchaseOrder/Reference')
from XMLTABLE X, SCOTT.EMP
where extractValue(value(x), '/PurchaseOrder/User') = EMP.ENAME
and EMP.EMPNO = 7876
/
--
-- extract() operations
--
set long 10000
select extract(value(X),
              '/PurchaseOrder/LineItems/LineItem/Description')
from XMLTABLE X
/
select extract(value(X),
              '/PurchaseOrder/LineItems/LineItem[1]')
from XMLTABLE X
/

```

```

set long 10000
set feedback on
select extractValue(value(t), '/Description')
from XMLTABLE X,
table ( xmlsequence (
          extract(value(X),
                  '/PurchaseOrder/LineItems/LineItem/Description')
        )
) t
/
update XMLTABLE t
set value(t) = updateXML(value(t),
'/PurchaseOrder/Reference/text()',
'MILLER-200203311200000000PST')
where existsNode(value(t),
                  '/PurchaseOrder[Reference="ADAMS-20011127121040988PST" ]') = 1
/
select value(t)
from XMLTABLE t
/
update XMLTABLE t
set value(t) =
updateXML(value(t),
          '/PurchaseOrder/LineItems/LineItem[2]',
          xmltype('<LineItem ItemNumber="4">
                  <Description>Andrei Rublev</Description>
                  <Part Id="715515009928" UnitPrice="39.95"
                  Quantity="2"/>
                  </LineItem>'
          )
)
where existsNode(value(t),
                  '/PurchaseOrder[Reference="MILLER-200203311200000000PST" ]'
) = 1
/
select value(t)
from XMLTABLE t
where existsNode(value(t),
                  '/PurchaseOrder[Reference="MILLER-200203311200000000PST" ]'
) = 1
/
select value(t).transform(xmltype(getDocument('purchaseOrder.xml')))
from XMLTABLE t
where existsNode(value(t),
                  '/PurchaseOrder[Reference="MILLER-200203311200000000PST" ]'

```

```
) = 1
/
begin
  dbms_xmlschema.registerSchema(
    'http://www.oracle.com/xsd/purchaseOrder.xsd',
    getDocument('purchaseOrder.xsd'),
    TRUE, TRUE, FALSE, FALSE
  );
end;
/
create table XML_PURCHASEORDER of XMLType
XMLSCHEMA "http://www.oracle.com/xsd/purchaseOrder.xsd"
ELEMENT "PurchaseOrder"
/
describe XML_PURCHASEORDER
insert into XML_PURCHASEORDER
values (xmltype(getDocument('Invoice.xml')))
/
alter table XML_PURCHASEORDER
add constraint VALID_PURCHASEORDER
check (XMLIsValid(sys_nc_rowinfo$)=1)
/
insert into XML_PURCHASEORDER
values (xmltype(getDocument('InvalidPurchaseOrder.xml')))
/
alter table XML_PURCHASEORDER
drop constraint VALID_PURCHASEORDER
/
create trigger VALIDATE_PURCHASEORDER
before insert on XML_PURCHASEORDER
for each row
declare
  XMLDATA xmltype;
begin
  XMLDATA := :new.sys_nc_rowinfo$;
  xmltype.schemavalidate(XMLDATA);
end;
/
insert into XML_PURCHASEORDER
values (xmltype(getDocument('InvalidPurchaseOrder.xml')))
/
drop table XML_PURCHASEORDER;
begin
  dbms_xmlSchema.deleteSchema('http://www.oracle.com/xsd/purchaseOrder.xsd',4);
end;
```

```

/
begin
  dbms_xmlschema.registerSchema(
    'http://www.oracle.com/xsd/purchaseOrder.xsd',
    getDocument('purchaseOrder1.xsd'),
    TRUE, TRUE, FALSE, FALSE
  );
end;
/
describe XML_PURCHASEORDER_TYPE
drop table XML_PURCHASEORDER;
begin
  dbms_xmlSchema.deleteSchema('http://www.oracle.com/xsd/purchaseOrder.xsd',4);
end;
/
begin
  dbms_xmlschema.registerSchema(
    'http://www.oracle.com/xsd/purchaseOrder.xsd',
    getDocument('purchaseOrder2.xsd'),
    TRUE, TRUE, FALSE, FALSE
  );
end;
/
describe XML_PURCHASEORDER_TYPE
quit

```

Loading Files

```

set echo on
connect &1/&2
declare
  result boolean;
begin
  result := dbms_xdb.createResource('/public/&3/&4',
    getFileContent(bfilename('DIR','&4')));
end;
/
commit;
quit

```

Chapter 3 Examples Script: invoice.xml

```

<Invoice
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:noNamespaceSchemaLocation="http://www.oracle.com/xsd/purchaseOrder.xsd">
<Reference>ADAMS-20011127121040988PST</Reference>
<Actions>
  <Action>
    <User>SCOTT</User>
    <Date xsi:nil="true"/>
  </Action>
</Actions>
<Reject/>
<Requestor>Julie P. Adams</Requestor>
<CostCenter>R20</CostCenter>
<ShippingInstructions>
  <name>Julie P. Adams</name>
  <address>300 Oracle Parkway, Redwood Shores, CA 94065</address>
  <telephone>650 506 7300</telephone>
</ShippingInstructions>
<SpecialInstructions>Ground</SpecialInstructions>
<LineItems>
  <LineItem ItemNumber="1">
    <Description>The Ruling Class</Description>
    <Part Id="715515012423" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>Diabolique</Description>
    <Part Id="037429135020" UnitPrice="29.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>8 1/2</Description>
    <Part Id="037429135624" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
</LineItems>
</Invoice>

```

Chapter 3 Examples Script: PurchaseOrder.xml

```

<PurchaseOrder
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.oracle.com/xdb/po.xsd">
  <Reference>ADAMS-20011127121040988PST</Reference>
  <Actions>
    <Action>
      <User>SCOTT</User>
      <Date xsi:nil="true"/>
    </Action>
  </Actions>

```

```
<Reject/>
<Requestor>Julie P. Adams</Requestor>
<User>ADAMS</User>
<CostCenter>R20</CostCenter>
<ShippingInstructions>
  <name>Julie P. Adams</name>
  <address>300 Oracle Parkway, Redwood Shores, CA 94065</address>
  <telephone>650 506 7300</telephone>
</ShippingInstructions>
<SpecialInstructions>Ground</SpecialInstructions>
<LineItems>
  <LineItem ItemNumber="1">
    <Description>The Ruling Class</Description>
    <Part Id="715515012423" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>Diabolique</Description>
    <Part Id="037429135020" UnitPrice="29.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>8 1/2</Description>
    <Part Id="037429135624" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
</LineItems>
</PurchaseOrder>
```

Note:

- The example XML schema, "[XML Schema Example, PurchaseOrder.xsd](#)" is located in [Appendix B, "XML Schema Primer"](#).
 - The example XSL file, "[XSL Stylesheet Example, PurchaseOrder.xsl](#)" is located in [Appendix D, "XSLT Primer"](#).
-
-

Chapter 3 Examples Script: FTP Script

```
#!/usr/bin/ksh
TESTDIR=$1
TESTFILENAME=$2
. ./config.sh
SCRIPTFILE=`date +%Y%m%d%H%M%S`
SCRIPTFILE=/tmp/$SCRIPTFILE.cmd
mkdir /tmp/$TESTDIR
```

```
echo "open $ORAHOSTNAME $ORAFTPPORT" > $SCRIPTFILE
echo "user $ORASQLUSER $ORASQLPASSWORD" >> $SCRIPTFILE
echo "cd public" >> $SCRIPTFILE
echo "cd $TESTDIR" >> $SCRIPTFILE
echo "put $TESTFILENAME" >> $SCRIPTFILE
echo "ls -l" >> $SCRIPTFILE
echo "get $TESTFILENAME /tmp/$TESTDIR/$TESTFILENAME" >> $SCRIPTFILE
echo "quit" >> $SCRIPTFILE
ftp -v -n < $SCRIPTFILE
rm $SCRIPTFILE
echo "Diff Results for $TESTFILENAME"
diff -b $TESTFILENAME /tmp/$TESTDIR/$TESTFILENAME
rm -rf /tmp/$TESTDIR
```

Chapter 3 Examples Script: Configuring FTP and HTTP Ports

```
#!/usr/bin/ksh
ORAHOSTNAME='hostname'
ORAFTPPORT=2100
ORAHHTTPPORT=8080

if [ "$LOGNAME" = "oracle2" ]
then
    ORAFTPPORT=2122
    ORAHHTTPPORT=8088
fi

echo "FTP Port = $ORAFTPPORT"
echo "HTTP Port = $ORAHHTTPPORT"

ORASQLUSER=DOC92
ORASQLPASSWORD=DOC92
```

Resource View and Path View Database and XML Schema

The following describes the Resource View and Path View structures.

Resource View Definition and Structure

The Resource View contains one row for each resource in the Repository. The following describes its structure:

Column	Datatype	Description
RES	XMLTYPE	A resource in Oracle XML Repository
ANY_PATH	VARCHAR2	A path that can be used to access the resource in the Repository

See Also: [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)

Path View Definition and Structure

The Path View contains one row for each unique path in the Repository. The following describes its structure:

Column	Datatype	Description
PATH	VARCHAR2	Path name of a resource
RES	XMLTYPE	The resource referred by PATH
LINK	XMLTYPE	Link property

See Also: [Chapter 15, "RESOURCE_VIEW and PATH_VIEW"](#)

XDBResource.xsd: XML Schema for Representing Oracle XML DB Resources

Here is the listing for the XML schema, `XDBResource.xsd`, used to represent Oracle XML DB resources.

XDBResource.xsd

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://xmlns.oracle.com/xdb/XDBResource.xsd"
version="1.0" elementFormDefault="qualified"
xmlns:res="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="LockScopeType">
  <restriction base="string">
    <enumeration value="Exclusive" fixed="false"/>
    <enumeration value="Shared" fixed="false"/>
  </restriction>
</simpleType>

<complexType name="LockType" mixed="false">
  <sequence>
    <element name="owner" type="string"/>
    <element name="expires" type="dateTime"/>
  </sequence>
</complexType>
```

```
        <element name="lockToken" type="hexBinary"/>
    </sequence>
    <attribute name="LockScope" type="res:LockScopeType" />
</complexType>

<complexType name="ResContentsType" mixed="false">
    <sequence >
        <any name="ContentsAny" />
    </sequence>
</complexType>

<complexType name="ResAclType" mixed="false">
    <sequence >
        <any name="ACLAny" />
    </sequence>
</complexType>

<complexType name="ResourceType" mixed="false">
    <sequence >
        <element name="CreationDate" type="dateTime"/>
        <element name="ModificationDate" type="dateTime"/>
        <element name="Author" type="res:ResMetaStr"/>
        <element name="DisplayName" type="res:ResMetaStr"/>
        <element name="Comment" type="res:ResMetaStr"/>
        <element name="Language" type="res:ResMetaStr"/>
        <element name="CharacterSet" type="res:ResMetaStr"/>
        <element name="ContentType" type="res:ResMetaStr"/>
        <element name="RefCount" type="nonNegativeInteger"/>
        <element name="Lock" type="res:LocksRaw"/>
        <element name="ACL" type="res:ResAclType"
minOccurs="0" maxOccurs="1"/>
        <element name="Owner" type="res:OracleUserName"
minOccurs="0" maxOccurs="1"/>
        <element name="Creator" type="res:OracleUserName"
minOccurs="0" maxOccurs="1"/>
        <element name="LastModifier" type="res:OracleUserName"
minOccurs="0" maxOccurs="1"/>
        <element name="SchemaElement" type="res:SchElemType"
minOccurs="0" maxOccurs="1"/>
        <element name="Contents" type="res:ResContentsType"
minOccurs="0" maxOccurs="1"/>
        <element name="VCRUID" type="res:GUID"/>
        <element name="Parents" type="hexBinary" minOccurs="0"
maxOccurs="1000"/>
        <any name="ResExtra"
namespace="##other" minOccurs="0" maxOccurs="65535"/>
    </sequence>
</complexType>
```

```

    </sequence>

    <attribute name="Hidden" type="boolean"/>
    <attribute name="Invalid" type="boolean"/>
    <attribute name="VersionID" type="integer"/>
    <attribute name="ActivityID" type="integer"/>
    <attribute name="Container" type="boolean"/>
    <attribute name="CustomRslv" type="boolean"/>
  </complexType>

  <element name="Resource" type="res:ResourceType"/>
</schema>

```

acl.xsd: XML Schema for Representing Oracle XML DBACLs

This section describes the XML schema used to represent Oracle XML DB ACLs:

ACL Representation XML Schema, acl.xsd

Here is the listing for the XML schema, `acl.xsd`, used to represent Oracle XML DB ACLs:

acl.xsd

```

<schema xmlns="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://xmlns.oracle.com/xdb/acl.xsd"
version="1.0"
  xmlns:xdb="http://xmlns.oracle.com/xdb"
  xmlns:xdbacl="http://xmlns.oracle.com/xdb/acl.xsd"
  elementFormDefault="qualified">

  <annotation>
    <documentation>
      This XML schema describes the structure of XDB ACL
      documents.

      Note : The "systemPrivileges" element below lists
      all supported
      system privileges and their aggregations.
    </documentation>
  <appinfo>

```

```
<xdb:systemPrivileges>
  <xdbacl:all>
    <xdbacl:read-properties/>
    <xdbacl:read-contents/>
    <xdbacl:read-acl/>
    <xdbacl:update/>
    <xdbacl:bind/>
    <xdbacl:unbind/>
    <xdbacl:unlink/>
    <xdbacl:write-acl-ref/>
    <xdbacl:update-acl/>
    <xdbacl:link/>
    <xdbacl:resolve/>
  </xdbacl:all>
</xdb:systemPrivileges>
</appinfo>
</annotation>

<!-- privilegeNameType (this is an emptycontent type) -->
<complexType name = "privilegeNameType"/>

<!-- privilegeName element
  All system and user privileges are in the
substitutionGroup
  of this element.
-->
<element name = "privilegeName"
type="xdbacl:privilegeNameType"
  xdb:defaultTable="" />

<!-- all system privileges in the XDB ACL namespace -->
<element name = "read-properties"
type="xdbacl:privilegeNameType"
  substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
<element name = "read-contents"
type="xdbacl:privilegeNameType"
  substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
<element name = "read-acl" type="xdbacl:privilegeNameType"

  substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
<element name = "update" type="xdbacl:privilegeNameType"
  substitutionGroup="xdbacl:privilegeName"
```

```

xdb:defaultTable="" />
  <element name = "bind" type="xdbacl:privilegeNameType"
    substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
  <element name = "unbind" type="xdbacl:privilegeNameType"
    substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
  <element name = "unlink" type="xdbacl:privilegeNameType"
    substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
  <element name = "write-acl-ref"
type="xdbacl:privilegeNameType"
    substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
  <element name = "update-acl"
type="xdbacl:privilegeNameType"
    substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
  <element name = "link" type="xdbacl:privilegeNameType"
    substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
  <element name = "resolve" type="xdbacl:privilegeNameType"
    substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />
  <element name = "all" type="xdbacl:privilegeNameType"
    substitutionGroup="xdbacl:privilegeName"
xdb:defaultTable="" />

  <!-- privilege element -->
  <element name = "privilege" xdb:SQLType = "XDB$PRIV_T"
xdb:defaultTable="">
    <complexType> <sequence>
      <any maxOccurs="unbounded" />
    </sequence> </complexType>
  </element>

  <!-- ace element -->
  <element name = "ace" xdb:SQLType = "XDB$ACE_T"
xdb:defaultTable="">
    <complexType> <sequence>
      <element name = "grant" type = "boolean"/>
      <element name = "principal" type = "string"/>
      <element ref="xdbacl:privilege" minOccurs="1"/>
    </sequence> </complexType>
  </element>

```

```
<!-- acl element -->
<element name = "acl" xdb:SQLType = "XDB$ACL_T"
xdb:defaultTable = "XDB$ACL">
  <complexType>
    <sequence>
      <element name = "schemaURL" type = "string"
minOccurs="0"/>
      <element name = "elementName" type = "string"
minOccurs="0" />
      <element ref = "xdbacl:ace" minOccurs="1" maxOccurs =
"unbounded"
          xdb:SQLCollType="XDB$ACE_LIST_T" />
    </sequence>

    <attribute name = "shared" type = "boolean"
default="true"/>
    <attribute name = "description" type = "string"/>
  </complexType>
</element>

</schema>
```

xdbcconfig.xsd: XML Schema for Configuring Oracle XML DB

This section lists `xdbcconfig.xsd`, the XML schema for configuring Oracle XML DB:

xdbcconfig.xsd

```
<schema
targetNamespace="http://xmlns.oracle.com/xdb/xdbcconfig.xsd"
xmlns="http://www.w3.org/2001/XMLSchema"

xmlns:xdbc="http://xmlns.oracle.com/xdb/xdbcconfig.xsd"
xmlns:xdb="http://xmlns.oracle.com/xdb"
version="1.0" elementFormDefault="qualified">

  <element name="xdbcconfig" 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="positiveInteger"
default="1000"/>
        <element name="invalid-pathname-chars" type="string"
default=""/>
        <element name="case-sensitive" type="boolean"
default="true"/>
        <element name="call-timeout" type="unsignedInt"
default="300"/>
        <element name="max-link-queue" type="unsignedInt"
default="65536"/>
        <element name="max-session-use" type="unsignedInt"
default="100"/>
        <element name="persistent-sessions" type="boolean"
default="false"/>
        <element name="default-lock-timeout"
type="unsignedInt"
default="3600"/>
        <element name="xdbc-core-logfile-path" type="string"
default="/sys/log/xdblog.xml"/>
        <element name="xdbc-core-log-level" type="unsignedInt"
default="1"/>

        <!-- protocol specific properties -->
        <element name="protocolconfig">
          <complexType><sequence>

            <!-- these apply to all protocols -->
            <element name="common">
              <complexType><sequence>
                <element name="extension-mappings">
                  <complexType><sequence>
                    <element name="mime-mappings"
type="xdbc:mime-mapping-type"/>
                    <element name="lang-mappings"
type="xdbc:lang-mapping-type"/>
                    <element name="charset-mappings"

type="xdbc:charset-mapping-type"/>
                    <element name="encoding-mappings"

```

```
type="xdbc:encoding-mapping-type"/>
    </sequence></complexType>
  </element>
  <element name="session-pool-size"
type="unsignedInt"

default="50"/>
    <element name="session-timeout"
type="unsignedInt"

default="6000"/>

    </sequence></complexType>
  </element>

  <!-- FTP specific -->
  <element name="ftpconfig">
<complexType><sequence>
  <element name="ftp-port" type="unsignedShort"
default="2100"/>
  <element name="ftp-listener" type="string"/>
  <element name="ftp-protocol" type="string">
    <!--simpleType>
      <restriction base="string">
        <enumeration value="tcp"/>
        <enumeration value="tcps"/>
      </restriction>
    </simpleType-->
  </element>
  <element name="logfile-path" type="string"

default="/sys/log/ftplog.xml"/>
  <element name="log-level" type="unsignedInt"
default="1"/>
  <element name="session-timeout"
type="unsignedInt"

default="6000"/>
  </sequence></complexType>
</element>

  <!-- HTTP specific -->
  <element name="httpconfig">
<complexType><sequence>
  <element name="http-port"
```

```

type="unsignedShort" default="8080"/>
    <element name="http-listener"
type="string"/>
    <element name="http-protocol" type="string">
        <!--simpleType>
            <restriction base="string">
                <enumeration value="tcp"/>
                <enumeration value="tcps"/>
            </restriction>
        </simpleType-->
    </element>
    <element name="max-http-headers"
type="unsignedInt"
default="64"/>
    <element name="max-header-size"
type="unsignedInt"
default="4096"/>
    <element name="max-request-body"
type="unsignedInt"
default="2000000000" minOccurs="1"/>
    <element name="session-timeout"
type="unsignedInt"
default="6000"/>
    <element name="server-name" type="string"/>
    <element name="logfile-path" type="string"
default="/sys/log/httplog.xml"/>
    <element name="log-level" type="unsignedInt"
default="1"/>
    <element name="servlet-realm" type="string"
minOccurs="0"/>
    <element name="webappconfig">
    <complexType><sequence>
        <element name="welcome-file-list"
type="xdbc:welcome-file-type"/>
        <element name="error-pages"
type="xdbc:error-page-type"/>
    </sequence>
    </complexType>
    </element>
    <element name="servletconfig"

```

```

type="xdbc:servlet-config-type"/>
    </sequence></complexType>
  </element>
</sequence></complexType>
</element>

</sequence></complexType>
</element>

</sequence></complexType>
</element>

<!-- users can add any properties they want here -->
<element name="userconfig">
  <complexType><sequence>
    <any maxOccurs="unbounded" namespace="##other"/>
  </sequence></complexType>
</element>

</sequence></complexType>

</element>

<complexType name="welcome-file-type">
  <sequence>
    <element name="welcome-file" minOccurs="0"
maxOccurs="unbounded">
      <simpleType>
        <restriction base="string">
          <pattern value="^[^/]*"/>
        </restriction>
      </simpleType>
    </element>
  </sequence>
</complexType>

<!-- customized error pages -->
<complexType name="error-page-type">
  <sequence>
    <element name="error-page" minOccurs="0"
maxOccurs="unbounded">
      <complexType><sequence>
        <choice>
          <element name="error-code">

```

```

        <simpleType>
            <restriction base="positiveInteger">
                <minInclusive value="100" />
                <maxInclusive value="999" />
            </restriction>
        </simpleType>
    </element>

    <!-- Fully qualified classname of a Java
exception type -->
    <element name="exception-type"
type="string" />

    <element name="OracleError">
        <complexType><sequence>
            <element name="facility" type="string"
default="ORA" />
            <element name="errnum"
type="unsignedInt" />
        </sequence></complexType>
    </element>
</choice>

    <element name="location" type="anyURI" />

    </sequence></complexType>
</element>
</sequence>
</complexType>

    <!-- parameter for a servlet: name, value pair and a
description -->
    <complexType name="param">
        <sequence>
            <element name="param-name" type="string" />
            <element name="param-value" type="string" />
            <element name="description" type="string" />
        </sequence>
    </complexType>

    <complexType name="servlet-config-type">
        <sequence>
            <element name="servlet-mappings">
                <complexType><sequence>
                    <element name="servlet-mapping" minOccurs="0"

```

```

        maxOccurs="unbounded">
        <complexType><sequence>
            <element name="servlet-pattern"
type="string"/>
            <element name="servlet-name"
type="string"/>
        </sequence></complexType>
    </element>
</sequence></complexType>
</element>

    <element name="servlet-list">
        <complexType><sequence>
            <element name="servlet" minOccurs="0"
maxOccurs="unbounded">
                <complexType><sequence>
                    <element name="servlet-name"
type="string"/>
                    <element name="servlet-language">
                        <simpleType>
                            <restriction base="string">
                                <enumeration value="C"/>
                                <enumeration value="Java"/>
                                <enumeration value="PL/SQL"/>
                            </restriction>
                        </simpleType>
                    </element>
                    <element name="icon" type="string"
minOccurs="0"/>
                    <element name="display-name"
type="string"/>
                    <element name="description" type="string"
minOccurs="0"/>
                    <choice>
                        <element name="servlet-class"
type="string" minOccurs="0"/>
                        <element name="jsp-file" type="string"
minOccurs="0"/>
                    </choice>
                    <element name="servlet-schema"
type="string" minOccurs="0"/>
                    <element name="init-param" minOccurs="0"
maxOccurs="unbounded"
type="xdbc:param"/>
                    <element name="load-on-startup"

```

```

type="string" minOccurs="0"/>
      <element name="security-role-ref"
minOccurs="0"
      maxOccurs="unbounded">
    <complexType><sequence>
      <element name="description"
type="string" minOccurs="0"/>
      <element name="role-name"
type="string"/>
      <element name="role-link"
type="string"/>
    </sequence></complexType>
  </element>
</sequence></complexType>
</element>
</sequence></complexType>
</element>
</sequence>
</complexType>

  <complexType name="lang-mapping-type"><sequence>
    <element name="lang-mapping" minOccurs="0"
maxOccurs="unbounded">
      <complexType><sequence>
        <element name="extension" type="xdbc:exttype"/>
        <element name="lang" type="string"/>
      </sequence></complexType>
    </element></sequence>
  </complexType>

  <complexType name="charset-mapping-type"><sequence>
    <element name="charset-mapping" minOccurs="0"
maxOccurs="unbounded">
      <complexType><sequence>
        <element name="extension" type="xdbc:exttype"/>
        <element name="charset" type="string"/>
      </sequence></complexType>
    </element></sequence>
  </complexType>

  <complexType name="encoding-mapping-type"><sequence>
    <element name="encoding-mapping" minOccurs="0"
maxOccurs="unbounded">
      <complexType><sequence>
        <element name="extension" type="xdbc:exttype"/>

```

```

        <element name="encoding" type="string" />
    </sequence></complexType>
</element></sequence>
</complexType>

<complexType name="mime-mapping-type"><sequence>
    <element name="mime-mapping" minOccurs="0"
maxOccurs="unbounded">
        <complexType><sequence>
            <element name="extension" type="xdbc:exttype" />
            <element name="mime-type" type="string" />
        </sequence></complexType>
    </element></sequence>
</complexType>

<simpleType name="exttype">
    <restriction base="string">
        <pattern value="[^\\*']*" />
    </restriction>
</simpleType>

</schema>

```

Glossary

Access control entry (ACE)

An entry in the access control list that grants or denies access to a given principal.

access control list (ACL)

A list of access control entries that determines which principals have access to a given resource or resources.

ACE

Access Control Entry. See access control entry.

ACL

Access Control List. See access control list.

API

Application Program Interface. See application program interface.

application program interface (API)

A set of public programmatic interfaces that consist of a language and message format to communicate with an operating system or other programmatic environment, such as databases, Web servers, JVMs, and so forth. These messages typically call functions and methods available for application development.

application server

A server designed to host applications and their environments, permitting server applications to run. A typical example is Oracle9iAS, which is able to host Java, C, C++, and PL/SQL applications in cases where a remote client controls the interface. See also Oracle Application Server.

attribute

A property of an element that consists of a name and a value separated by an equals sign and contained within the start-tags after the element name. In this example, `<Price units='USD'>5</Price>`, `units` is the attribute and `USD` is its value, which must be in single or double quotes. Attributes may reside in the document or DTD. Elements may have many attributes but their retrieval order is not defined.

BC4J

Business Components for Java, a J2EE application development framework that comes with JDeveloper. BC4J is an object-relational mapping tool that implements J2EE Design Patterns.

BFILES

External binary files that exist outside the database tablespaces residing in the operating system. BFILES are referenced from the database semantics, and are also known as External LOBs.

Binary Large Object (BLOB)

A Large Object datatype whose content consists of binary data. Additionally, this data is considered raw as its structure is not recognized by the database.

BLOB

See Binary Large Object.

Business-to-Business (B2B)

A term describing the communication between businesses in the selling of goods and services to each other. The software infrastructure to enable this is referred to as an exchange.

Business-to-Consumer (B2C)

A term describing the communication between businesses and consumers in the selling of goods and services.

callback

A programmatic technique in which one process starts another and then continues. The second process then calls the first as a result of an action, value, or other event. This technique is used in most programs that have a user interface to allow continuous interaction.

cartridge

A stored program in Java or PL/SQL that adds the necessary functionality for the database to understand and manipulate a new datatype. Cartridges interface through the Extensibility Framework within Oracle 8 or later. Oracle Text is such a cartridge, adding support for reading, writing, and searching text documents stored within the database.

Cascading Style Sheets

A simple mechanism for adding style (fonts, colors, spacing, and so on) to Web documents.

CDATA

See character data.

CDF

Channel Definition Format. Provides a way to exchange information about channels on the internet.

CGI

See Common Gateway Interface.

character data (CDATA)

Text in a document that should not be parsed is put within a CDATA section. This allows for the inclusion of characters that would otherwise have special functions, such as &, <, >, and so on. CDATA sections can be used in the content of an element or in attributes.

child element

An element that is wholly contained within another, which is referred to as its parent element. For example `<Parent><Child></Child></Parent>` illustrates a child element nested within its parent element.

Class Generator

A utility that accepts an input file and creates a set of output classes that have corresponding functionality. In the case of 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.

CLASSPATH

The operating system environmental variable that the JVM uses to find the classes it needs to run applications.

client/server

The term used to describe the application architecture where the actual application runs on the client but accesses data or other external processes on a server across a network.

Character Large Object (CLOB)

The LOB datatype whose value is composed of character data corresponding to the database character set. A CLOB may be indexed and searched by the Oracle Text search engine.

CLOB

See Character Large Object.

command line

The interface method in which the user enters in commands at the command interpreter's prompt.

Common Gateway Interface (CGI)

The programming interfaces enabling Web servers to execute other programs and pass their output to HTML pages, graphics, audio, and video sent to browsers.

Common Object Request Broker API (CORBA)

An Object Management Group standard for communicating between distributed objects across a network. These self-contained software modules can be used by applications running on different platforms or operating systems. CORBA objects and their data formats and functions are defined in the Interface Definition Language (IDL), which can be compiled in a variety of languages including Java, C, C++, Smalltalk and COBOL.

Common Oracle Runtime Environment (CORE)

The library of functions written in C that provides developers the ability to create code that can be easily ported to virtually any platform and operating system.

Content

The body of a resource in Oracle XML DB and what you get when you treat the resource like a file and ask for its contents. Content is always an `XMLType`.

CORBA

See Common Object Request Broker API.

CSS

See Cascading Style Sheets.

Database Access Descriptor (DAD)

A DAD is a named set of configuration values used for database access. A DAD specifies information such as the database name or the Oracle Net service name, the `ORACLE_HOME` directory, and Globalization Support configuration information such as language, sort type, and date language.

datagram

A text fragment, which may be in XML format, that is returned to the requester embedded in an HTML page from a SQL query processed by the XSQL Servlet.

DBURITYPE

The Oracle9i datatype used for storing instances of the datatype that permits XPath-based navigation of database schemas.

DOCTYPE

The term used as the tag name designating the DTD or its reference within an XML document. For example, `<!DOCTYPE person SYSTEM "person.dtd">` declares the root element name as `person` and an external DTD as `person.dtd` in the file system. Internal DTDs are declared within the DOCTYPE declaration.

Document Object Model (DOM)

An in-memory tree-based object representation of an XML document that 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 the APIs for programmatic access. DOM views the parsed document as a tree of objects.

Document Type Definition (DTD)

A set of rules that define the allowable structure of an XML document. DTDs are text files that derive their format from SGML and can either be included in an XML document by using the DOCTYPE element or by using an external file through a DOCTYPE reference.

DOM

See Document Object Model.

DOM fidelity

To assure the integrity and accuracy of this data, for example, when regenerating XML documents stored in Oracle XML DB, Oracle XML DB uses a data integrity mechanism, called DOM fidelity. DOM fidelity refers to when the returned XML documents are identical to the original XML document, particularly for purposes of DOM traversals. Oracle XML DB assures DOM fidelity by using a binary attribute, `SYS_XDBPD$`.

DTD

See Document Type Definition.

EDI

Electronic Data Interchange.

element

The basic logical unit of an XML document that can serve as a container for other elements such as children, data, and attributes and their values. Elements are identified by start-tags, such as `<name>`, and end-tags, such as `</name>`, or in the case of empty elements, `<name/>`.

empty element

An element without text content or child elements. It can only contain attributes and their values. Empty elements are of the form `<name/>` or `<name></name>`, where there is no space between the tags.

Enterprise Java Bean (EJB)

An independent program module that runs within a JVM on the server. CORBA provides the infrastructure for EJBs, and a container layer provides security, transaction support, and other common functions on any supported server.

entity

A string of characters that may represent either another string of characters or special characters that are not part of the document's character set. Entities and the text that is substituted for them by the parser are declared in the DTD.

existnode

The SQL operator that returns a TRUE or FALSE based upon the existence of an XPath within an XMLType.

eXtensible Markup Language (XML)

An open standard for describing data developed by the World Wide Web Consortium (W3C) using a subset of the SGML syntax and designed for Internet use.

eXtensible Stylesheet Language Formatting Object (XSLFO)

The W3C standard specification that defines an XML vocabulary for specifying formatting semantics. See FOP.

eXtensible Stylesheet Language Transformation (XSLT)

Also written as XSL-T. The XSL W3C standard specification that defines a transformation language to convert one XML document into another.

eXtensible Stylesheet Language (XSL)

The language used within stylesheets to transform or render XML documents. There are two W3C recommendations covering XSL stylesheets—XSL Transformations (XSLT) and XSL Formatting Objects (XSLFO).

(W3C) eXtensible Stylesheet Language. XSL consists of two W3C recommendations: XSL Transformations for transforming one XML document into another and XSL Formatting Objects for specifying the presentation of an XML document. XSL is a language for expressing stylesheets. It consists of two parts:

- A language for transforming XML documents (XSLT), and
- An XML vocabulary for specifying formatting semantics (XSLFO).

An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary.

extract

The SQL operator that retrieves fragments of XML documents stored as XMLType.

Folder

A directory or node in the Oracle XML DB repository that contains or can contain a resource. A folder is also a resource.

Foldering

A feature in Oracle XML DB that allows content to be stored in a hierarchical structure of resources.

FOP

Print formatter driven by XSL formatting objects. It is a Java application that reads a formatting object tree and then renders the resulting pages to a specified output. Output formats currently supported are PDF, PCL, PS, SVG, XML (area tree representation), Print, AWT, MIF and TXT. The primary output target is PDF.

function-based index

A database index that, when created, permits the results of known queries to be returned much more quickly.

HASPATH

The SQL operator that is part of Oracle Text and used for querying `XMLType` datatypes for the existence of a specific XPath.

hierarchical indexing

The data relating a folder to its children is managed by the Oracle XML DB hierarchical index, which provides a fast mechanism for evaluating path names similar to the directory mechanisms used by operating system filesystems. Any path name-based access will normally use the Oracle XML DB hierarchical index.

HTML

See Hypertext Markup Language.

HTTP

See Hypertext Transport Protocol.

HTTPURITYPE

The datatype used for storing instances of the datatype that permits XPath-based navigation of database schemas in remote databases.

hypertext

The method of creating and publishing text documents in which users can navigate between other documents or graphics by selecting words or phrases designated as hyperlinks.

Hypertext Markup Language (HTML)

The markup language used to create the files sent to Web browsers and that serves as the basis of the World Wide Web. The next version of HTML will be called xHTML and will be an XML application.

Hypertext Transport Protocol (HTTP)

The protocol used for transporting HTML files across the Internet between Web servers and browsers.

iAS

See Oracle9iAS.

IDE

See Integrated Development Environment.

iFS

See Oracle9iFS.

INPATH

The SQL operator that is part of Oracle Text and is used for querying `XMLType` datatypes for searching for specific text within a specific XPath.

instantiate

A term used in object-based languages such as Java and C++ to refer to the creation of an object of a specific class.

Integrated Development Environment (IDE)

A set of programs designed to aide in the development of software run from a single user interface. JDeveloper is an IDE for Java development as it includes an editor, compiler, debugger, syntax checker, help system, and so on, to permit Java software development through a single user interface.

interMedia

The collection of complex datatypes and their access in Oracle. These include text, video, time-series, and spatial data.

Internet Inter-ORB Protocol (IIOP)

The protocol used by CORBA to exchange messages on a TCP/IP network such as the Internet.

J2EE

See Java 2 Platform, Enterprise Edition.

Java

A high-level programming language developed and maintained by Sun Microsystems where applications run in a virtual machine known as a JVM. The JVM is responsible for all interfaces to the operating system. This architecture permits developers to create Java applications and applets that can run on any operating system or platform that has a JVM.

Java 2 Platform, Enterprise Edition (J2EE)

The Java platform (Sun Microsystems) that defines multitier enterprise computing.

Java API for XML Processing (JAXP)

Enables applications to parse and transform XML documents using an API that is independent of a particular XML processor implementation.

JavaBean

An independent program module that runs within a JVM, typically for creating user interfaces on the client. Also known as Java Bean. The server equivalent is called an Enterprise JavaBean (EJB). See also Enterprise JavaBean.

Java Database Connectivity (JDBC)

The programming API that enables Java applications to access a database through the SQL language. JDBC drivers are written in Java for platform independence but are specific to each database.

Java Developer's Kit (JDK)

The collection of Java classes, runtime, compiler, debugger, and usually source code for a version of Java that makes up a Java development environment. JDKs are designated by versions, and Java 2 is used to designate versions from 1.2 onward.

Java Naming and Directory Interface

A programming interface from Sun for connecting Java programs to naming and directory services such as DNS, LDAP and NDS. Oracle XML DB Resource API for Java/JNDI supports JNDI.

Java Runtime Environment (JRE)

The collection of compiled classes that make up the Java virtual machine on a platform. JREs are designated by versions, and Java 2 is used to designate versions from 1.2 onward.

Java Server Page (JSP)

An extension to the servlet functionality that enables a simple programmatic interface to Web pages. JSPs are HTML pages with special tags and embedded Java code that is executed on the Web server or application server providing dynamic functionality to HTML pages. JSPs are actually compiled into servlets when first requested and run in the server's JVM.

Java Virtual Machine (JVM)

The Java interpreter that converts the compiled Java bytecode into the machine language of the platform and runs it. JVMs can run on a client, in a browser, in a middle tier, on an intranet, on an application server such as Oracle9iAS, or in a database server such as Oracle.

JAXP

See Java API for XML Processing.

JDBC

See Java Database Connectivity.

JDeveloper

Oracle's Java IDE that enables application, applet, and servlet development and includes an editor, compiler, debugger, syntax checker, help system, an integrated UML class modeler, and so on. JDeveloper has been enhanced to support XML-based development by including the Oracle XDK for Java, integrated for easy use along with XML support, in its editor.

JDK

See Java Developer's Kit.

JNDI**JServer**

The Java Virtual Machine that runs within the memory space of the Oracle database. In Oracle 8i Release 1 the JVM was Java 1.1 compatible while Release 2 is Java 1.2 compatible.

JVM

See Java virtual machine.

LAN

See local area network.

Large Object (LOB)

The class of SQL data type that is further divided into Internal LOBs and External LOBs. Internal LOBs include BLOBs, CLOBs, and NCLOBs while External LOBs include BFILES. See also BFILES, Binary Large Object, Character Large Object.

lazy type conversions

A mechanism used by Oracle XML DB to only convert the XML data for Java when the Java application first asks for it. This saves typical type conversion bottlenecks with JDBC.

listener

A separate application process that monitors the input process.

LOB

See Large Object.

local area network (LAN)

A computer communication network that serves users within a restricted geographical area. LANs consist of servers, workstations, communications hardware (routers, bridges, network cards, and so on) and a network operating system.

name-level locking

Oracle XML DB provides for name-level locking rather than collection-level locking. When a name is added to a collection, an exclusive write lock is not placed on the collection, only that name within the collection is locked. The name modification is put on a queue, and the collection is locked and modified only at commit time.

namespace

The term to describe a set of related element names or attributes within an XML document. The namespace syntax and its usage is defined by a W3C Recommendation. For example, the `<xsl:apply-templates/ >` element is identified as part of the XSL namespace. Namespaces are declared in the XML document or DTD

before they are used be using the following attribute syntax:

```
xmlns:xsl="http://www.w3.org/TR/WD-xsl" .
```

national Character Large Object (NCLOB)

The LOB datatype whose value is composed of character data corresponding to the database national character set.

NCLOB

See National Character Large Object.

node

In XML, the term used to denote each addressable entity in the DOM tree.

Notation Attribute Declaration

In XML, the declaration of a content type that is not part of those understood by the parser. These types include audio, video, and other multimedia.

N-tier

The designation for a computer communication network architecture that consists of one or more tiers made up of clients and servers. Typically two-tier systems are made up of one client level and one server level. A three-tier system utilizes two server tiers, typically a database server as one and a Web or application server along with a client tier.

OAG

Open Applications Group.

OAI

Oracle Applications Integrator. Runtime with Oracle iStudio development tool that provides a way for CRM applications to integrate with other ERP systems besides Oracle ERP. Specific APIs must be "message-enabled." It uses standard extensibility hooks to generate or parse XML streams exchanged with other application systems. In development.

OASIS

See Organization for the Advancement of Structured Information.

object-relational

The term to describe a relational database system that can also store and manipulate higher-order data types, such as text documents, audio, video files, and user-defined objects.

Object Request Broker (ORB)

Software that manages message communication between requesting programs on clients and between objects on servers. ORBs pass the action request and its parameters to the object and return the results back. Common implementations are JCORB and EJBs. See also CORBA.

Object View

A tailored presentation of the data contained in one or more object tables or other views. The output of an Object View query is treated as a table. Object Views can be used in most places where a table is used.

OC4J

Oracle9iAS Containers for J2EE, a J2EE deployment tool that comes with JDeveloper.

OCT

See Ordered Collection in Tables.

OE

Oracle Exchange.

OIS

See Oracle Integration Server.

Oracle9iAS (iAS)

The Oracle application server that integrates all the core services and features required for building, deploying, and managing high-performance, n-tier, transaction-oriented Web applications within an open standards framework.

Oracle9iFS

The Oracle file system and Java-based development environment that either runs inside the database or on a middle tier and provides a means of creating, storing, and managing multiple types of documents in a single database repository.

ORACLE_HOME

The operating system environmental variable that identifies the location of the Oracle database installation for use by applications.

Ordered Collection in Tables (OCT)

When elements of a VARRAY are stored in a separate table, they are referred to as an Ordered Collection in Tables.

Oracle Text

An Oracle tool that provides full-text indexing of documents and the capability to do SQL queries over documents, along with XPath-like searching.

Oracle XML DB

A high-performance XML storage and retrieval technology provided with Oracle database server. It is based on the W3C XML data model.

ORB

See Object Request Broker.

Organization for the Advancement of Structured Information (OASIS)

An organization of members chartered with promoting public information standards through conferences, seminars, exhibits, and other educational events. XML is a standard that OASIS is actively promoting as it is doing with SGML.

parent element

An element that surrounds another element, which is referred to as its child element. For example, `<Parent><Child></Child></Parent>` illustrates a parent element wrapping its child element.

Parsed Character Data (PCDATA)

The element content consisting of text that should be parsed but is not part of a tag or nonparsed data.

parser

In XML, a software program that accepts as input an XML document and determines whether it is well-formed and, optionally, valid. The Oracle XML Parser supports both SAX and DOM interfaces.

path name

The name of a resource that reflects its location in the repository hierarchy. A path name is composed of a root element (the first /), element separators (/) and various sub-elements (or path elements). A path element may be composed of any character in the database character set except ("\", "/"). These characters have a special meaning for Oracle XML DB. Forward slash is the default name separator in a path name and backward slash may be used to escape characters.

PCDATA

See Parsed Character Data.

PDA

Personal Digital Assistant, such as a Palm Pilot.

PL/SQL

The Oracle procedural database language that extends SQL. It is used to create programs that can be run within the database.

principal

An entity that may be granted access control privileges to an Oracle XML DB resource. Oracle XML DB supports as principals:

- Database users.
- Database roles. A database role can be understood as a group, for example, the DBA role represents the DBA group of all the users granted the DBA role.

Users and roles imported from an LDAP server are also supported as a part of the database's general authentication model.

prolog

The opening part of an XML document containing the XML declaration and any DTD or other declarations needed to process the document.

PUBLIC

The term used to specify the location on the Internet of the reference that follows.

RDF

Resource Definition Framework.

renderer

A software processor that outputs a document in a specified format.

repository

The set of database objects, in any schema, that are mapped to path names. There is one root to the repository ("/") which contains a set of resources, each with a path name.

resource

An object in the repository hierarchy.

resource name

The name of a resource within its parent folder. Resource names must be unique (potentially subject to case-insensitivity) within a folder. Resource names are always in the UTF8 character set (NVARCHAR).

result set

The output of a SQL query consisting of one or more rows of data.

root element

The element that encloses all the other elements in an XML document and is between the optional prolog and epilog. An XML document is only permitted to have one root element.

SAX

See Simple API for XML.

schema

The definition of the structure and data types within a database. It can also be used to refer to an XML document that support the XML Schema W3C recommendation.

Secure Sockets Layer (SSL)

The primary security protocol on the Internet; it utilizes a public key /private key form of encryption between browsers and servers.

Server-Side Include (SSI)

The HTML command used to place data or other content into a Web page before sending it to the requesting browser.

servlet

A Java application that runs in a server, typically a Web or application server, and performs processing on that server. Servlets are the Java equivalent to CGI scripts.

session

The active connection between two tiers.

SGML

See Structured Generalized Markup Language.

Simple API for XML (SAX)

An XML standard interface provided by XML parsers and used by event-based applications.

Simple Object Access Protocol (SOAP)

An XML-based protocol for exchanging information in a decentralized, distributed environment.

SOAP

See Simple Object Access Protocol.

SQL

See Structured Query Language.

SSI

See Server-Side Include.

SSL

See Secure Sockets Layer.

Structured Generalized Markup Language (SGML)

An ISO standard for defining the format of a text document implemented using markup and DTDs.

Structured Query Language (SQL)

The standard language used to access and process data in a relational database.

Stylesheet

In XML, the term used to describe an XML document that consists of XSL processing instructions used by an XSL processor to transform or format an input XML document into an output one.

SYSTEM

Specifies the location on the host operating system of the reference that follows.

SYS_XMLAGG

The term used to specify the location on the host operating system of the reference that follows.

SYS_XMLGEN

The native SQL function that returns as an XML document the results of a passed-in SQL query. This can also be used to instantiate an `XMLType`.

tag

A single piece of XML markup that delimits the start or end of an element. Tags start with < and end with >. In XML, there are start-tags (<name>), end-tags (</name>), and empty tags (<name/>).

TCP/IP

See Transmission Control Protocol/Internet Protocol.

thread

In programming, a single message or process execution path within an operating system that supports concurrent execution (multithreading).

Transmission Control Protocol/Internet Protocol (TCP/IP)

The communications network protocol that consists of the TCP which controls the transport functions and IP which provides the routing mechanism. It is the standard for Internet communications.

Transviewer

The Oracle term used to describe the Oracle XML JavaBeans included in the XDK for Java.

TransXUtility

TransXUtility is a Java API that simplifies the loading of translated seed data and messages into a database.

UDDI

See Universal Description, Discovery and Integration.

UIX

See User Interface XML.

Uniform Resource Identifier (URI)

The address syntax that is used to create URLs and XPaths.

Uniform Resource Locator (URL)

The address that defines the location and route to a file on the Internet. URLs are used by browsers to navigate the World Wide Web and consist of a protocol prefix, port number, domain name, directory and subdirectory names, and the file name. For example `http://technet.oracle.com:80/tech/xml/index.htm` specifies the location and path a browser will travel to find OTN's XML site on the World Wide Web.

Universal Description, Discovery and Integration (UDDI)

This specification provides a platform-independent framework using XML to describe services, discover businesses, and integrate business services on the Internet.

URI

See Uniform Resource Identifier.

URL

See Uniform Resource Locator.

user interface (UI)

The combination of menus, screens, keyboard commands, mouse clicks, and command language that defines how a user interacts with a software application.

User Interface XML (UIX)

A set of technologies that constitute a framework for building web applications.

valid

The term used to refer to an XML document when its structure and element content is consistent with that declared in its referenced or included DTD.

W3C

See World Wide Web Consortium (W3C).

WAN

See wide area network.

WebDAV

See World Wide Web distributed authoring and versioning.

Web Request Broker (WRB)

The cartridge within OAS that processes URLs and sends them to the appropriate cartridge.

Web Services Description Language (WSDL)

A general purpose XML language for describing the interface, protocol bindings, and deployment details of Web services.

well-formed

The term used to refer to an XML document 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.

wide area network (WAN)

A computer communication network that serves users within a wide geographic area, such as a state or country. WANs consist of servers, workstations, communications hardware (routers, bridges, network cards, and so on), and a network operating system.

Working Group (WG)

The committee within the W3C that is made up of industry members that implement the recommendation process in specific Internet technology areas.

World Wide Web Consortium (W3C)

An international industry consortium started in 1994 to develop standards for the World Wide Web. It is located at www.w3c.org.

World Wide Web Distributed Authoring and Versioning (WebDAV)

The Internet Engineering Task Force (IETF) standard for collaborative authoring on the Web. Oracle XML DB Foldering and Security features are WebDAV-compliant.

Wrapper

The term describing a data structure or software that wraps around other data or software, typically to provide a generic or object interface.

WSDL

See Web Services Description Language.

XDBbinary

An XML element defined by the Oracle XML DB schema that contains binary data. `XDBbinary` elements are stored in the repository when completely unstructured binary data is uploaded into Oracle XML DB.

XDK

See XML Developer's Kit.

XLink

The XML Linking language consisting of the rules governing the use of hyperlinks in XML documents. These rules are being developed by the XML Linking Group under the W3C recommendation process. This is one of the three languages XML supports to manage document presentation and hyperlinks (XLink, XPointer, and XPath).

XML

See eXtensible Markup Language.

XML Developer's Kits (XDKs)

The set of libraries, components, and utilities that provide software developers with the standards-based functionality to XML-enable their applications. In the case of the Oracle XDK for Java, the kit contains an XML parser, an XSLT processor, the XML Class Generator, the Transviewer JavaBeans, and the XSQL Servlet.

XML Gateway

A set of services that allows for easy integration with the Oracle e-Business Suite to create and consume XML messages triggered by business events.

XML Query

The W3C's effort to create a standard for the language and syntax to query XML documents.

XML Schema

The W3C's effort to create a standard to express simple data types and complex structures within an XML document. It addresses areas currently lacking in DTDs, including the definition and validation of data types. Oracle XML Schema Processor automatically ensures validity of XML documents and data used in e-business applications, including online exchanges. It adds simple and complex datatypes to XML documents and replaces DTD functionality with an XML Schema definition XML document.

XMLType

An `XMLType` column stores XML data using an underlying CLOB column in the database.

XMLType views

Oracle XML DB provides a way to wrap existing relational and object-relational data in XML format. This is especially useful if, for example, your legacy data is not in XML but you need to migrate it to an XML format.

XPath

The open standard syntax for addressing elements within a document used by XSL and XPointer. XPath is currently a W3C recommendation. It specifies the data model and grammar for navigating an XML document utilized by XSLT, XLink and XML Query.

XPointer

The term and W3C recommendation to describe a reference to an XML document fragment. An XPointer can be used at the end of an XPath-formatted URI. It specifies the identification of individual entities or fragments within an XML document using XPath navigation.

XSL

See eXtensible Stylesheet Language.

XSLFO

See eXtensible Stylesheet Language Formatting Object.

XSLT

See eXtensible Stylesheet Language Transformation.

XSQL

The designation used by the Oracle Servlet providing the ability to produce dynamic XML documents from one or more SQL queries and optionally transform the document in the server using an XSL stylesheet.

Index

A

- access control entries (ACEs), 18-9
 - elements in, 18-6
- access control lists (ACLs), 18-2
 - bootstrap, 18-4
 - concurrency resolution, 18-5
 - default, 18-4
 - defined, 13-5
 - features, 18-5
 - managing, 18-11
 - managing from Enterprise Manager, 21-22
 - restrictions, 18-12
 - security, row-level, 18-13
 - setting the resource property, 18-11
 - summary, 1-12
 - term explained, 1-30
 - updating, 18-10
 - using, 18-9
- accessing
 - Java examples for, 9-4
 - using XDBUriType, 3-44
 - with JDBC, 9-4
 - XML documents using Java, 9-3
- adding
 - XMLType columns, 4-8
- Advanced Queuing (AQ)
 - definition, 23-2
 - enqueueing, 23-9
 - hub-and-spoke architecture support, 23-4
 - IDAP, 23-5
 - message management support, 23-4
 - messaging scenarios, 23-2
 - point-to-point support, 23-2
 - publish/subscribe support, 23-2
 - XML servlet, 23-11
 - XMLType queue payloads, 23-6
- aggregating
 - XSQL and XMLAgg, 10-51
- ALTER INDEX, using sections, 7-10
- any, 13-6
- attributes
 - collection, 5-38
 - columnProps, 5-71
 - Container, 13-7
 - defaultTable, 5-71
 - for root element, 3-21
 - in elements, 5-26
 - maintainDOM, 5-22
 - maintainOrder, 5-38
 - mapping any, 5-46
 - maxOccurs, 5-38
 - namespaces, 5-5
 - noNameSpaceSchemaLocation, 3-21
 - of XMLFormat, 10-44
 - passing to SYS_DBURIGEN, 12-30
 - REF, 5-39, 5-48
 - schemaLocation, 3-21, 5-9
 - setting to NULL, 4-35
 - SQLInLine, 5-38, 5-39
 - SQLName, 5-25
 - SQLSchema, 5-25
 - SQLType, 5-25, 5-29, 5-40
 - storeVarrayAsTable, 5-72
 - SYS_XDBPDS\$, 5-58
 - tableProps, 5-71
 - xdb.defaultTable, 3-40
 - xdb.SQLType, 3-30

- XMLAttributes in XMLElement, 10-7
- XMLDATA, 4-13, 5-51
- XMLType, in AQ, 23-6
- xsi.NamespaceSchemaLocation, 5-5
- xsi.noNamespaceSchemaLocation, 11-10

authenticatedUser

- DBuri security, 12-37

AUTO_SECTION_GROUP

- using, 7-10

B

- B*Tree, 1-12, 4-6, 5-52
 - indexing, 3-26
- bootstrap ACL, 18-4

C

- CASCADE mode, 5-13
- cascading style sheets, see CSS, D-7
- catalog views, F-18
- cfg_get, 16-9, A-9
- cfg_refresh, A-10
- CharacterData, 8-21
- Classes
 - XDBCContextFactory
 - initial contexts for JNDI, 17-6
- CLOB storage of XMLType, 4-5
- collection attribute, 5-38
- collection index, 5-61
- collections, 3-32, 18-6
- columnProps attribute, 5-71
- complexType
 - collections, 3-32
 - cycling, 5-49
 - cycling between, 5-47
 - elements, B-3
 - handling inheritance, 5-42
 - in XML schema, explained, B-35
 - mapping any and any attributes, 5-46
 - mapping to SQL, 5-38
 - restrictions, 5-42
- concatenating
 - elements using XMLConcat, 10-16
- configuring

- API, A-9
 - protocol server in Oracle XML DB, 19-4
 - servlet, example, 20-12
 - servlets in Oracle XML DB, 20-4
 - using Enterprise Manager, 21-7
- constraints
 - on XMLType columns, 5-52
 - structured storage, 3-33
 - using with XMLType tables, 3-22
- CONTAINS, 4-38, 7-6
 - compared against existsNode, 7-38
 - compared to existsNode(),extract(), 4-41
- contents, element, 13-6
- CREATE TABLE
 - XMLType storage, 5-51
- createFolder(), 16-3
- createXML
 - inserting with CLOB, example, 4-15
 - inserting with string, 4-16
 - summarized, 3-17
- creating
 - XML schema-based tables, columns, 5-23
 - XMLType columns, 4-8
 - XMLType table, 4-7
- CSS and XSL, D-7
- CTX_DDL.Add_Field_Section, 7-26
- CTXAPP
 - role, 7-6
- CTXSYS.PATH_SECTION_GROUP, 7-36
- CTXXPATH, 4-41
 - indexes, 7-45
 - storage preferences, 7-47
- cycling in complexTypes
 - self-referencing, 5-49

D

- data integrity
 - in structured, unstructured storage, 3-26
 - Oracle XML DB, 3-33
- date
 - format conversion in updateXML(), 5-71
 - format conversions for XML, 5-63
 - mapping to SQL, 5-36
- DBMS_METADATA, 12-5

- DBMS_XDB, F-19
 - AclCheckPrivileges, database objects, 18-13
 - cfg_get, A-9
 - cfg_refresh, A-10
 - changePrivilege, 18-12
 - checkPrivileges, 18-13
 - configuration management, 16-8
 - getAclDocument, 18-12
 - Link, 16-2
 - LockResource, 16-2
 - overview, 16-2
 - rebuilding hierarchical index, 16-11
 - security, 16-5
- DBMS_XDB_VERSION, 14-2, F-24
 - subprograms, 14-9
- DBMS_XDBT, F-25
- DBMS_XMLDOM, 8-5, 8-12, F-6
- DBMS_XMLGEN, 10-20, F-22
 - generating complex XML, 10-29
 - generating XML, 10-2
- DBMS_XMLPARSER, 8-24, F-13
- DBMS_XMLSCHEMA, 5-7, F-15
 - deleteSchema, 5-7
 - generateSchema() function, 5-18
 - generateSchemas() function, 5-18
 - mapping of types, 5-32
 - registerSchema, 5-7
- DBMS_XSLPROCESSOR, 8-28, F-14
- dbmsxsch.sql, F-15
- DBUri, 12-10
 - and object references, 12-17
 - identifying a row, 12-15
 - identifying a target column, 12-16
 - retrieving column text value, 12-16
 - retrieving the whole table, 12-14
 - security, 12-37
 - Servlet, 12-34
 - Servlet, installation, 12-36
 - syntax guidelines, 12-13
 - URL specification, 12-12
 - XPath expressions in, 12-13
- DBUri-refs, 12-9
 - HTTP access, 12-34
 - where it can be used, 12-17
- DBUriType
 - defined, 12-2
 - examples, 12-25
 - notation for fragments, 12-12
 - stores references to data, 12-6
- default table
 - creating, 5-71
 - defining a, 3-40
- defaultTable attribute, 5-71
- deleteSchema, 9-21
- deleting
 - resources, 3-39, 15-10
 - rows using extract(), 4-37
 - rows with XMLType columns, 4-37
 - using extract(), 4-37
 - XML schema using DBMS_XMLSCHEMA, 5-13
- DEPTH, 15-8
- dequeuing
 - with AQ XML Servlet, 23-11
- document
 - fidelity, explained, 1-4
 - no order, 5-65
 - no order with extract(), 5-69
 - order, 5-63
 - order with extract(), 5-68
 - order, query rewrites with collection, 5-61
 - ordering preserved in mapping, 5-69
- DOM
 - differences, and SAX, 8-6
 - explained, 1-28
 - fidelity, 5-21
 - fidelity, default, 5-57
 - fidelity, in structured storage, 4-5
 - fidelity, structured or unstructured storage, 3-27
 - fidelity, summarized, 1-4
 - fidelity, SYS_XDBPDS, 5-22
 - introduced, 8-5
 - Java API, 9-2
 - Java API features, 9-16
 - Java API, XMLType classes, 9-18
 - NodeList, 8-21
 - non-supported, 8-5
- DOM API for PL/SQL, 8-5
- DOMDocument, 9-19
- dropping

- XMLType columns, 4-8
- DTD
 - limitations, B-33

E

- elementFormDefault, 5-62
- elements
 - access control entries (ACEs), 18-6
 - any, 13-6
 - complexType, B-3
 - Contents, Resource index, 13-6
 - simpleType, B-3
 - XDBBinary, 13-11
 - XML, 8-4
- enqueueing
 - adding new recipients after, 23-14
 - using AQ XML servlet, 23-9
- EQUALS_PATH
 - summary, 15-5
 - syntax, 15-8
- existsNode
 - and CONTAINS, querying, 7-38
 - dequeing messages, 2-10
 - finding XML elements, nodes, 4-20
 - indexing with CTXXPATH, 7-45
 - query rewrite, 5-52
 - XPath rewrites, 5-63
- extract, 5-68
 - deleting, 4-37
 - dequeing messages, 2-10
 - mapping, 5-69
 - query rewrite, 5-52
 - querying XMLType, 4-26
 - rewrite in XPath expressions, 5-68
- extracting
 - data from XML, 4-27
- extractValue, 4-23
 - creating indexes, query rewrite, 5-67
 - query rewrite, 5-52
 - rewrites in XPath expressions, 5-66

F

- factory method, 12-25

- folder, defined, 13-4
- foldering
 - explained, 13-2
 - summary, 1-12
- FORCE mode option, 5-13
- Frequently Asked Questions (FAQs)
 - AQ and XML, 23-13
 - Oracle Text, 7-64
 - versioning, 14-12
- FTP
 - configuration parameters, Oracle XML DB, 19-5
 - creating default tables, 5-71
 - protocol server, features, 19-7
 - protocol server, using, 3-44
- function-based index
 - creating in Enterprise Manager, 21-42
 - creating on extract or existsNode, 4-38
- functions
 - createXML, 4-15
 - DBUriType, 12-18
 - isSchemaValid, 6-10
 - isSchemaValidated, 6-9
 - schemaValidate, 6-9
 - setSchemaValidated, 6-10
 - SYS_DBURIGEN, 12-29
 - SYS_XMLAgg, 10-50
 - SYS_XMLGEN, 10-41
 - transform, 6-2
 - updateXML, 5-70
 - XMLAgg, 10-17
 - XMLColAttVal, 10-19
 - XMLConcat, 10-15
 - XMLElement, 10-5
 - XMLForest, 10-9
 - XMLSequence, 10-11, 10-13
 - XMLTransform, 6-2
 - XMLType, 4-7

G

- genbean flag, 9-21
- generating
 - DBUriType using SYS_DBURIGEN, 12-29
- generating XML
 - DBMS_XMLGEN example, 10-29

- element forest
 - XMLColAttVal, 10-19
- from SQL, DBMS_XMLGEN, 10-20
- one document from another, 10-12
- SQL, SYS_XMLGEN, 10-41
- SYS_XMLAgg, 10-50
- using SQL functions, 10-2
- XML SQL Utility (XSU), 10-54
- XMLAgg, 10-17
- XMLConcat, 10-15
- XMLElement, 10-5
- XMLForest, 10-9
- XMLSequence, 10-11
- XSQL, 10-51
- getClobVal
 - summarized, 3-17
- getNameSpace
 - summarized, 3-17
- getRootElement
 - summarized, 3-17
- getXMLType, 9-18
- global XML schema, 5-11
- groups, 18-6

H

- HASPATH, 4-39, 7-10
 - operator, 7-12
 - path existence searching, 7-20
 - path searching, 7-19
- HTTP
 - access for DBUri-refs, 12-34
 - accessing Java servlet or XMLType, 20-3
 - accessing Repository resources, 13-11
 - configuration parameters, WebDAV, 19-5
 - creating default tables, 5-71
 - HttpUriType, DBUriType, 12-23
 - improved performance, 19-2
 - Oracle XML DB servlets, 20-8
 - protocol server, features, 19-8
 - requests, 20-8
 - servlets, 20-4
 - UriFactory, 12-38
 - using UriRefs to store pointers, 12-7
- HttpUriType

- accesses remote pages, 12-6
- defined, 12-2
- hub-and-spoke architecture, enabled by AQ, 23-4

I

- IDAP
 - architecture, 23-6
 - transmitted over Internet, 23-5
 - XML schema, 23-12
- index
 - collection, 5-61
- Index Organized Table (IOT), 5-72
- indexing
 - B*Tree, 3-26
 - function-based on existsNode(), 4-38
 - in structured, unstructured storage, 3-26
 - Oracle Text, CTXXPATH, 7-45
 - Oracle Text, XMLType, 7-34
 - XMLType, 4-38
- Information Set
 - W3C introducing XML, C-26
- inheritance
 - in XML schema, restrictions in
 - complexType, 5-44
- INPATH, 4-39, 7-10
 - operator, 7-12
- INPATH operator, 7-12
- inserting
 - into XMLType, 4-9
 - new instances, 5-52
 - XML data into XMLType columns, 4-15
- installing
 - from scratch, Oracle XML DB, A-2
 - manually without DBCA, A-3
- instance document
 - specifying root element namespace, 5-5
 - XML, described, B-36
- Internet Data Access Presentation (IDAP)
 - SOAP specification for AQ, 23-5
- isFragment
 - summarized, 3-17
- isSchemaValid, 6-10
- isSchemaValidated, 6-9

J

- Java
 - Oracle XML DB guidelines, 20-3
 - using JDBC to access XMLType objects, 20-3
 - writing Oracle XML DB applications, 20-2
- Java bean
 - deleteSchema, 9-21
 - genbean flag, 9-21
 - generated example, 9-24
 - generated names, 9-21
 - generated with XML schema registration, 5-17
 - only for XML schema-based, 5-18
- Java Bean API for XMLType, 9-20, E-6
 - using, 9-21
- Java DOM API for XMLType, E-2
- Java Stored Procedure, 9-26
- JDBC
 - accessing documents, 9-4
 - manipulating data, 9-6
 - using SQL to determine object properties, 17-13
- JNDI, 9-3, E-7
 - bind(), 9-17
 - context inputs, 17-8
 - context outputs, 17-10
 - determining properties, 17-11
 - oracle.xdb.spi, 17-3
 - support, 17-2
 - using Resource API, 17-4

L

- Lazy Manifestation (LM), 3-25
- LDAP
 - Oracle XML DB, in, 18-5
- Least Recently Used (LRU), 3-25
- Link, 16-2
- LOBs
 - mapping XML fragments to, 5-40
- location path, C-5
- LockResource, 16-2

M

- maintainDOM, 5-65
- maintainOrder attribute, 5-38

- mapping
 - collection predicates, 5-60
 - complexType any, 5-46
 - complexTypes to SQL, 5-38
 - overriding using SQLType attribute, 5-33
 - predicates, 5-59
 - scalar nodes, 5-59
 - simpleContent to object types, 5-45
 - simpleType XML string to VARCHAR2, 5-37
 - simpleTypes, 5-34
 - SQL to Java, 9-23
 - type, setting element, 5-32
 - XML to Java bean, XML schema entities, 9-23
- maxOccurs, 5-38
- MIME
 - overriding with DBUri servlet, 12-35
- modes
 - CASCADE, 5-13
 - FORCE, 5-13
- MULTISET operator
 - using with SYS_XMLGEN selects, 10-46

N

- NamedNodeMap object, 8-21
- namespace
 - defining, 3-28
 - handling in query rewrites, 5-62
 - handling in XPath, 5-62
 - W3C introducing, C-18
 - XDBResource, 13-17
 - XML schema URL, 5-5
 - xmlns, D-4
 - XMLSchema-Instance, 3-21
- naming SQL objects
 - SQLName, SQLType attributes, 5-25
- navigational access, 13-9
- nested
 - generating nested XML using DBMS_XMLGEN, 10-31
 - object tables, 3-32
 - sections in Oracle Text, 7-54
 - XML, generating with XMLElement, 10-7
 - XMLAgg functions and XSQL, 10-51
- newDOMDocument, 8-20

NodeList object, 8-21

NULL

mapping to in XPath, 5-61

O

object references and DBUri, 12-17

operators

CONTAINS, 4-38, 7-6

CONTAINS compared, 4-41

DEPTH, 15-8

EQUALS_PATH, 15-8

HASPATH, 7-12

INPATH, 7-12

MULTISET and SYS_XMLGEN, 10-46

PATH, 15-8

UNDER_PATH, 15-6

WITHIN, 7-7, 7-11

XMLIsValid, 6-9

Oracle Enterprise Manager

configuring Oracle XML DB, 21-7

console, 21-7

creating a view based on XML schema, 21-39

creating function-based index, 21-42

creating resources, 21-12

features, 21-3

granting privileges, XML Tab, 21-23

managing security, 21-22

managing XML schema, 21-27

Oracle Net Services, 1-12

Oracle Text

advanced techniques, 7-45, 7-49

ALTER INDEX, 7-10

attribute sections, constraints, 7-52

building query applications, 7-21

comparing CONTAINS and existsNode, 7-38

conference Proceedings example, 7-56

CONTAINS and XMLType, 4-38

CONTAINS operator, 7-6

creating index on XMLType columns, 4-11

creating on XMLType columns, 4-39

CTXSYS, 7-5

CTXXPATH, 7-45

DBMS_XDBT, F-25

installing, 7-4

Oracle XML DB, and, 7-37

querying, 7-6

querying within attribute sections, 7-30

searching data with, 7-3

searching XML in CLOBs, 1-24

section_group, deciding which to use, 7-23

users and roles, 7-5

XMLType, 7-4

XMLType indexing, 7-34

Oracle XML DB, 3-5

access models, 2-7

advanced queueing, 1-24

application language, 2-8

architecture, 1-8

benefits, 1-3

configuring with Enterprise Manager, 21-7

designing, 2-3

features, 1-4

foldering, 13-2

installation, A-2

installing, 2-2

installing manually, A-3

introducing, 1-2

Java applications, 20-2

processing models, 2-9

Repository, 1-6, 3-34, 13-4

storage models, 2-10

storing XMLType, 4-4

upgrading, A-4

using XSL/XSLT with, 3-16

versioning, 14-2

when to use, 2-2

Oracle XML DB Resource API for Java/JNDI

calling sequence, 17-5

examples, 17-11

using, 17-4

oracle.xdb, E-2

oracle.xdb.bean, 9-21, E-6

oracle.xdb.dom, E-2

oracle.xdb.spi, 17-9, E-7

JNDI and WebDAV, 17-3

XDBResource.getContent(), E-10

XDBResource.getContentType, E-10

XDBResource.getCreateDate, E-10

XDBResource.getDisplayName, E-10

- XDBResource.getLanguage(), E-10
- XDBResource.getLastModDate, E-10
- XDBResource.getOwnerId, E-10
- XDBResource.setACL, E-10
- XDBResource.setAuthor, E-10
- XDBResource.setComment, E-10
- XDBResource.setContent, E-10
- XDBResource.setContentType, E-11
- XDBResource.setCreateDate, E-11
- XDBResource.setDisplayName, E-11
- XDBResource.setInheritedACL, E-11
- XDBResource.setLanguage, E-11
- XDBResource.setLastModDate, E-11
- XDBResource.setOwnerId, E-11
- oracle.xdb.XDBResource
 - JNDI, 17-7
- oracle.xdb.XMLType, 9-18
 - JNDI, 17-7
- ora.contains
 - creating a policy for, 7-42
 - XPath full-text searches, 7-40
- ordered collections in tables (OCTs), 5-72
 - default storage of VARRAY, 5-38
 - rewriting collection index, 5-61

P

- PATH, 15-8
- PATH_SECTION_GROUP
 - using, 7-10
- PATH_VIEW, 3-38
 - structure, 15-3
- Path-based access
 - explained, 13-9
- pathname
 - resolution, 13-7
- performance
 - improved using Java writeToStream, 13-12
 - improvement for structured storage, 3-25
- piecewise update, 1-5
- PL/SQL DOM
 - examples, 8-22
 - methods, 8-12
- PL/SQL DOM API for XMLType, F-6
- PL/SQL Parser for XMLType, F-13

- PL/SQL XSLT Processor for XMLType, F-14
- point-to-point
 - support in AQ, 23-2
- Positional Descriptor (PD), 5-22
- predicates, 5-59
 - collection, mapping of, 5-60
- principals, 18-6
- privileges
 - aggregate, 18-8
 - atomic, 18-7
 - granting from Oracle Enterprise Manager, 21-23
- processXSL, 8-31
- protocol server
 - architecture, 19-3
 - configuration parameters, 19-4
 - FTP, 19-7
 - FTP configuration parameters, 19-5
 - HTTP, 19-8
 - HTTP/WebDAV configuration
 - parameters, 19-5
 - Oracle XML DB, 19-2
 - using, 3-44
 - WebDAV, 19-10
- protocols
 - access calling sequence, 13-11
 - access to Repository resources, 13-10
 - retrieving resource data, 13-11
 - storing resource data, 13-11
- publish/subscribe
 - support in AQ, 23-2
- purchase order
 - XML, 3-5
- purchaseorder.xml, D-7

Q

- query access
 - using RESOURCE_VIEW and PATH_VIEW, 15-2
- query results
 - presenting, 7-21, 7-34
- query rewrite, 5-52
- query-based access
 - using SQL, 13-12
- querying

- resources, 3-38
- XML data, 4-17
- XMLType, 4-18
- XMLType.transient data, 4-26

R

- rebuilding hierarchical indexes, 16-11
- REF attribute, 5-39, 5-48
- registerHandler, 12-26
- Reinstalling
 - Oracle XML DB
 - reinstalling, A-4
- Repository, 3-34, 13-4
 - based on WebDAV, 3-35
 - hierarchy explained, 3-35
 - query-based access, 3-36
 - where is the data stored, 13-6
- resource id
 - new version, 14-5
- RESOURCE_VIEW
 - explained, 15-2
 - PATH_VIEW differences, 15-4
 - querying resource documents, 3-38
 - structure, 15-3
- resources
 - access using UNDER_PATH, 15-9
 - accessing Repository, 13-8
 - accessing with protocols, 19-6
 - changing privileges, 18-12
 - configuration management, 16-8
 - controlling access to, 18-7
 - creating from Enterprise Manager, 21-12
 - defined, 13-4
 - deleting, 13-7
 - deleting non-empty containers, 15-11
 - deleting using DELETE, 15-10
 - extending metadata properties, 13-17
 - inserting data using RESOURCE_VIEW, 15-9
 - management using DBMS_XDB, 16-2
 - managing, 18-11
 - multiple simultaneous operations, 15-12
 - required privileges for operations, 18-8
 - retrieving access control lists (ACLs), 18-12
 - setting property in access control lists

- (ACLs), 18-11
- storage options, 3-40
- updating, 15-11

ResourceType

- extending, 13-18

S

- scalar nodes, mapping, 5-59
- scalar value
 - converting to XML document using SYS_XMLGEN, 10-45
- schemaLocation, 5-9
- schemaValidate, 3-24, 6-9
 - method, 5-15
- searching CLOBs, 1-24
- security
 - access control entries (ACEs), 18-6
 - aggregate privileges, 18-8
 - DBUri, 12-37
 - management using DBMS_XDB, 16-5
 - management with Enterprise Manager, 21-22
 - user and group access, 18-6
- servlets
 - accessing Repository data, 13-13
 - and session pooling, 20-9
 - APIs, 20-10
 - AQ XML, 23-11
 - configuring, 20-12
 - configuring Oracle XML DB, 20-4
 - DBUri, URL into a query, 12-34
 - example, 20-10
 - explained, 1-30
 - installing, 20-11
 - session pooling, 20-9
 - session pooling and Oracle XML DB, 19-2
 - testing the, 20-12
 - writing Oracle XML DB HTTP, 20-4
 - XML manipulation, with, 20-3
- session pooling, 20-9
 - protocol server, 19-2
- setSchemaValidated, 6-10
- Simple Object Access Protocol (SOAP) and IDAP, 23-5
- simpleContent

- mapping to object types, 5-45
- simpleType
 - elements, B-3
 - mapping to SQL, 5-34
 - mapping to VARCHAR2, 5-37
- SOAP
 - access through Advanced Queuing, 1-12
 - and IDAP, 23-5
- space
 - requirements in structured, unstructured storage, 3-26
- SQL functions
 - existsNode, 4-20
 - extractValue, 4-23
- SQL*Loader, 22-2
- SQLInLine attribute, 5-38
- SQLName, 5-25
- SQLType, 5-25, 5-29
 - attribute, 5-40
- SQLX
 - generating XML, 10-5
 - generating XML, for, 10-2
 - operators, explained, 1-5
 - Oracle extensions, 10-2
- storage
 - of complex types, 3-32
 - options for resources, 3-40
 - structured, 3-33
 - XMLDATA, 4-13
 - structured or unstructured, 3-24
 - structured, query rewrite, 5-52
 - VARRAYs and OCT, 5-72
 - XML-schema-based, 5-21
 - XMLType CREATE TABLE, 5-51
- storeVarrayAsTable attribute, 5-72
- substitution
 - creating columns by inserting HttpUriType, 12-24
- /sys, restrictions, 13-3
- SYS_DBURIGEN, 12-29
 - examples, 12-31
 - inserting database references, 12-31
 - passing columns or attributes, 12-30
 - retrieving object URLs, 12-33
 - returning partial results, 12-32

- returning Uri-refs, 12-33
- text function, 12-31

- SYS_REFCURSOR
 - generating a document for each row, 10-13
- SYS_XDBPDS\$, 5-22, 5-65
 - in query rewrites, 5-58
 - suppressing, 5-22
- SYS_XMLAgg, 10-50
- SYS_XMLGEN, 10-41
 - converting a UDT to XML, 10-46
 - converting XMLType instances, 10-47
 - generating XML in SQL queries, 4-12
 - inserting, 4-16
 - object views, 10-48
 - static member function create, 10-44
 - using with object views, 10-48
 - XMLFormat attributes, 10-44
 - XMLGenFormatType object, 10-43

T

- tableProps attribute, 5-71
- tablespace
 - do not drop, A-2
- transform, 6-2
- triggers
 - BEFORE INSERT trigger, 3-24
 - BEFORE INSERT, using with XMLType, 3-23
 - creating XMLType, 4-37
 - using with XMLType, 4-37

U

- UDT
 - generating an element from, 10-9
 - generating an element using XMLForest, 10-10
- UNDER_PATH, 3-38, 15-6
 - summary, 15-5
- UPDATE_VAL
 - binding XMLType objects to SQL, 17-2
- updateXML, 5-70
 - creating views, 4-35
 - mapping NULL values, 4-35
 - replacing contents of node tree, 3-13
 - updating text node value, 3-12

- updating
 - resources, 3-38, 15-11
 - same node more than once, 4-36
- upgrading
 - existing installation, A-4
- URI
 - base, C-28
- UriFactory, 12-25
 - configuring to handle DBUri-ref, 12-38
 - factory method, 12-25
 - generating UriType instances, 12-25
 - registering ecom protocol, 12-27
 - registering new UriType subtypes, 12-26
 - registerURLHandler, 12-26
- Uri-ref, see also Uri-reference, 12-4
- Uri-reference
 - database and session, 12-17
 - datatypes, 12-6
 - DBUri, 12-10
 - DBUri and object references, 12-17
 - DBUri syntax guidelines, 12-13
 - DBUri-ref, 12-9
 - DBUri-ref uses, 12-17
 - DBUriType examples, 12-25
 - explained, 12-4
 - HTTP access for DBUri-ref, 12-34
 - UriFactory package, 12-25
 - UriType examples, 12-23
 - UriTypes, 12-22
- UriTypes, 12-22
 - benefits, 12-6
 - creating Oracle Text index on column, 7-37
 - examples, 12-23
 - subtypes, advantages, 12-28
- URL
 - identifying XML schema, for, 5-5

V

- validating
 - examples, 6-10
 - isSchemaValid, 6-10
 - isSchemaValidated, 6-9
 - schemaValidate, 6-9
 - SetSchemaValidated, 6-10

- with XML schema, 5-7
- VARRAYS
 - inline, 3-32
 - storage using OCT, 5-72
- VCR, 14-4, 14-6
 - access control and security, 14-8
- version-controlled resource (VCR), 14-4, 14-6
- versioning, 1-11, 14-2
 - FAQs, 14-12
- views
 - RESOURCE and PATH, 15-2

W

- W3C DOM Recommendation, 8-9
- WebDAV
 - oracle.xdb.spi, 17-3
 - protocol server, features, 19-10
 - standard introduced, 3-35
- WebFolder
 - creating in Windows 2000, 19-11
- WITHIN
 - in Oracle Text, 7-7
 - limitations, 7-11
 - syntax, 7-11
- writeToStream, 13-12

X

- XDB\$RESOURCE table, 13-18
- XDBBinary, 13-5, 13-11
 - explained, 1-29
- xdbconfig.xml, 19-2
- XDBResource
 - JNDI Context output, 17-10
 - namespace, 13-17
 - xsd, 13-17
- XDBSchema.xsd, 5-20
- XDBUri, 12-5
- XDBUriType
 - accessing non-schema content, 3-44
 - accessing Repository content, 3-44
 - defined, 12-2
 - stores references to Repository, 12-6
- XDK for PL/SQL, backward compatibility, 8-2

XML

- binary datatypes, 5-35
- fragments, mapping to LOBs, 5-40
- primitive datatypes, 5-37
- primitive numeric types, 5-35
- XML DB Resource API for Java/JNDI, 17-2
- XML DB, Oracle, 3-5
- XML schema
 - and Oracle XML DB, 1-7, 5-5
 - benefits, 5-6
 - compared to DTD, B-31
 - complexType declarations, 5-42
 - creating default tables during registration, 5-71
 - cyclical references, 5-75
 - cyclical references between, 5-72
 - DTD limitations, B-33
 - elementFormDefault, 5-62
 - Enterprise Manager, managing them
 - from, 21-27
 - features, B-34
 - generating Java beans, 5-17
 - global, 5-11
 - inheritance in, complexType restrictions, 5-44
 - key feature explained, 1-5
 - local, 5-10
 - local and global, 5-10
 - managing and storing, 5-20
 - mapping to SQL object types, 8-10
 - navigating in Enterprise Manager, 21-28
 - registering, 5-8
 - registering using DBMS_XMLSCHEMA, 5-8
 - root, 5-20
 - SQL and Java datatypes, 9-23
 - SQL mapping, 5-29
 - storage and access, 5-12
 - storage of XMLType, 3-28
 - unsupported constructs in query rewrites, 5-55
 - updateXML(), 5-70
 - URLs, 5-16
 - validating an XML document, 3-22
 - W3C Recommendation, 3-17, 5-3
 - xdb.SQLType, 3-30
 - XMLType methods, 5-20
- XML schema, creating a view in Enterprise Manager, 21-39
- XML Schema, introducing W3C, B-2
- XML SQL Utility
 - generating XML, 10-54
- XML SQL Utility (XSU)
 - generating XML, 10-3
- XML storage, 3-24
- XML_SECTION_GROUP
 - using, 7-8
- XMLAgg, 10-17
- XMLAttributes, 10-7
- XMLColAttVal, 10-19
- XMLConcat, 10-15
 - concatenating XML elements in
 - argument, 10-16
 - returning XML elements by
 - concatenating, 10-16
- XMLDATA
 - column, 5-59
 - optimizing updates, 5-70
 - parameter, F-3
 - pseudo-attribute of XMLType, 5-51
 - structured storage, 4-13
- XMLElement, 10-5
 - attribute, 10-7
 - generating elements from DTD, 10-9
 - using namespaces to create XML
 - document, 10-8
- XMLForest, 10-9
 - generating elements, 10-10
 - generating elements from DTD, 10-10
- XMLFormat
 - XMLAgg, 10-17
- XMLFormat object type
 - SYS_XMLGEN
 - XMLFormatType object, 10-43
- XMLGenFormatType object, 10-43
- XMLIsValid
 - validating
 - XMLIsValid, 6-9
- XMLSequence, 10-11
 - extracting description nodes, 3-11
 - generating an XML document for each
 - row, 10-13
 - generating one document from another, 10-12
 - unnesting collections in XML to SQL, 10-14

- XMLTransform, 4-36, 6-2
- XMLType, 4-2
 - adding columns, 4-8
 - API, F-2
 - benefits, 4-3
 - CLOB storage, 4-5
 - column, 3-3
 - constraints, specifying, 4-14
 - CONTAINS operator, 4-38
 - CREATE TABLE, 5-51
 - creating columns, 4-8
 - creating columns, example, 4-8
 - creating Oracle Text index, 4-11
 - deleting a row containing, 4-10
 - deleting rows, 4-37
 - deleting using extract(), 4-37
 - dropping columns, 4-8
 - extracting data, 4-27
 - functions, 4-7
 - guidelines for using, 4-11
 - how to use, 4-7
 - indexing, 7-34
 - indexing columns, 4-38
 - inserting into, 4-9
 - inserting with createXML() using string, 4-16
 - inserting with SYS_XMLGEN(), 4-16
 - inserting XML data, 4-15
 - instances, PL/SQL APIs, 8-2
 - Java
 - writeToStream, 13-12
 - loading data, 22-2
 - manipulating data in columns, 4-14
 - Oracle Text support, 7-4
 - querying, 4-17, 4-18
 - querying transient data, 4-26
 - querying with extract() and existsNode(), 4-26
 - querying XMLType columns, 4-26
 - queue payloads, 23-6
 - storage architecture, 1-12
 - storage characteristics, 4-12
 - storing data in Oracle XML DB, 4-4
 - summarized, 1-4
 - table, 3-3
 - table storage, 1-10
 - table, querying with JDBC, 9-4
 - tables, storing, 5-24
 - tables, views, columns, 5-14
 - triggers, 4-37
 - updating column, example, 4-9
 - using in SQL SELECT statement, 4-9
 - views, access with PL/SQL DOM APIs, 8-11
 - when to use, 4-4
 - Xpath support, 4-38
- XMLType, loading with SQL*Loader, 22-2
- XPath
 - basics, D-6
 - explained, 1-28
 - expressions, mapping, 5-58
 - mapping for extract(), 5-68
 - mapping for extract() without document order, 5-69
 - mapping for extractValue(), 5-66
 - mapping to NULL in, 5-61
 - mapping, simple, 5-58
 - rewrites for existNode(), 5-63
 - rewriting expressions, 5-54
 - support, 4-38
 - text(), 5-59
 - unsupported constructs in query rewrites, 5-55
 - use for searching data, 1-5
 - using with Oracle XML DB, 3-5
 - W3C introducing, C-2
- XPath expressions
 - supported, 5-55
- xsi.noNamespaceSchemaLocation, 5-5
- XML
 - and CSS, D-7
 - basics, D-2
 - defined, 1-28
- XSL stylesheet, example, D-7
- XSLT, 8-12
 - 1.1 specification, D-5
 - explained, D-5
- xml
 - include-xml
 - aggregating results into one XML, 10-51
 - generating XML from database, 10-52
- XSQL Pages Publishing Framework
 - generating XML, 10-3, 10-51

