Oracle® XML DB

Developer's Guide

10*g* Release 1 (10.1)

Part No. B10790-01

December 2003

This manual describes Oracle XML DB components and related tools, such as SQL*Loader and Oracle Enterprise Manager, and how to use them with Oracle XML DB. It includes guidelines and examples for managing, loading, storing, processing, accessing, generating, and searching XML data stored in Oracle Database.

ORACLE®

Oracle XML DB Developer's Guide, 10*g* Release 1 (10.1)

Part No. B10790-01

Copyright © 2002, 2003, Oracle. All rights reserved.

Primary Author: Shelley Higgins

Contributing Author: Drew Adams, Nipun Agarwal, Abhay Agrawal, Omar Alonso, David Anniss, Sandeepan Banerjee, Mark Bauer, Ravinder Booreddy, Stephen Buxton, Yuen Chan, Sivasankaran Chandrasekar, Vincent Chao, Ravindranath Chennoju, Dan Chiba, Mark Drake, Fei Ge, Wenyun He, Thuvan Hoang, Sam Idicula, Namit Jain, Neema Jalali, Bhushan Khaladkar, Viswanathan Krishnamurthy, Muralidhar Krishnaprasad, Geoff Lee, 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, Asha Tarachandani, Tarvinder Singh, Simon Slack, Muralidhar Subramanian, Asha Tarachandani, Priya Vennapusa, James Warner

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

The Programs (which include both the software and documentation) contain proprietary information; 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. This document is not warranted to be 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.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth 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 we disclaim liability for any damages caused by such use of the Programs.

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

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

5 XML Schema Storage and Query: The Basics

6 XML Schema Storage and Query: Advanced Topics

7 XML Schema Evolution

8 Transforming and Validating XMLType Data

9 Full Text Search Over XML

Part III Using APIs for XMLType to Access and Operate on XML

10 PL/SQL API for XMLType

11 DBMS_XMLSTORE

12 Java API for XMLType

13 Using C API for XML With Oracle XML DB

17 Creating and Accessing Data Through URLs

21 PL/SQL Access and Management of Data Using DBMS_XDB

22 Java Access to Repository Data Using Resource API for Java

23 Oracle XML DB Resource Security

24 FTP, HTTP, and WebDAV Access to Repository Data

25 Writing Oracle XML DB Applications in Java

Part VI Oracle Tools that Support Oracle XML DB

26 Managing Oracle XML DB Using Oracle Enterprise Manager

C XPath and Namespace Primer

D XSLT Primer

E Java APIs: Quick Reference

F SQL and PL/SQL APIs: Quick Reference

G C API for XML (Oracle XML DB): Quick Reference

H Oracle XML DB-Supplied XML Schemas and Additional Examples

I Oracle XML DB Feature Summary

Index

Send Us Your Comments

Oracle XML DB Developer's Guide, 10*g* Release 1 (10.1)

Part No. B10790-01

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation 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 Manager
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

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

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

Preface

This manual describes Oracle XML DB, and how it stores, generates, manipulates, manages, and queries XML 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 DB repository, including versioning and security, how to access and manipulate repository resources using protocols, SQL, PL/SQL, or Java, and how to manage your Oracle XML DB application using Oracle Enterprise Manager. It also introduces you to XML messaging and Oracle Streams Advanced Queuing `XMLType` support.

This Preface contains these topics:

Intended Audience

This manual is intended for developers building XML Oracle Database applications.

Prerequisite Knowledge

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

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

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 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 does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Structure

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. It includes criteria for planning and designing your Oracle XML DB applications.

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 ways you can store, retrieve, validate, and transform XML data using Oracle Database 10*g* database native `XMLType` Application Program Interface (API).

Chapter 4, "XMLType Operations"

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

Chapter 5, "XML Schema Storage and Query: The Basics"

Describes how to use Oracle XML DB mapping from SQL to XML and back, provides an overview of how to register XML schema, deleting and updating XML schema, and how you can either use the default mapping of Oracle XML DB or specify your own.

Chapter 6, "XML Schema Storage and Query: Advanced Topics"

Describes advanced techniques for mapping from simpleType and complexType XML to SQL structures. It also describes the use of query rewrites and how to use Ordered Collections in Tables (OCTs) in Oracle XML DB.

Chapter 7, "XML Schema Evolution"

Describes how to update an XML schema registered with Oracle XML DB manually or using `DBMS_XMLSCHEMA.CopyEvolve()`.

Chapter 8, "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 entered into the database.

Chapter 9, "Full Text Search Over XML"

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

Part III, Using APIs for XMLType to Access and Operate on XML

Describes the PL/SQL and Java APIs for XMLType, as well as the C DOM API for XML, and how to use them.

Chapter 10, "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 11, "DBMS_XMLSTORE"

Describes how to use PL/SQL package `DBMS_XMLSTORE` to insert, update, and delete XML data.

Chapter 12, "Java API for XMLType"

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

Chapter 13, "Using C API for XML With Oracle XML DB"

Introduces the C API for XML used for XDK and Oracle XML DB applications. This chapter focuses on how to use C API for XML with Oracle XML DB.

Chapter 14, "Using ODP.NET With Oracle XML DB"

Describes how to use Oracle Data Provider for .NET (ODP.NET) with Oracle XML DB.

Part IV, Viewing Existing Data as XML

Chapter 15, "Generating XML Data from the Database"

Discusses SQL/XML, Oracle SQL/XML extension functions, and SQL functions for generating XML. SQL/XML functions include `XMLElement()` and `XMLForest()`. Oracle SQL/XML 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 16, "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 17, "Creating and Accessing Data Through URLs"

Introduces you to how Oracle Database works with URIs and URLs. It describes how to use `UriType` 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, security, the protocol server, and the various associated Oracle XML DB resource APIs.

Chapter 18, "Accessing Oracle XML DB Repository Data"

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 19, "Managing Oracle XML DB Resource Versions"

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

Chapter 20, "SQL Access Using 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 21, "PL/SQL Access and Management of Data Using DBMS_XDB"

Describes the Oracle XML DB resource API for PL/SQL.

Chapter 22, "Java Access to Repository Data Using Resource API for Java"

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

Chapter 23, "Oracle XML DB Resource Security"

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

Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data"

Introduces Oracle XML DB protocol server and how to use FTP, HTTP, and WebDAV with Oracle XML DB.

Chapter 25, "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

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

Chapter 26, "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 27, "Loading XML Data into Oracle XML DB Using SQL*Loader"

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

Chapter 28, "Importing and Exporting XMLType Tables"

Describes the IMPORT/EXPORT utility support for loading `XMLType` tables.

Part VII, XML Data Exchange Using Oracle Streams Advanced Queuing

Describes Oracle Streams Advanced Queuing support for XML and `XMLType` messaging.

Chapter 29, "Exchanging XML Data With Oracle Streams AQ"

Introduces how you can use Oracle Streams Advanced Queuing to exchange XML data. It briefly describes Oracle Streams, 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 APIs: Quick Reference"

Provides a summary of the Oracle XML DB Java API reference information.

Appendix F, "SQL and PL/SQL APIs: Quick Reference"
Provides a summary of the Oracle XML DB PL/SQL API reference information.

Appendix G, "C API for XML (Oracle XML DB): Quick Reference"
Provides a summary of the C API for XML reference information.

Appendix H, "Oracle XML DB-Supplied XML Schemas and Additional Examples"
Describes the `RESOURCE_VIEW` and `PATH_VIEW` structures and lists the sample resource XML schema supplied by Oracle XML DB.

Appendix I, "Oracle XML DB Feature Summary"
Provides a brief summary of Oracle XML DB features. It includes a list of standards supported and limitations.

Related Documents

For more information, see these Oracle Database resources:

- *Oracle Database New Features* for information about the differences between Oracle Database 10*g* and the Oracle Database 10*g* Enterprise Edition and the available features and options. This book also describes features new to Oracle Database 10*g* release 1 (10.1).
- *Oracle XML API Reference*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database Error Messages*. Oracle Database error message documentation is only available in HTML. If you only have access to the Oracle Database Documentation CD, you can browse the error messages by range. Once you find the specific range, use your browser's "find in page" feature to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle Database online documentation.
- *Oracle Text Application Developer's Guide*
- *Oracle Text Reference*
- *Oracle Database Concepts.*
- *Oracle Database Java Developer's Guide*
- *Oracle Database Application Developer's Guide - Fundamentals*
- *Oracle Streams Advanced Queuing User's Guide and Reference*
- *PL/SQL Packages and Types Reference*

Many 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 *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

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/membership/

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

http://otn.oracle.com/documentation/

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**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`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 start 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.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run *`old_release`*`.SQL` where *`old_release`* 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 (digits [ , precision ])`
`{ }`	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:
	■ 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 subquery;`
		
		`SELECT col1, col2, ... , coln FROM employees;`

. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	`SQL> SELECT NAME FROM V$DATAFILE;`
		`NAME`
		`----------`
		`/fsl/dbs/tbs_01.dbf`
		`/fs1/dbs/tbs_02.dbf`
		`.`
		`.`
		`.`
		`/fsl/dbs/tbs_09.dbf`
		`9 rows selected.`

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;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

`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;`

Convention	Meaning	Example

`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;`

What's New In Oracle XML DB?

This chapter describes the new features and functionality, enhancements, APIs, and product integration support added to Oracle XML DB for Oracle Database 10*g* Release 1 (10.1).

Oracle XML DB: Oracle Database 10*g* Release 1 (10.1), Enhancements

This section summarizes the Oracle XML DB enhancements provided with Oracle Database 10*g* Release 1 (10.1).

> **See Also:** Oracle Database 10*g* Release Notes Release 1 (10.1) available with your software.

Exporting and Importing XML Data

Oracle Database 10*g* Release 1 (10.1) provides enhanced IMPORT/EXPORT utility support to assist in loading XML data into Oracle XML DB. See Chapter 28, "Importing and Exporting XMLType Tables".

XML Schema Evolution Support

In prior releases an XML schema, once registered with Oracle XML DB at a particular URL, could not be modified or evolved since there may be `XMLType` tables that depend on the XML schema. There was no standard procedure for schema evolution. This release supports XML schema evolution by providing a PL/SQL procedure named `CopyEvolve()` as part of the `DBMS_XMLSCHEMA` package

DBMS_XMLGEN Now Supports Hierarchical Queries

`DBMS_XMLGEN` now supports hierarchical queries. See Chapter 15, "Generating XML Data from the Database", Generating XML from Oracle Database Using DBMS_XMLGEN on page 15-19.

Globalization Support: Character Encoding and Multibyte Characters

You can now set your client character set different from the database character set. Appropriate conversion will take place to present the XML data in the character set of the client. In addition, using FTP or HTTP, you can specify multibyte characters in the directory, filename, or URL, and you can transfer or receive data encoded in a different character set from the database. Oracle XML DB can handle all popular XML character encodings as long as the database character set supports characters in use. For full support of all valid XML characters, use UTF-8 as your database character set.

C and C++ APIs for XML

The C API for XML is used for both XDK (XML Developer's Kit) and Oracle XML DB. It is a C-based DOM API for XML and can be used to handle XML inside and outside the database. See Chapter 13, "Using C API for XML With Oracle XML DB".

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML API Reference*

SQL*Loader Supports XMLType Tables and Columns Independent of Storage

In this release, SQL*Loader supports `XMLType` tables as well as `XMLType` columns. It can load `XMLType` data regardless of whether the data is stored in LOBs or in an object-relational manner. See Chapter 27, "Loading XML Data into Oracle XML DB Using SQL*Loader".

DBMS_XMLGEN: Pretty Printing Option Can be Turned Off

`DBMS_XMLGEN` has an option to turn off pretty printing.

See Also:

`http://otn.oracle.com/tech/xml/content.html` for the latest Oracle XML DB updates and notes

Oracle Text Enhancements

This release offers the following Oracle Text enhancements:

- CTXXPATH index now supports the following Xpath expressions:
  - Positional predicates such as `/A/B[3]`
  - Attribute existence expressions such as `/A/B/@attr` or `/A/B[@attr]`
- Highlighting is now supported for `INPATH` and `HASPATH` operators for `ConText` indextype.
- The syntax for the XPath function `ora:contains` has changed.

See Also: Chapter 9, "Full Text Search Over XML"

Oracle Streams Advanced Queuing (AQ) Support

With this release, the Oracle Streams Advanced Queuing (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 the database 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 Also:

- Chapter 29, "Exchanging XML Data With Oracle Streams AQ"
- *Oracle Streams Advanced Queuing User's Guide and Reference*

Oracle XDK Support for XMLType

See Also:

- "Generating XML Using XSQL Pages Publishing Framework" on page 15-52 and "Generating XML Using XML SQL Utility (XSU)" on page 15-54
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML API Reference*

Part I

Introducing Oracle XML DB

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

1

Introducing Oracle XML DB

This chapter introduces you to Oracle XML DB. It describes Oracle XML DB features and architecture.

This chapter contains these topics:

Introducing Oracle XML DB

Oracle XML DB provides high-performance storage and retrieval of XML. It extends Oracle Database, by delivering the functionality associated with both a native XML database and a relational database. It include the following features:

- Supports the World Wide Web Consortium (W3C) XML and XML Schema data models and provides standard access methods for navigating and querying XML. It absorbs these data models into Oracle Database.
- Lets you store, query, update, transform, or otherwise process XML, while at the same time provides SQL access to the same XML data. Similarly it allows XML operations on SQL data.
- Includes a simple, light-weight XML repository that allows XML content to be organized and managed using a file/folder/URL metaphor.
- Provides a storage-independent, content-independent and programming-language-independent infrastructure for storing and managing XML data. It delivers new methods for navigating and querying XML content

stored in the database. For example, Oracle XML DB XML repository facilitates this by managing XML document hierarchies.

- Provides industry-standard methods for accessing and updating XML, including W3C XPath recommendation and the ISO-ANSI SQL/XML standard. FTP, HTTP, and WebDAV support make it possible to move XML-content in and out of Oracle Database. Industry-standard APIs allow for programmatic access and manipulation of XML content using Java, C, and PL/SQL.
- XML-specific memory management and optimizations.
- Enterprise-level Oracle Database features, such as reliability, availability, scalability, and unbreakable security for XML content.

Oracle XML DB is Not a Separate Server

Oracle XML DB is not a separate server but rather the name for a distinct group of technologies related to high-performance XML storage and retrieval available in 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.

Use XDK with Oracle XML DB

You can build applications using Oracle XML DB in conjunction with Oracle XML Developer's Kit (XDK). XDK provides common development-time utilities that can run in the middle tier in Oracle Application Server or in Oracle Database.

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide.* for more information about XDK

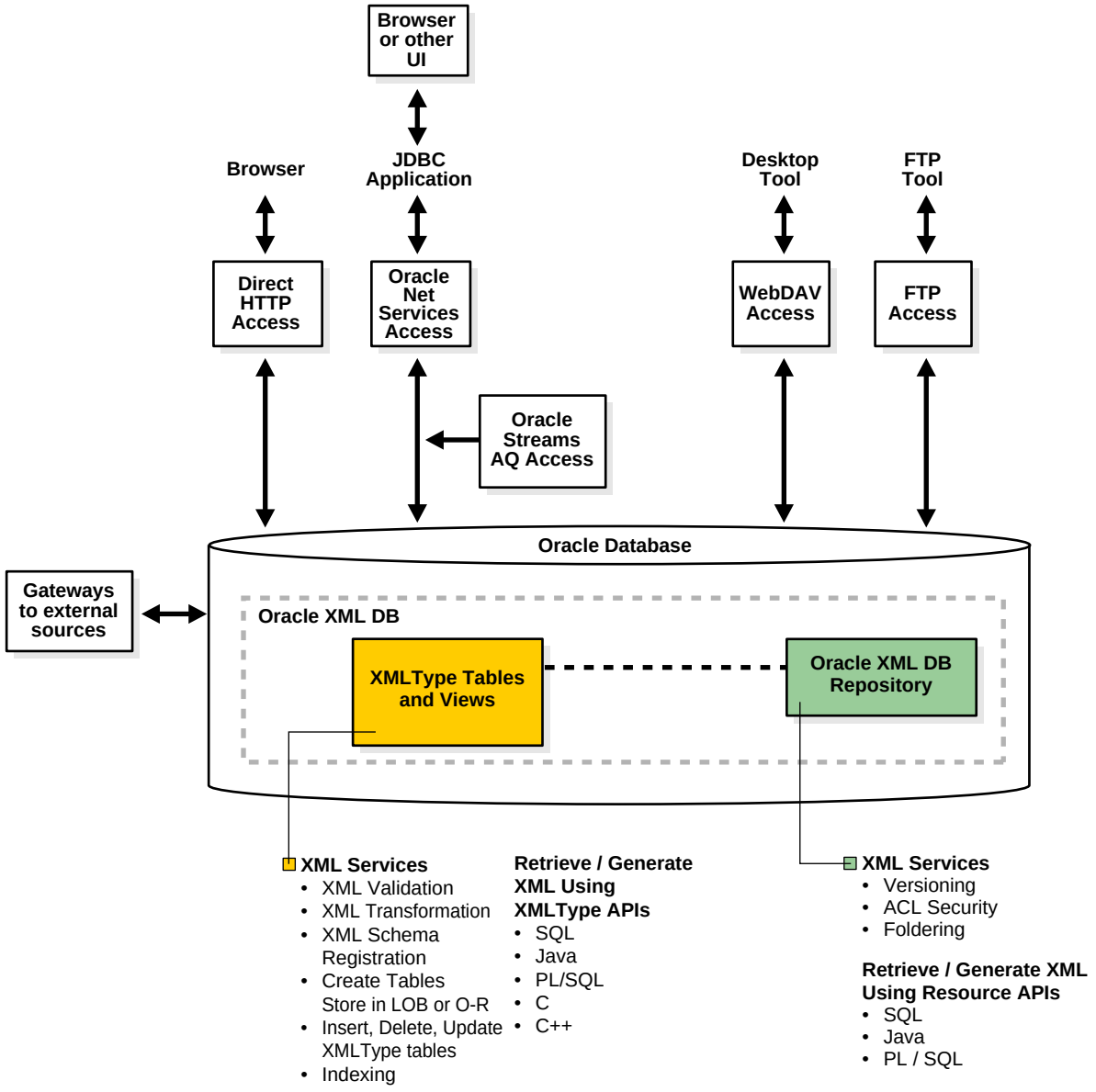
Oracle XML DB Architecture

Figure 1–1 and Figure 1–2 show Oracle XML DB architecture. The two main features in Oracle XML DB architecture are:

- `XMLType` tables and views storage
- Oracle XML DB repository, also referred to in this manual as XML repository or repository

XMLType Storage describes the architecture in more detail.

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



XMLType Storage

Figure 1–2 describes the `XMLType` tables and views storage architecture.

When XML Schema are registered with Oracle XML DB, XML elements for `XMLType` tables, tables with `XMLType` columns, and `XMLType` views, are mapped to database tables. These can be viewed and accessed in XML repository.

Data in `XMLType` tables and tables with `XMLType` columns can be stored in Character Large Objects (`CLOB`) or natively using structured XML.

Data in `XMLType` views can be stored in local tables or remote tables. The latter can be accessed through 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 Oracle Streams Advanced Queuing (AQ) and Web Services.

See Also:

- Part II. "Storing and Retrieving XML Data in Oracle XML DB"
- Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data"
- Chapter 26, "Managing Oracle XML DB Using Oracle Enterprise Manager"
- Chapter 29, "Exchanging XML Data With Oracle Streams AQ"

Oracle XML DB Repository

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

See Also: Chapter 18, "Accessing Oracle XML DB Repository Data"

The contents of Oracle XML DB repository are referred to as *resources*. These can be either containers (directories or 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.

APIs for Accessing and Manipulating XML

Figure 1–1 shows the following Oracle XML DB XML application program interfaces (APIs):

- ***Oracle XML DB Resource APIs.*** These are used to access `XMLType` and other data. In other words, to access data in 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, through `DBMS_XDB` and `DBS_XMLSTORE` APIs
  - Java through the Resource API for Java
  - C (OCI) through the C API for XML
  - Oracle Data Provider for .NET (ODP.NET)

  See Also: Part V. "Oracle XML DB Repository: Foldering, Security, and Protocols"

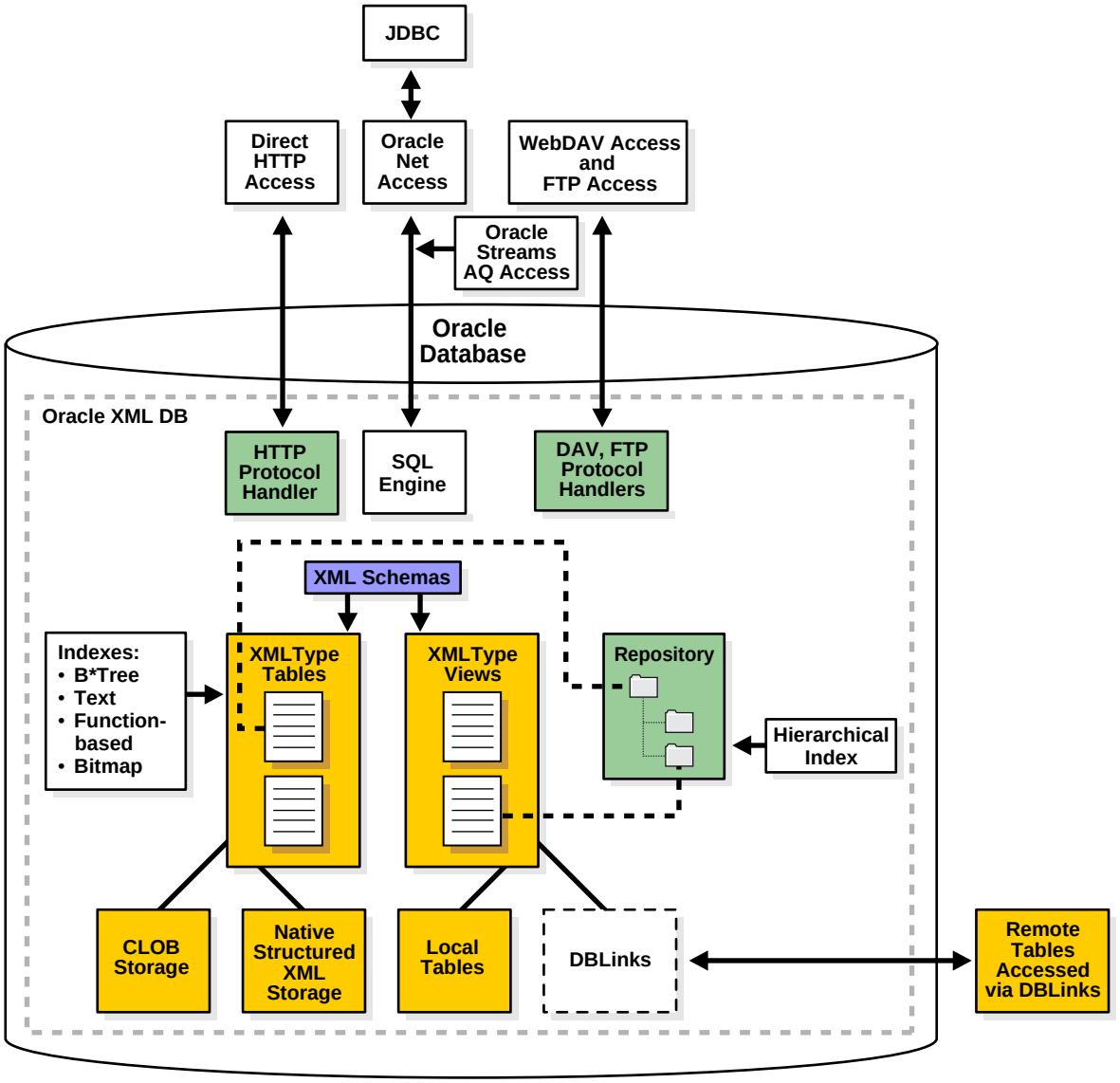
- ***Oracle XML DB Protocol Server.*** Oracle XML DB supports FTP, HTTP, and WebDAV protocols, as well as JDBC, for fast access of XML data stored in Oracle Database in `XMLType` tables and columns. See Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data".

XML Services

Besides supporting APIs that access and manipulate data Oracle XML DB repository provides API for the following services:

- **Versioning**. Oracle XML DB uses the `DBMS_XDB_VERSION` PL/SQL package for versioning resources in Oracle XML DB repository. Subsequent updates to the resource results in new versions being created while the data corresponding to previous versions is retained. Versioning support is based on the IETF WebDAV standard.
- **ACL Security.** Oracle XML DB resource security is based on the ACL (Access Control Lists) mechanism. Every resource or document in Oracle XML DB has an associated ACL that lists its privileges. Whenever resources are accessed or manipulated, the ACLs determine if the operation is legal. An ACL is an XML document that contains a set of Access Control Entries (ACE). Each ACE grants or revokes a set of permissions to a particular user or group (database role). This access control mechanism is based on the WebDAV specification.
- **Foldering.** Oracle XML DB repository foldering module manages a persistent hierarchy of containers (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.

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



XML Repository Architecture

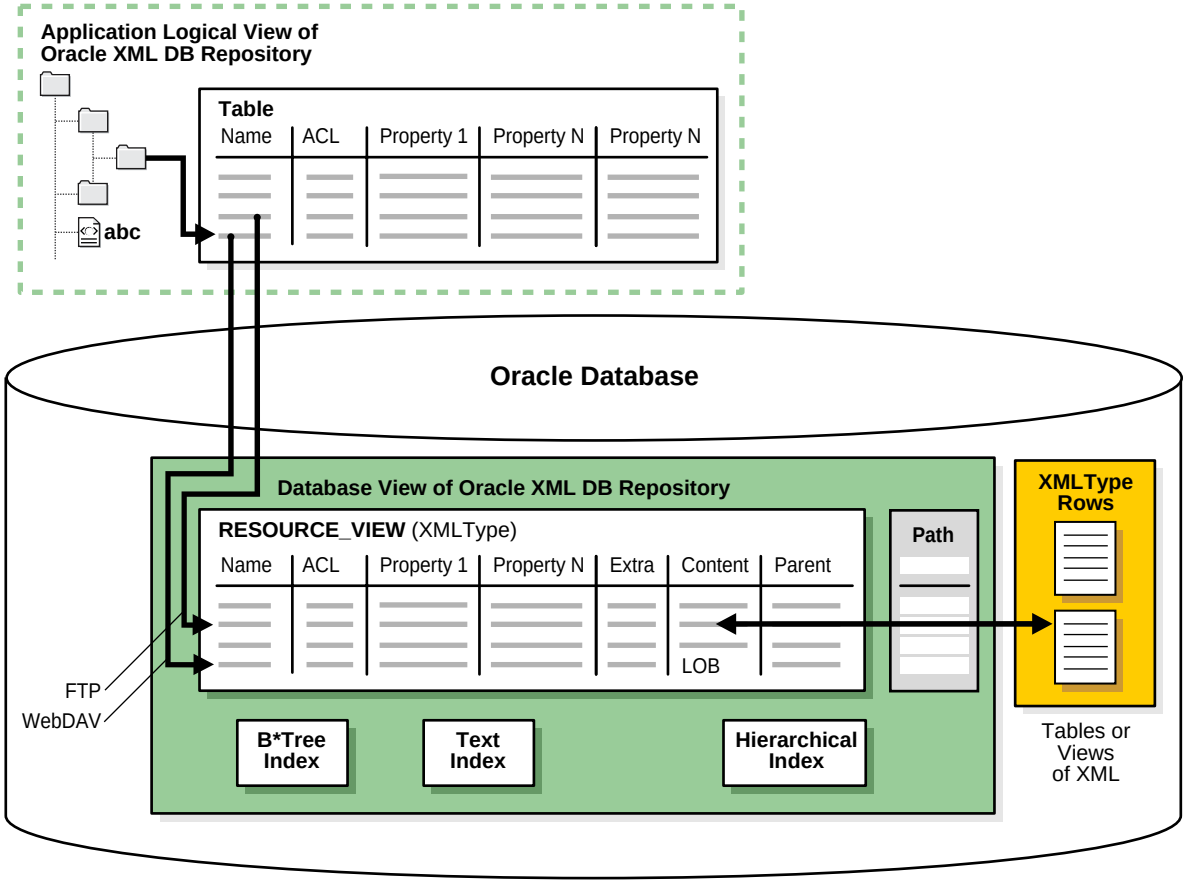
Figure 1–3 describes the Oracle XML DB repository architecture. A resource is any piece of content managed by Oracle XML DB. Each resource has a name, an associated access control list that determines who can see the resource, certain static properties, and additional properties that are extensible by the application. Applications using the repository obtain a logical view of parent-child folders. You can access this Oracle Database repository, for example, in SQL, using the `RESOURCE_VIEW` API.

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

See Also:

- Chapter 18, "Accessing Oracle XML DB Repository Data"
- Chapter 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"

Figure 1–3 Oracle XML DB Repository Architecture



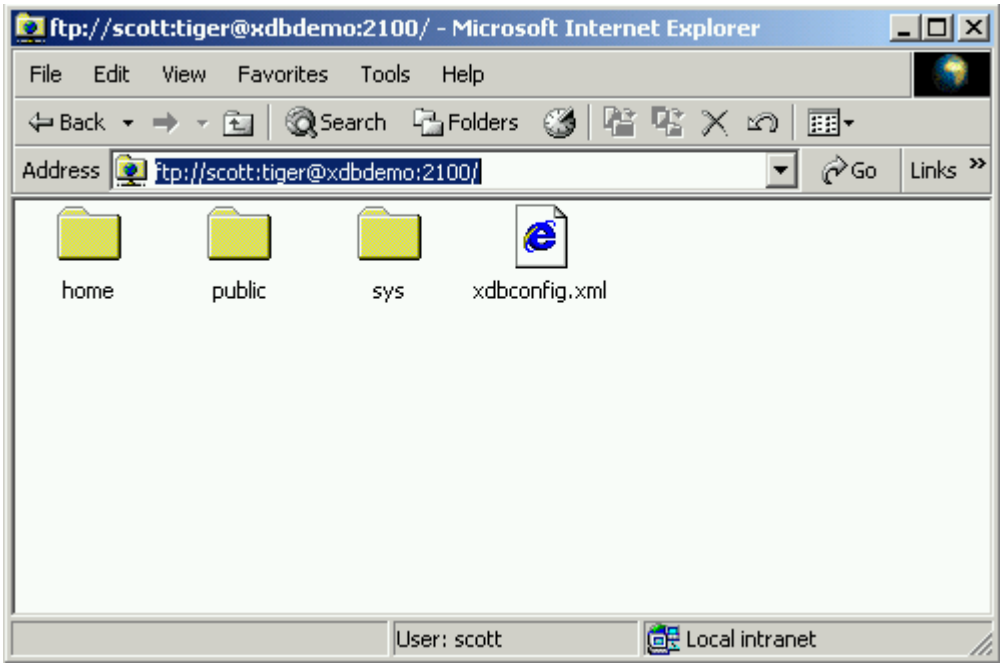
How Does Oracle XML DB Repository Work?

The relational model table-row-column metaphor, is accepted as an effective mechanism for managing structured data. The model is not as effective for managing semi-structured and unstructured data, such as document- or content-oriented XML. For example, a book is not easily represented as a set of rows in a table. It is more natural to represent a book as a hierarchy, book:chapter:section:paragraph, and to represent the hierarchy as a set of folders and subfolders.

- A hierarchical metaphor manages document-centric XML content. Relational databases are traditionally not good at managing hierarchical structures and traversing a path or URL. Oracle XML DB provides a hierarchically organized XML repository that can be queried and through which document-centric XML content can be managed.
- A hierarchical index speeds up folder and path traversals. Oracle XML DB includes a new, patented hierarchical index that speeds up folder and path traversals in Oracle XML DB repository. The hierarchical index is transparent to end users, and allows Oracle XML DB to perform folder and path traversals at speeds comparable to or faster than conventional file systems.
- Access XML documents using FTP, HTTP, and WebDAV protocols; SQL, PL/SQL, Java, and C languages. You can access XML documents in the repository using standard connect-access protocols such as FTP, HTTP, and WebDAV, in addition to languages SQL, PL/SQL, Java, and C. Oracle XML DB repository provides content authors and editors direct access to XML content stored in Oracle Database.

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

Figure 1–4 Microsoft Web Folder View of Oracle XML DB Repository



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

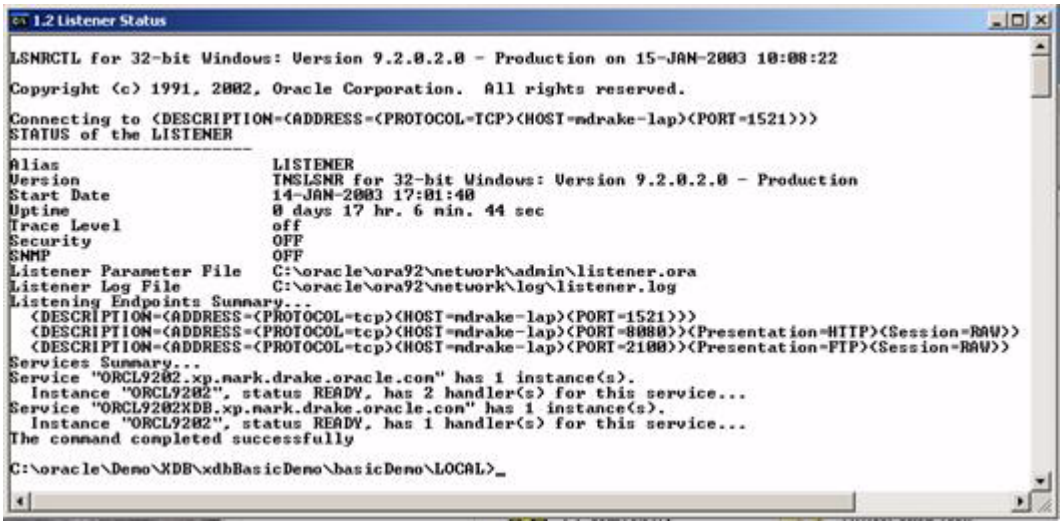
Hence, WebDAV clients such as Microsoft Windows Explorer can connect directly to XML DB repository. No additional Oracle Database or Microsoft-specific software or other complex middleware is needed. End users can work directly with Oracle XML DB repository using familiar tools and interfaces.

Oracle XML DB Protocol Architecture

One key features of the Oracle XML DB architecture is that HTTP, WebDAV, and FTP protocols are supported using the same architecture used to support Oracle Data Provider for .NET (ODP.NET) in a shared server configuration. The Listener listens for HTTP and FTP requests in the same way that it listens for ODP .NET service requests. When the listener receives an HTTP or FTP request it hands it off to an Oracle Database shared server process which services it and sends the appropriate response back to the client.

As can be seen from Figure 1–5, you can use the TNS Listener command `lsnrctl status` to verify that HTTP and FTP support has been enabled.

Figure 1–5 Listener Status with FTP and HTTP Protocol Support Enabled



See Also: Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data"

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

All Oracle XML DB functionality is accessible from C, PL/SQL, and Java. Today, the most popular methods for building web-based applications are servlets plus JSPs (Java Server Pages) and XSL plus XSPs (XML Style Sheets plus XML Server Pages). Typical API implementation includes:

- Servlets and JSPs. These APIs access data using JDBC.
- XSL/XSPs. These APIs expect data in the form of XML documents that are processed using a Document Object Model (DOM) API implementation.

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

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

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

Oracle XML DB Features

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

Handling the Hierarchical Nature of XML

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

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

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

The major features of Oracle XML DB are:

- XMLType
- XML Schema
- Structured Versus Unstructured Storage
- XML / SQL Duality
- SQL/XML ICITS Standard Operators
- XPath and XQuery Rewrite
- XMLType Storage. This was described previously on page 1-3.
- Oracle XML DB Repository. This was described previously on page 1-4.

XMLType

`XMLType` is a native server datatype that allows the database to understand that a column or table contains XML. This is similar to the way that the `DATE` datatype allows the database to understand that a column contains a date. `XMLType` also provides methods that allow common operations such as XML schema validation and XSL transformations on XML content.

You can use the `XMLType` data-type like any other datatype. For example, you can use `XMLType` when:

- Creating a column in a relational table
- Declaring PL/SQL variables
- Defining and calling PL/SQL procedures and functions

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

The following example shows creating a simple table with an `XMLType` column.

Oracle XML DB Stores XML Text in CLOBs

Oracle XML DB stores the content of the document as XML text using the Character Large Object (`CLOB`) datatype. This allows for maximum flexibility in terms of the

shape of the XML structures that can be stored in a single table or column and the highest rates of ingestion and retrieval.

XMLType Tables and Columns Can Conform to an XML Schema

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

- The database will ensure that only XML documents that validate against the XML schema can be stored in the column or table.
- Since the contents of the table or column conform to a known XML structure, Oracle XML DB can use the information contained in the XML schema to provide more intelligent query and update processing of the XML.
- Constraining the `XMLType` to an XML schema provides the option of storing the content of the document using *structured-storage* techniques. Structured-storage decomposes or 'shreds' the content of the XML document and stores it as a set of SQL objects rather than simply storing the document as text in a `CLOB`. The object-model used to store the document is automatically derived from the contents of the XML schema.

The XMLType API

The `XMLType` datatype provides the following structures:

- **Constructors**. These allow an `XMLType` value to be created from a `VARCHAR`, `CLOB`, `BLOB`, or `BFILE` value.
- **Methods**. A number of XML-specific methods that can operate on `XMLType` objects. The methods provided by `XMLType` provide support for common operations such as:
  - Extracting a subset of nodes contained in the `XMLType`, using `extract()`
  - Checking whether or not a particular node exists in the `XMLType`, using `existsNode()`
  - Validating the contents of the `XMLType` against an XML schema, using `schemaValidate()`
  - Performing an XSL Transformation, using `transform()`

See Also: Chapter 4, "XMLType Operations" and Chapter 8, "Transforming and Validating XMLType Data"

XML Schema

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

XML Schema Unifies Document and Data Modeling

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 use Oracle XML DB to enforce this data model for all your data.

You Can Create XMLType Tables and Columns, Ensure DOM Fidelity

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

Use XMLType Views to Wrap Relational Data

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 Large Objects (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 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.

W3C's Schema for Schemas

The W3C Schema Working Group publishes an XML Schema, often referred to as the "Schema for Schemas". This XML schema provides the definition, or vocabulary, of the XML Schema language. An XML schema definition (XSD) is an XML document, that is compliant with the vocabulary defined by the "Schema for Schemas". An XML schema uses vocabulary defined by W3C XML Schema Working Group to create a collection of type definitions and element declarations that declare a shared vocabulary for describing the contents and structure of a new class of XML documents.

XML Schema's Base Set of Data Types Can be Extended

The XML Schema language provides strong typing of elements and attributes. It defines 47 scalar data types. The base set of data types can be extended using object-oriented techniques like inheritance and extension to define more complex types. W3C XML Schema vocabulary also includes constructs that allow the definition of complex types, substitution groups, repeating sets, nesting, ordering, and so on. Oracle XML DB supports all of constructs defined by the XML Schema Recommendation, except for redefines.

XML schema are most commonly used as a mechanism for validating that instance documents conform with their specifications. Oracle XML DB includes methods and SQL operators that allow an XML schema to be used for this.

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

See Also: Appendix B, "XML Schema Primer" and Chapter 5, "XML Schema Storage and Query: The Basics" for more information about using XML schema and using XML schema with Oracle XML DB

Structured Versus Unstructured Storage

One key decision to make when using Oracle XML DB for persisting XML documents is when to use *structured*- and when to use *unstructured* storage.

- **Unstructured-storage** provides for the highest possible throughput when inserting and retrieving entire XML documents. It also provides the greatest degree of flexibility in terms of the structure of the XML that can be stored in a `XMLType` table or column. These throughput and flexibility benefits come at the expense of certain aspects of intelligent processing. There is little the database can do to optimize queries or updates on XML stored using a `CLOB` datatype.
- **Structured-storage** has a number of advantages for managing XML, including optimized memory management, reduced storage requirements, b-tree indexing and in-place updates. These advantages are at a cost of somewhat increased processing overhead during ingestion and retrieval and reduced flexibility in terms of the structure of the XML that can be managed by a given `XMLType` table or column.

Table 1–1 outlines the merits of structured and unstructured storage.

Table 1–1 XML Storage Options: Structured or Unstructured

	Unstructured Storage	Structured Storage
Throughput	Highest possible throughput when ingesting and retrieving the entire content of an XML document.	The decomposition process results in slightly reduced throughput when ingesting retrieving the entire content of an XML document.
Flexibility	Provides the maximum amount of flexibility in terms of the structure of the XML documents that can be stored in an `XMLType` column or table.	Limited Flexibility. Only document that conform with the XML Schema can be stored in the `XMLType` table or column. Changes to the XML Schema may require data to be unloaded and re-loaded.
XML Fidelity	Delivers Document Fidelity: Maintains the original XML byte for byte, which may be important to some applications.	DOM Fidelity: A DOM created from an XML document that has been stored in the database will be identical to a DOM created from the original document. However trailing new lines, white space characters between tags and some data formatting may be lost.
Update Operations	When any part of the document is updated the entire document must be written back to disk.	The majority of update operations can be performed using Query Rewrite. This allows in-place, piece-wise update, leading to significantly reduced response times and greater throughput.
XPath based queries	XPath operations evaluated by constructing DOM from `CLOB` and using functional evaluations. This can be very expensive when performing operations on large collections of documents.	XPath operations may be evaluated using query-rewrite, leading to significantly improved performance, particularly with large collections of documents.

Table 1–1 (Cont.) XML Storage Options: Structured or Unstructured

	Unstructured Storage	Structured Storage
SQL Constraint Support	SQL constraints are not currently available.	SQL constraints are supported.
Indexing Support	Text and function-based indexes.	B-Tree, text and function-based indexes.
Optimized Memory Management	XML operations on the document require creating a DOM from the document.	XML operations can be optimized to reduce memory requirements.

Much valuable information in an organization is in the form of semi-structured and unstructured data. Typically this data is in files stored on a file server or in a `CLOB` column inside a database. The information in these files is in proprietary- or application-specific formats. It can only be accessed through specialist tools, such as word processors or spreadsheets, or programmatically using complex, proprietary APIs. Searching across this information is limited to facilities provided by a crawler or full text indexing.

Major reasons for the rapid adoption of XML are that it allows for:

- Stronger data management
- More open access to semi-structured and unstructured content.

Replacing proprietary file formats with XML allows organizations to achieve much higher levels of reuse of their semi-structured and unstructured data. The content can be accurately described using XML Schema. The content can be easily accessed and updated using standard APIs based on DOM and XPath.

For example, information contained in an Excel spreadsheet is only accessible to the Excel program, or to a program that uses Microsoft's COM APIs. The same information, stored in an XML document is accessible to any tool that can leverage the XML programming model. Structured data on the other hand does not suffer from these limitations. Structured data is typically stored as rows in tables within a relational database. These tables are accessed and searched using the relational model and the power and openness of SQL from a variety of tools and processing engines.

XML / SQL Duality

A key objective of Oracle XML DB is to provide XML/ SQL duality. This means that the XML programmer can leverage the power of the relational model when working with XML content and the SQL programmer can leverage the flexibility of XML when working with relational content. This provides application developers with maximum flexibility, allowing them to use the most appropriate tools to solving a particular business problem.

Relational and XML Metaphors are Interchangeable: Oracle XML DB erases the traditional boundary between applications that work with structured data and those that work with semi-structured and unstructured content. With Oracle XML DB the relational and XML metaphors become interchangeable.

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

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

- Relational data can quickly and easily be converted into HTML pages. Oracle XML DB provides new SQL operators that make it possible to generate XML directly from a SQL query. The XML can be transformed into other formats, such as HTML using the database-resident XSLT processor.
- You can easily leverage all of the information contained in their XML documents without the overhead of converting back and forth between different formats. With Oracle XML DB you can access XML content using SQL queries, On-line Analytical Processing (OLAP), and Business-Intelligence/Data Warehousing operations.
- Text, spatial data, and multimedia operations can be performed on XML Content.

SQL/XML ICITS Standard Operators

Oracle XML DB provides an implementation of the majority of operators incorporated into the forthcoming SQL/XML standard. SQL/XML is defined by specifications prepared by the International Committee for Information Technology Standards (Technical Committee H2), the main standards body for developing standards for the syntax and semantics of database languages, including SQL.

See `http://www.ncits.org/tc_home/h2.htm` for more information. SQL/XML operators fall into two categories:

- The first category consists of a set of operators that make it possible to *query and access XML* content as part of normal SQL operations.
- The second category consists of a set of operators that provide an industry standard method for *generating XML* from the result of a SQL `SELECT` statement.

With these SQL/XML operators you can address XML content in any part of a SQL statement. They use XPath notation to traverse the XML structure and identify the node or nodes on which to operate. The XPath Recommendation is described in detail at `http://www.w3.org/TR/xpath`. The ability to embed XPath expressions in SQL statements greatly simplifies XML access. The following describes briefly the provided SQL/XML operators:

- `existsNode()`. This is used in the `WHERE` clause of a SQL statement to restrict the set of documents returned by a query. The `existsNode()` operator takes an XPath expression and applies it an XML document. The operator and returns true (`1`) or false (`0`) depending on whether or not the document contains a node which matches the XPath expression.
- `extract()`. This takes an XPath expression and returns the nodes that match the expression as an XML document or fragment. If only a single node matches the XPath expression, the result is a well-formed XML document. If multiple nodes match the XPath expression, the result is a document fragment.
- `extractValue()`. This takes an XPath expression and returns the corresponding leaf level node. The XPath expression passed to `extractValue()` should identify a single attribute, or an element which has precisely one text node child. The result is returned in the appropriate SQL data type.
- `updateXML()`. This allows partial updates to be made to an XML document, based on a set of XPath expressions. Each XPath expression identifies a target node in the document, and a new value for that node. The `updateXML()` operator allows multiple updates to be specified for a single XML document.
- `XMLSequence()`. This makes it possible to expose the members of a collection as a virtual table

Detailed examples of the way in which these functions are used are provided in the `PurchaseOrder` examples in Chapter 3, "Using Oracle XML DB".

XPath and XQuery Rewrite

The SQL/XML operators, and corresponding `XMLType` methods, allow XPath expressions to be used to search collections of XML documents and to access a subset of the nodes contained within an XML document

How XPath Expressions are Evaluated by Oracle XML DB

Oracle XML DB has two methods of evaluating XPath expressions that operate on XMLType columns and tables. For XML:

- **Stored using structured storage techniques**, Oracle XML DB attempts to translate the XPath expression in a SQL/XML operator into an equivalent SQL query. The SQL query references the object-relational data structures that underpin a schema-based `XMLType`. While this process is referred to as query-rewrite, it can also occur when performing `UPDATE` operations.
- **Stored using unstructured storage**, Oracle XML DB will evaluate the XPath using functional evaluation. Functional evaluation builds a DOM tree for each XML document and then resolves the XPath programmatically using the methods provided by the DOM API. If the operation involves updating the DOM tree, the entire XML document has to be written back to disc when the operation is completed.

Query-rewrites Allow Efficient Processing of SQL Containing XPath Expressions

Query-rewrites allow the database to efficiently process SQL statements containing one or more XPath expressions using conventional relational SQL. By translating the XPath expression into a conventional SQL statement, Oracle XML DB insulates the database optimizer from having to understand XPath notation and the XML data model. The database optimizer simply processes the re-written SQL statement in the same manner as other SQL statements.

This means that the database optimizer can derive an execution plan based on conventional relational algebra. This allows Oracle XML DB to leverage all the features of the database and ensure that SQL statements containing XPath expressions are executed in a highly performant and efficient manner. To sum up, there is little overhead with query-rewrites and Oracle XML DB can execute XPath-based queries at near-relational speed, while preserving the XML abstraction.

When Can Query-Rewrites Occur?

Query-rewrites are possible when:

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

What is the Query-Rewrite Process?

The query-rewrite process is described as follows:

1. Identify the set of XPath expressions included in the SQL statement.
2. Translate each XPath expression into an object relational SQL expression that references the tables, types, and attributes of the underlying SQL: 1999 object model.
3. Re-write the original SQL statement into an equivalent object relational SQL statement.
4. Pass the new SQL statement to the database optimizer for plan generation and query execution.

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

In general, functional evaluation of a SQL statement is more expensive than query-rewrite, particularly if the number of documents that needs to be processed is large. However the major advantage of functional evaluation is that it is always possible, regardless of whether or not the `XMLType` is stored using structured storage and regardless of the complexity of the XPath expression. When documents are stored using unstructured storage (in a CLOB), functional evaluation is necessary any time the `extract()`, `extractvalue()`, `updatexml()` operators are used. The `existsNode()` operator will also result in functional evaluation unless a `CTXXPATH` index or function-based index can be used to resolve the query.

Understanding the concept of query-re-write, and the conditions under which query re-write can take place, is a key step in developing Oracle XML DB applications that will deliver the required levels of scalability and performance.

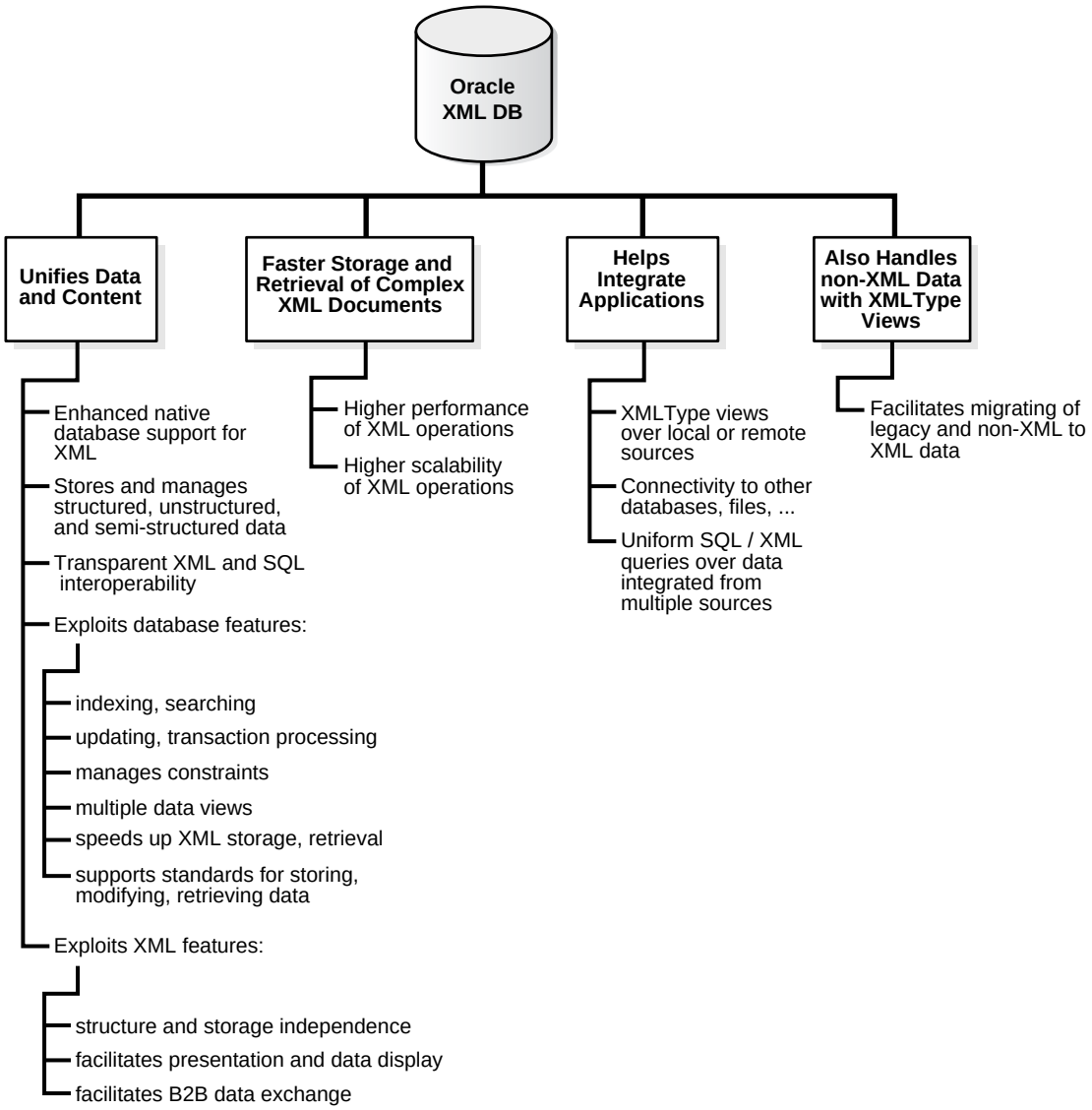
Oracle XML DB Benefits

The following sections describe several benefits for using Oracle XML DB advantages including:

- 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

Figure 1–6 summarizes the Oracle XML DB benefits.

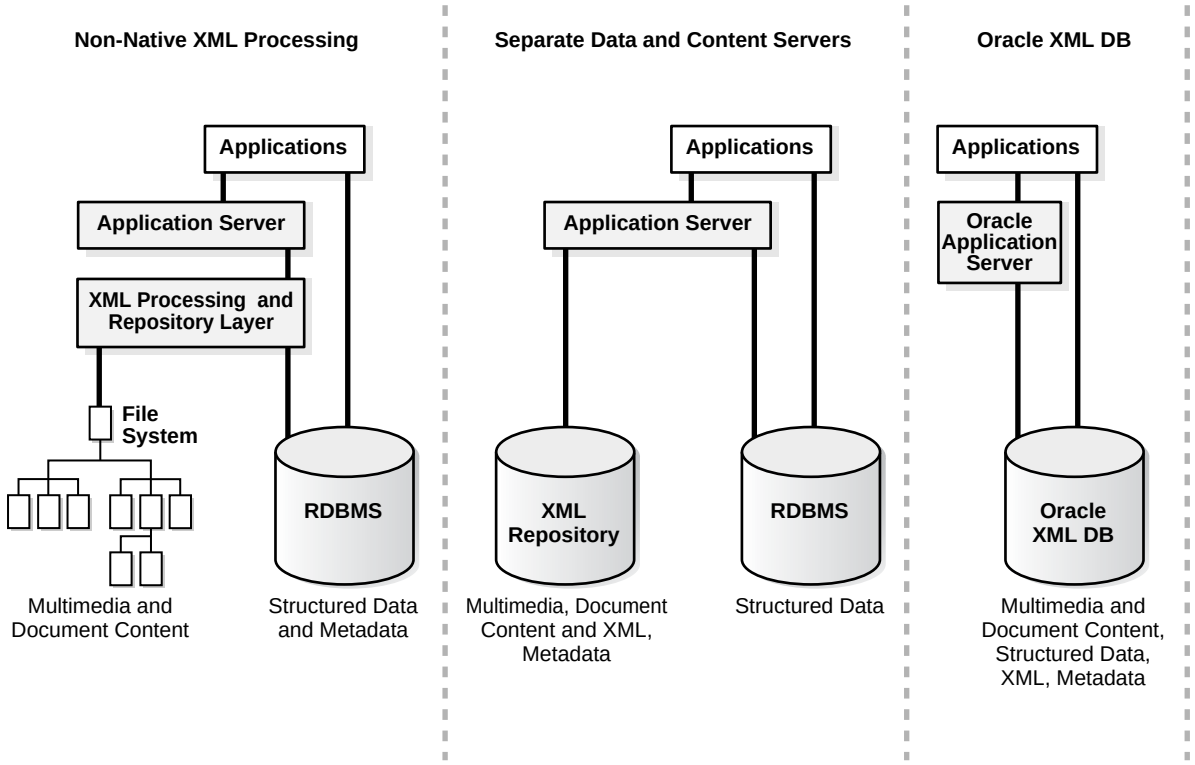
Figure 1–6 Oracle XML DB Benefits



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 1–7. Oracle XML DB is effective at accommodating XML content. It provides enhanced native support for XML.

Figure 1–7 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

Oracle Database has strong XML support with the following key capabilities:

- ***Indexing and Search***: Applications use queries such as "find all the product definitions created between March and April 2002", a query that is typically supported by a B*Tree index on a date column. Oracle XML DB can enable efficient structured searches on XML data, saving content-management vendors the need to build proprietary query APIs to handle such queries. See Chapter 4, "XMLType Operations", Chapter 9, "Full Text Search Over XML", and Chapter 15, "Generating XML Data from the Database".

- ***Updates and Transaction Processing***: Commercial relational databases use fast updates of subparts of records, with minimal contention between users trying to update. As traditionally document-centric data participate in collaborative environments through XML, this requirement becomes more important. File or `CLOB` storage cannot provide the granular concurrency control that Oracle XML DB does. See Chapter 4, "XMLType Operations".
- ***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, "XML Schema Storage and Query: The Basics" and the purchase-order examples at the end of Chapter 4, "XMLType Operations".
- ***Multiple Views of Data***: Most enterprise applications need to group data together in different ways for different modules. This is why relational views are necessary—to allow for these multiple ways to combine data. By allowing views on XML, Oracle XML DB creates different logical abstractions on XML for, say, consumption by different types of applications. See Chapter 16, "XMLType Views".
- ***Performance and Scalability***: Users expect data storage, retrieval, and query to be fast. Loading a file or `CLOB` value, and parsing, are typically slower than relational data access. Oracle XML DB dramatically speeds up XML storage and retrieval. See 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, and Java.

  See Also:

  - Chapter 12, "Java API for XMLType"
  - Chapter 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"
  - Chapter 21, "PL/SQL Access and Management of Data Using 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, "XML Schema Storage and Query: The Basics".
- ***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, "XML Schema Storage and Query: The Basics" and Chapter 18, "Accessing Oracle XML DB Repository Data".

- ***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 8, "Transforming and Validating XMLType Data".
  - Chapter 15, "Generating XML Data from the Database".
  - Chapter 16, "XMLType Views".
  - *Oracle XML Developer's Kit Programmer's Guide*, in the chapter, "XSQL Pages Publishing Framework". It includes `XMLType` examples.
- ***Ease of Interchange***: XML is the language of choice in business-to-business (B2B) data exchange. If you are forced to store XML in an arbitrary table structure, you are using some kind of proprietary translation. Whenever you translate a language, information is lost and interchange suffers. By natively understanding XML and providing DOM fidelity in the storage/retrieval process, Oracle XML DB enables a clean interchange. See:
  - Chapter 8, "Transforming and Validating XMLType Data"
  - Chapter 16, "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, "XMLType Operations".
- The lazily evaluated virtual DOM support. See Chapter 10, "PL/SQL API for XMLType".
- Database-integrated XPath and XSLT support. This support is described in several chapters, including Chapter 4, "XMLType Operations" and Chapter 8, "Transforming and Validating XMLType Data".
- XML Schema-caching support. See Chapter 5, "XML Schema Storage and Query: The Basics".
- CTXPATH Text indexing. See Chapter 9, "Full Text Search Over XML".
- The hierarchical index over the repository. See Chapter 18, "Accessing Oracle XML DB Repository Data".

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 16, "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.

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 (`CLOB` datatype) 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.

Oracle9*i* release 2 (9.2) and higher 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.
- Index type `CTXXPATH`, which allows higher performance XPath searching in Oracle XML DB under `existsNode()`.

See Also:

- Chapter 9, "Full Text Search Over XML"
- *Oracle Text Application Developer's Guide*
- *Oracle Text Reference*

Building Messaging Applications using Oracle Streams Advanced Queuing

Oracle Streams Advanced Queuing supports the use of:

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

See Also:

- *Oracle Streams Advanced Queuing User's Guide and Reference* for information about using `XMLType` with Oracle Streams Advanced Queuing
- Chapter 29, "Exchanging XML Data With Oracle Streams 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 26, "Managing Oracle XML DB Using Oracle Enterprise Manager"

Requirements for Running Oracle XML DB

Oracle XML DB is available with Oracle9*i* release 2 (9.2) and higher.

See Also:

- `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 SQL/XML functions to query XML from SQL. The task force defining these specifications falls under the auspices of the International Committee for Information Technology Standards

(INCITS). The SQL/XML specification will be fully aligned with SQL:2003. SQL/XML functions are sometimes referred to as SQLX functions.

- Java Database Connectivity (JDBC) API. JDBC access to XML is available for Java programmers.
- W3C XSL 1.0 Recommendation. You can transform XML documents at the server using XSLT.
- W3C DOM Recommendation Levels 1.0 and 2.0 Core. You can retrieve XML stored in the server as an XML DOM, for dynamic access.
- Protocol support. You can store or retrieve XML data from Oracle XML DB using standard protocols such as HTTP, FTP, and IETF WebDAV, as well as Oracle Net. See Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data".
- 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 25, "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 Oracle9*i*AS, using WSDL and UDDI. You can use Oracle Streams Advanced Queuing IDAP, the SOAP specification for queuing operations, on XML stored in Oracle Database. See Chapter 29, "Exchanging XML Data With Oracle Streams AQ" and *Oracle Streams Advanced Queuing User's Guide and Reference.*

Oracle XML DB Technical Support

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

`http://otn.oracle.com/tech/xml/`

Oracle XML DB Examples Used in This Manual

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

> **See Also:** Appendix H, "Oracle XML DB-Supplied XML Schemas and Additional Examples"

Further Oracle XML DB Case Studies and Demonstrations

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

Oracle XML DB Examples and Tutorials

You can peruse more Oracle XML DB examples on OTN:

`http://otn.oracle.com/tech/xml/index.html`

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

Oracle XML DB Case Studies and Demonstrations

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

- Oracle XML DB Downloadable Demonstration. This detailed demonstration illustrates how to use many Oracle XML DB features. Parts of this demonstration are also included in Chapter 3, "Using Oracle XML DB".
- Content Management System (CMS) application. This illustrates how you can store files on the database using Oracle XML DB repository in hierarchically organized folders, place the files under version control, provide security using ACLs, transform XML content to a desired format, search content using Oracle Text, and exchange XML messages using Oracle Streams Advanced Queueing (to request privileges on files or for sending externalization requests). See `http://otn.oracle.com/sample_code/tech/xml/xmldb/cmsxdb/content.html`.
- XML Dynamic News. This is a complete J2EE 1.3 based application that demonstrates Java and Oracle XML DB features for an online news portal. News feeds are stored and managed persistently in Oracle XML DB. Various other news portals can customize this application to provide static or dynamic news services to end users. End users can personalize their news pages by setting their preferences. The application also demonstrates the use of Model View Controller architecture and various J2EE design patterns. See `http://otn.oracle.com/sample_code/tech/xml/xmlnews/content.html`
- SAX Loader Application. This demonstrates an efficient way to break up large files containing multiple XML documents outside the database and insert them into the database as a set of separate documents. This is provided as a standalone and a web-based application.
- Oracle XML DB Utilities Package. This highlights the subprograms provided with the `XDB_Utilities` package. These subprograms operate on `BFILE` values, `CLOB` values, DOM, and Oracle XML DB Resource APIs. With this package, you can perform basic XML DB foldering operations, read and load XML files into a database, and perform basic DOM operations through PL/SQL.
- Card Payment Gateway Application. This application use s Oracle XML DB to store all your data in XML format and enables access to the resulting XML data using SQL. It illustrates how a credit card company can store its account and transaction data in the database and also maintain XML fidelity.
- Survey Application. This application determines what members want from Oracle products. OTN posts the online surveys and studies the responses. This Oracle XML DB application demonstrates how a company can create dynamic, interactive HTML forms, deploy them to the Internet, store the responses as XML, and analyze them using the XML enabled Oracle Database.

2

Getting Started with Oracle XML DB

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

This chapter contains these topics:

Installing Oracle XML DB

Oracle XML DB is installed as part of the general purpose database shipped with Oracle Database. For a manual installation or de-installation of Oracle XML DB, see Appendix A, "Installing and Configuring Oracle XML DB".

When to Use 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 Oracle 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 2-4 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 2-5.

c. Application Language

In which language(s) will you be programming your application? See "Oracle XML DB Application Design: c. Application Language" on page 2-6.

d. Processing

Will you need to generate XML? See Chapter 15, "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 8, "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). "Oracle XML DB Application Design: d. Processing Models" on page 2-7.

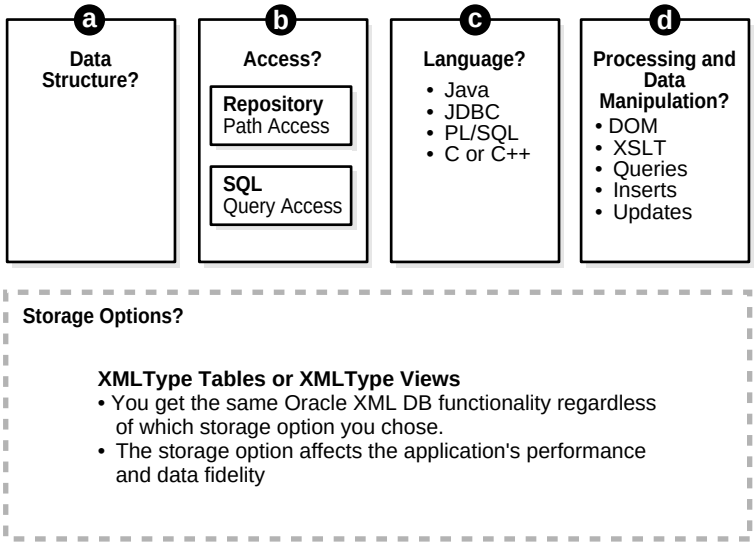
Will you be exchanging XML data with other applications, across gateways? Will you need Advanced Queuing (AQ) or SOAP compliance? See Chapter 29, "Exchanging XML Data With Oracle Streams AQ".

e. Storage

How and where will you store the data, XML data, XML schema, and so on? See "Oracle XML DB Design: e. Storage Models" on page 2-8.

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:

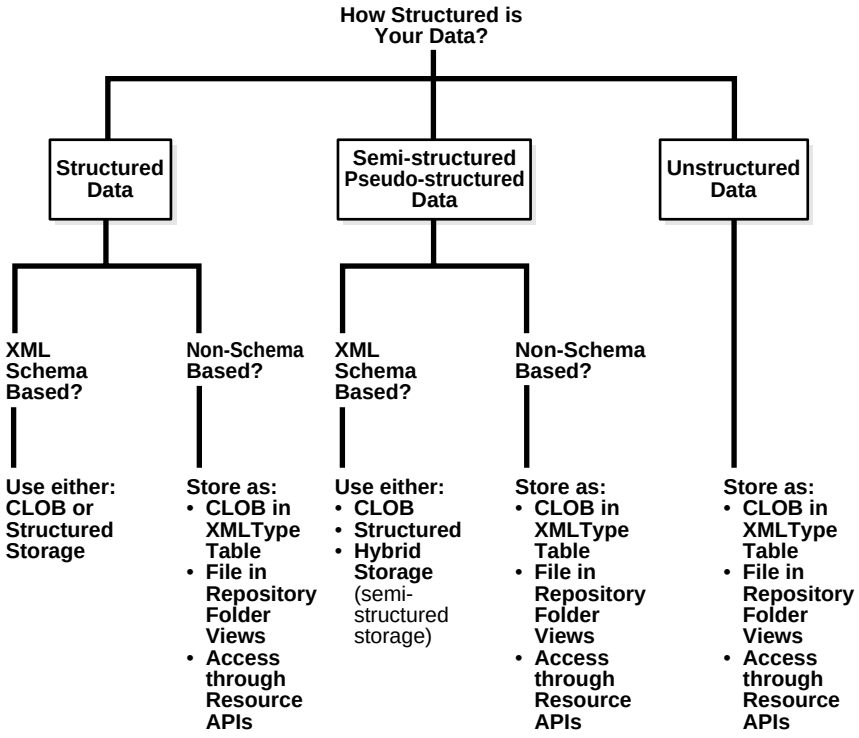
- **Structured data.** Is your data highly structured? In other words, is your data mostly XML data?
- **Semi/pseudo-structured data**. Is your data semi/pseudo-structured? In other words does your data include some XML data?
- **Unstructured data**. Is your data unstructured? In other words, is your 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: e. Storage Models" on page 2-8.
  - 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 `CLOB` values 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:

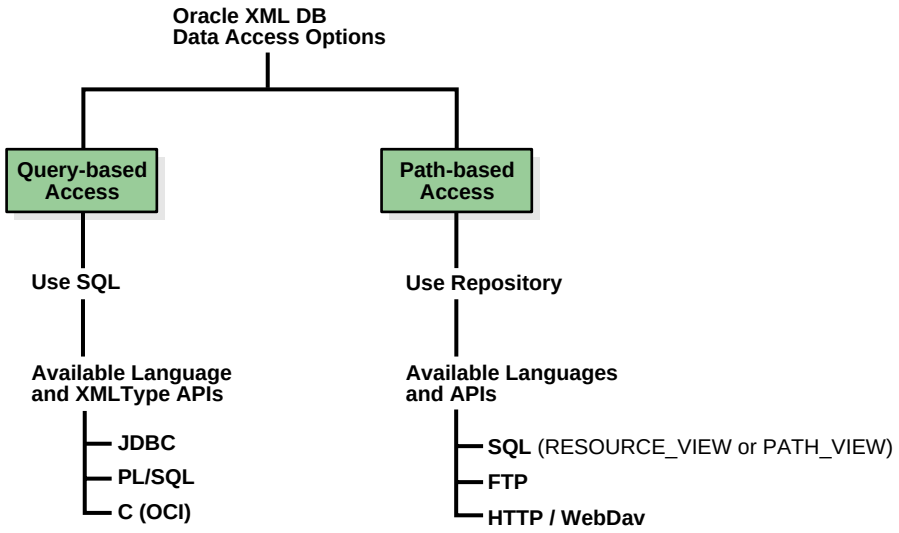
- ***Navigation- or path-based access.*** This is suitable for both content/document and data oriented applications. Oracle XML DB provides the following languages and access APIs:
  - SQL access through Resource/Path Views. See Chapter 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW".
  - PL/SQL access through `DBMS_XDB`. See Chapter 21, "PL/SQL Access and Management of Data Using DBMS_XDB".
  - Java access. See Chapter 22, "Java Access to Repository Data Using Resource API for Java".
  - Protocol-based access using HTTP/WebDAV or FTP, most suited to content-oriented applications. See Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data".
- ***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 Java API for XMLType.
  - PL/SQL access. See Chapter 10, "PL/SQL API for XMLType".

These options for accessing repository data are also discussed in Chapter 18, "Accessing Oracle XML DB Repository Data".

You can also consider the following access model criteria:

- What level of security do you need? See Chapter 23, "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, "XMLType Operations" and Chapter 9, "Full Text Search Over XML".
- Do you need to version the data? If yes, see Chapter 19, "Managing Oracle XML DB Resource Versions".

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)

  See Also:

  - Chapter 12, "Java API for XMLType"
  - Chapter 22, "Java Access to Repository Data Using Resource API for Java"
  - Chapter 25, "Writing Oracle XML DB Applications in Java"
  - Appendix E, "Java APIs: Quick Reference"

- PL/SQL

  See Also:

  - Chapter 10, "PL/SQL API for XMLType"
  - Chapter 21, "PL/SQL Access and Management of Data Using DBMS_XDB"
  - Appendix F, "SQL and 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 8, "Transforming and Validating XMLType Data".
- DOM. See Chapter 10, "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 white space. If you are using XML schema, see the discussion on DOM fidelity in Chapter 5, "XML Schema Storage and Query: The Basics".

- XPath searching. You can use XPath syntax embedded in a SQL statement or as part of an HTTP request to query XML content in the database. See Chapter 4, "XMLType Operations", Chapter 9, "Full Text Search Over XML", Chapter 18, "Accessing Oracle XML DB Repository Data", and Chapter 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW".
- XML Generation and `XMLType` views. Will you need to generate or regenerate XML? If yes, see Chapter 15, "Generating XML Data from the Database".

How often will XML documents be accessed, updated, and manipulated? See Chapter 4, "XMLType Operations" and Chapter 20, "SQL Access Using 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, "XML Schema Storage and Query: The Basics".

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

- Store more than one type of XML document in the same queue. The documents are stored internally as CLOBs.
- Selectively dequeue messages with `XMLType` attributes using the operators such as `existsNode()` and `extract()`.
- Define transformations to convert Oracle Database objects to `XMLType`.
- Define rule-based subscribers that query message content using `XMLType` operators such as `existsNode()` and `extract()`.

See Also:

- Chapter 29, "Exchanging XML Data With Oracle Streams AQ"
- *Oracle Streams Advanced Queuing User's Guide and Reference*

Oracle XML DB Design: e. 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 preservation of whitespace. This means that if you store your XML documents as `CLOB` values, 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` value.

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:

- Chapter 1, "Introducing Oracle XML DB", "XMLType Storage" on page 1-3
- Chapter 3, "Using Oracle XML DB"
- Chapter 4, "XMLType Operations"
- Chapter 5, "XML Schema Storage and Query: The Basics", "DOM Fidelity" on page 5-17

Using XMLType Tables

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

- `CLOB` (unstructured) storage
- Structured storage

- Hybrid or semi-structured storage

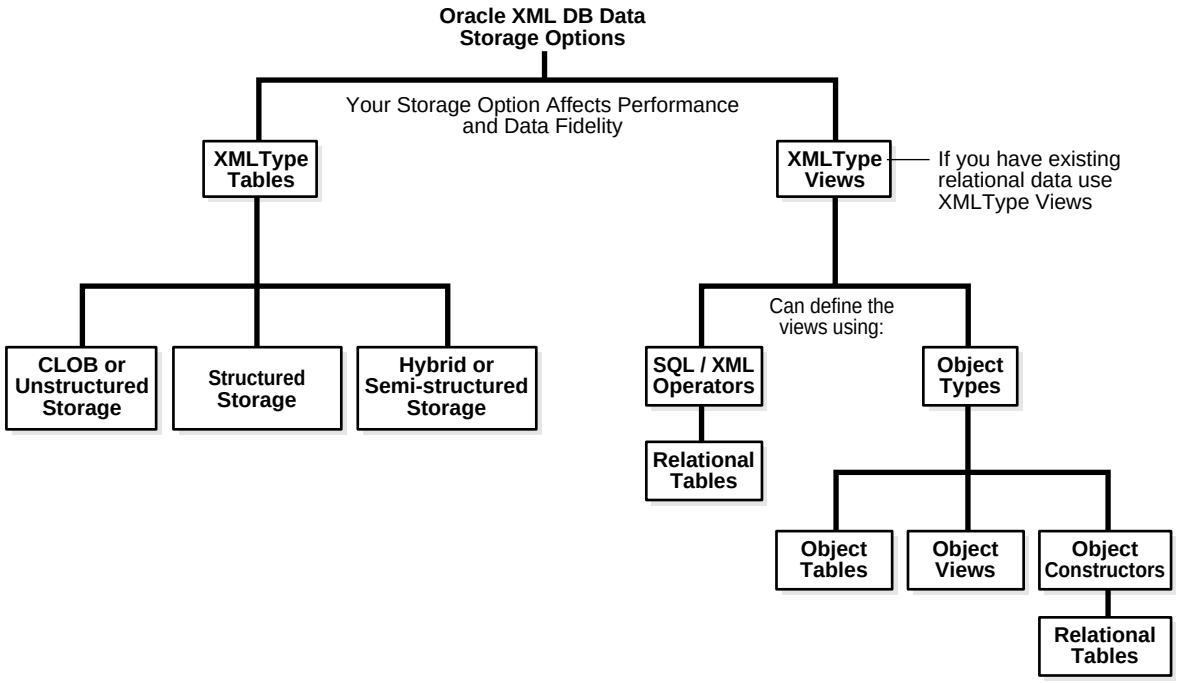
> **Note:** For `XMLType`s stored as `CLOB` values, use thin JDBC. Oracle XML DB currently only supports thin JDBC (for both XML schema- and non-schema-based applications). For `XMLType`s stored in an object-relational manner, use thick JDBC.

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 15, "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



Oracle XML DB Performance

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

- XML Storage Requirements

XML Storage Requirements

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

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

Storing Structured Documents in Oracle XML DB Saves Space

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

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

XML Memory Management

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

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

Oracle XML DB Reduces Memory Overhead for XML Schema-Based Documents by Using XML Objects (XOBs)

Oracle XML DB reduces memory overhead associated with DOM programming by managing XML schema-based XML documents using an internal in-memory structure

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

XOB Uses Lazily-Loaded Virtual DOM

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

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

XML Parsing Optimizations

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

With Oracle XML DB No Re-Parsing is Needed

Oracle XML DB eliminates the need to keep re-parsing documents. Once an XML document has been stored using structured storage techniques no further parsing is required when the document is loaded from disk into memory. Oracle XML DB is able to map directly between the on disk format and in-memory format using information derived from the associated XML schema. When changes are made to the contents of a schema-based `XMLType`, Oracle XML DB is able to write just the updated data back to disk.

Again, when the contents of the XMLType are *not* based on an XML schema a traditional DOM is used instead.

Node-Searching Optimizations

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

XML Schema Optimizations

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

Oracle XML DB Can Minimize XML Schema Overhead Once it Is Registered

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

Load Balancing Through Cached XML Schema

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

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

Reduced Non-Native Code Bottlenecks

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

Oracle XML DB Implements Java and PL/SQL APIs Over Native C

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

Reduced Java Type Conversion Bottlenecks

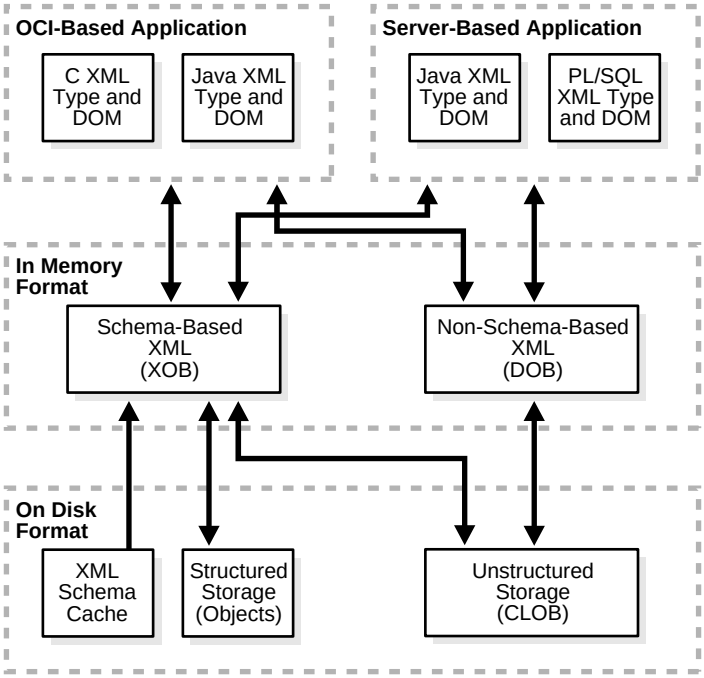
One of the biggest bottlenecks when using Java and XML is with type conversions. Internally Java uses UCS-2 to represent character data. Most XML files and databases do not contain UCS-2 encoded data. This means that all data contained in an XML file has to be converted from 8 Bit or UTF8 encoding to UCS-2 encoding before it can be manipulated in a Java program.

Oracle XML DB Uses Lazy Type Conversion to Avoid Unneeded Type Conversions

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

Consider a JSP that loads a name from the Oracle Database and prints it out in the generated HTML output. Typical JSP implementations read the name from the database (that probably contains data in the ASCII or ISO8859 character sets) convert the data to UCS-2, and return it to Java as a string. The JSP would not look at the string content, but only print it out after printing the enclosing HTML, probably converting back to the same ASCII or ISO8859 for the client browser. Oracle XML DB provides a write interface on `XMLType` so that any element can write itself directly to a stream (such as a `ServletOutputStream`) without conversion through Java character sets. Figure 2–5 shows the Oracle XML DB Application Program Interface (API) stack.

Figure 2–5 Oracle XML DB Application Program Interface (API) Stack



3

Using Oracle XML DB

This chapter provides an overview of how to use Oracle XML DB. The examples here illustrate techniques for accessing and managing XML content in purchase orders (POs). The format and data of XML POs are well suited for Oracle XML DB storage and processing techniques because POs are highly structured XML documents. However, the majority of techniques introduced here can also be used to manage other types of XML documents, such as containing non-structured or semi-structured data. This chapter also further explains Oracle XML DB concepts introduced in Chapter 1, "Introducing Oracle XML DB".

This chapter contains these topics:

Storing XML as XMLType

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

- By using Oracle XML Developer's Kit (XDK) to parse the XML document outside Oracle Database and store the XML data as rows in one or more tables in the database. In this case Oracle Database was unaware that it was managing XML content.
- By storing the XML document in Oracle Database using a Character Large Object (`CLOB`), Binary Large Object (`BLOB`), Binary File (`BFILE`), or `VARCHAR` column. Again, in this case, Oracle Database was unaware that it was managing XML content.

The introduction of Oracle XML DB and the `XMLType` datatype provides new techniques that facilitate the *persistence of XML content in the database.* These techniques include the ability to store XML documents in an `XMLType` column or table, or in Oracle XML DB repository.

Storing XML as an `XMLType` column or table makes Oracle Database aware that the content is XML. This allows the database to:

- Perform XML-specific validations, operations, and optimizations on the XML content
- Facilitate highly efficient processing of XML content by Oracle XML DB

What is XMLType

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

- `XMLType` can 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 Oracle Database.
- Functionality is also available through a set of Application Program Interfaces (APIs) provided in PL/SQL and Java.
- `XMLType` can be used in PL/SQL stored procedures as parameters, return values, and variables

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, C, Java (through JDBC), and Oracle Data Provider for .NET (ODP.NET).

The `XMLType` API provides a number of useful functions that operate on XML content. Many of these functions are provided as both SQL functions and XMLType methods. For example, the `extract()` function extracts one or more nodes from an `XMLType` instance.

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

Benefits of the XMLType Datatype and API

The `XMLType` datatype and application programming interface (API) provide significant advantages as they enable both SQL operations on XML content and XML operations on SQL content:

- ***Versatile API.*** `XMLType` has a versatile API for application development, because it includes built-in functions, indexing, and navigation support.
- ***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. Oracle Database can then determine an optimal way to run these queries.
- ***Indexing.*** Oracle XML DB lets you create Btree indexes on the object-relational tables that are used to provide structured storage of `XMLType` tables and columns. Oracle Text indexing supports text indexing of the content of structured and unstructured `XMLType` tables and columns. The `CTXXPATH` domain index type of Oracle Text provides an XML-specific text index with transactional semantics. This index type can speed up certain XPath-based searches on both structured and unstructured content. Finally, function-based indexes can be used to create indexes on explicit `XPATH` expressions for both structured and unstructured `XMLType`.

When to Use XMLType

Use `XMLType` any time you want to use the database a persistent storage of XML. For example, you can use XMLType functionality to perform the following tasks:

- SQL queries on part of or the whole XML document: The `XMLType` functions `existsNode()` and `extract()` provide the necessary SQL query functions over XML documents.
- Strong typing inside SQL statements and PL/SQL functions: The strong typing offered by `XMLType` ensures 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 queries 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`. Because Oracle 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.

There are Two Main Ways to Store XMLType Data: LOBs and Structured

`XMLType` data can be stored in two ways:

- ***In Large Objects (LOBs)***. LOB storage maintains content fidelity, that is, the original XML is preserved including whitespace. XML documents are stored composed as whole documents such as files. For non-schema-based storage, `XMLType` offers a Character Large Object (CLOB) storage option.
- ***In Structured storage (in tables and views)***. Structured storage maintains DOM (Document Object Model) fidelity.

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, because each storage option has its own benefits.

Advantages and Disadvantages of XML Storage Options in Oracle XML DB

Table 3–1 summarizes some advantages and disadvantages to consider when selecting your Oracle XML DB storage option. Storage options are also discussed in Table 1–1, " XML Storage Options: Structured or Unstructured" and Chapter 2, "Getting Started with Oracle XML DB".

Table 3–1 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 content fidelity, 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:

- When you are interested in storing and retrieving the whole document.
- When you do not need to perform piece-wise updates on XML documents.

Creating XMLType Tables and Columns

The following examples create `XMLType` columns and tables for managing XML content in Oracle Database:

Example 3–1 Creating a Table with an XMLType Column

```
CREATE TABLE example1
```

```
(
key_column VARCHAR2(10) primary key,
xml_column XMLType
);

Table created.
```

Example 3–2 Creating a Table of XMLType

```
CREATE TABLE example2 of XMLType;

Table created.
```

Loading XML Content Into Oracle XML DB

You can load XML content into Oracle XML DB using several techniques, including the following:

- Table-based loading techniques:
  - Loading XML Content into Oracle XML DB Using SQL or PL/SQL
  - Loading XML Content into Oracle XML DB Using Java
  - Loading XML Content into Oracle XML DB Using C
  - Loading Very Large XML Files of Smaller XML Documents into Oracle Database
  - Loading Large XML Files into Oracle Database Using SQL*Loader
- Path-based loading techniques:
  - Loading XML Documents into Oracle XML DB Repository
  - Loading Documents into Oracle XML DB Repository Using Protocols

Loading XML Content into Oracle XML DB Using SQL or PL/SQL

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

> **See Also:** Chapter 4, "XMLType Operations", Appendix F, "SQL and PL/SQL APIs: Quick Reference", and Oracle XML API Reference for a description of the `XMLType` constructors

The `XMLType` constructors allow an `XMLType` instance to be created from different sources including `VARCHAR` and `CLOB` datatypes. The constructors also accept additional arguments that reduce the amount of processing associated with `XMLType` creation. For example, if the source XML document is well-formed and valid, the constructor accepts flags that disable the default checking typically performed when instantiating the `XMLType`.

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

First Create a SQL Directory That Points to the Needed Directory

Before using this procedure you must create a SQL directory object that points to the directory containing the file to be processed. To do this, you must have the `CREATE ANY DIRECTORY` privilege.

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

```
CREATE DIRECTORY xmldir AS 'The path to the folder containing the XML File';
```

Example 3–3 shows how to create an `XMLType` instance from a `CLOB` value using PL/SQL procedure call `getFileContent()`. This procedure returns the content of the specified file as a `CLOB` value. It also uses the `DBMS_LOB` package to create the `CLOB` value from a `BFILE` value.

Example 3–3 Inserting XML Content into an XMLType Table

```
INSERT INTO example2
       VALUES
       (
         xmltype
         (
           bfilename('XMLDIR', 'purchaseOrder.xml'),
           nls_charset_id('AL32UTF8')
         )
       );

1 row created.
```

The following code lists the `getFileContent()` procedure definition:

```
CREATE OR REPLACE FUNCTION getFileContent(filename varchar2,
                          directoryName varchar2 default USER,
                          charset varchar2 default 'AL32UTF8')
return CLOB
is
   fileContent     CLOB := NULL;
   file            bfile := bfilename(directoryName,filename);
   dest_offset     number :=  1;
   src_offset      number := 1;
   lang_context    number := 0;
   conv_warning    number := 0;
begin
    DBMS_LOB.createTemporary(fileContent,true,DBMS_LOB.SESSION);
    DBMS_LOB.fileopen(file, DBMS_LOB.file_readonly);
    DBMS_LOB.loadClobfromFile
    (
       fileContent,
       file,
       DBMS_LOB.getLength(file),
       dest_offset,
       src_offset,
       nls_charset_id(charset),
       lang_context,
       conv_warning
    );
    DBMS_LOB.fileclose(file);
    return fileContent;
end;
/
```

> **See Also:** *Oracle Database Application Developer's Guide - Large Objects* and *PL/SQL Packages and Types Reference* for information on `DBMS_LOB` and methods used in this procedure

After calling this procedure you must dispose of the temporary `CLOB` value by calling procedure `DBMS_LOB.freeTemporary`. If the file with XML content is not stored in the same character set as the database, the character set of the file must be passed as a third argument to the `getFileContent()` procedure, so that the contents of the file are converted to the appropriate database character set as the `CLOB` value is created.

Loading XML Content into Oracle XML DB Using Java

Example 3–4 shows how to load XML content into Oracle XML DB by first creating an `XMLType` instance in Java given a Document Object Model (DOM).

Example 3–4 Inserting XML Content into an XML Type Table Using Java

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

1 row selected.
```

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

Loading XML Content into Oracle XML DB Using C

Example 3–5 shows, in C, how to insert XML content into an `XMLType` table by creating an `XMLType` instance given a DOM.

Example 3–5 Inserting XML Content into an XMLType Table Using C

```
#include <xml.h>
#include <string.h>
#include <ocixmldb.h>

OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIServer *srvhp;
OCIDuration dur;
OCISession *sesshp;

oratext *username;
oratext *password;
```

```
oratext *filename;
oratext *schemaloc;

/*---------------------------------------------------------*/
/* execute a sql statement which binds xml data */
/*---------------------------------------------------------*/

sword exec_bind_xml(OCISvcCtx *svchp, OCIError *errhp, OCIStmt *stmthp,
                    void *xml, OCIType *xmltdo, OraText *sqlstmt)
{

  OCIBind *bndhp1 = (OCIBind *) 0;
  sword  status = 0;
  OCIInd ind = OCI_IND_NOTNULL;
  OCIInd *indp = &ind;

  if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                             (ub4)strlen((const char *)sqlstmt),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    return OCI_ERROR;

  if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *) 0,
                           (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                           (ub2 *)0, (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    return OCI_ERROR;

  if(status = OCIBindObject(bndhp1, errhp, (CONST OCIType *) xmltdo,
                            (dvoid **) &xml, (ub4 *) 0,
                            (dvoid **) &indp, (ub4 *) 0))
    return OCI_ERROR;

  if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                             (ub4) OCI_DEFAULT))
    return OCI_ERROR;

  return OCI_SUCCESS;
}


/*---------------------------------------------------------*/
/* initialize oci handles and connect                      */
/*---------------------------------------------------------*/

sword init_oci_connect()
{
  sword status;
  if (OCIEnvCreate((OCIEnv **) &(envhp), (ub4) OCI_OBJECT,
                   (dvoid *) 0, (dvoid * (*)(dvoid *,size_t)) 0,
                   (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
                   (void (*)(dvoid *, dvoid *)) 0, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIEnvCreate()\n");
    return OCI_ERROR;
  }
  /* allocate error handle */
  if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &(errhp),
                     (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on errhp\n");
```

```
  return OCI_ERROR;
}

/* allocate server handle */
if (status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp,
                        (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
{

  printf("FAILED: OCIHandleAlloc() on srvhp\n");
  return OCI_ERROR;
}

/* allocate service context handle */
if (status = OCIHandleAlloc((dvoid *) envhp,
                           (dvoid **) &(svchp), (ub4) OCI_HTYPE_SVCCTX,
                           (size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on svchp\n");
  return OCI_ERROR;
}

/* allocate session handle */
if (status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &sesshp ,
                        (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on sesshp\n");
  return OCI_ERROR;

}

/* allocate statement handle */
if (OCIHandleAlloc((dvoid *)envhp, (dvoid **) &stmthp,
                  (ub4)OCI_HTYPE_STMT, (CONST size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on stmthp\n");
  return status;
}

if (status = OCIServerAttach((OCIServer *) srvhp, (OCIError *) errhp,
                            (CONST oratext *)"", 0, (ub4) OCI_DEFAULT))
{
  printf("FAILED: OCIServerAttach() on srvhp\n");
  return OCI_ERROR;
}


/* set server attribute to service context */
if (status = OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                       (dvoid *) srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER,
                       (OCIError *) errhp))
{
  printf("FAILED: OCIAttrSet() on svchp\n");
  return OCI_ERROR;
}

/* set user attribute to session */
if (status = OCIAttrSet((dvoid *)sesshp, (ub4) OCI_HTYPE_SESSION,
                       (dvoid *)username,
                       (ub4) strlen((const char *)username),
                       (ub4) OCI_ATTR_USERNAME, (OCIError *) errhp))
```

```
  {
    printf("FAILED: OCIAttrSet() on authp for user\n");
    return OCI_ERROR;
  }

  /* set password attribute to session */
  if (status = OCIAttrSet((dvoid *) sesshp, (ub4) OCI_HTYPE_SESSION,
                          (dvoid *)password,
                          (ub4) strlen((const char *)password),
                          (ub4) OCI_ATTR_PASSWORD, (OCIError *) errhp))
  {
    printf("FAILED: OCIAttrSet() on authp for password\n");
    return OCI_ERROR;
  }

  /* Begin a session  */
  if (status = OCISessionBegin((OCISvcCtx *) svchp,
                               (OCIError *) errhp,
                               (OCISession *) sesshp, (ub4) OCI_CRED_RDBMS,
                               (ub4) OCI_STMT_CACHE))
  {
    printf("FAILED: OCISessionBegin(). Make sure database is up and
            the username/password is valid. \n");
    return OCI_ERROR;
  }


  /* set session attribute to service context */
  if (status = OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                          (dvoid *)sesshp, (ub4) 0, (ub4) OCI_ATTR_SESSION,
                          (OCIError *) errhp))
  {
    printf("FAILED: OCIAttrSet() on svchp\n");
    return OCI_ERROR;
  }
}


/*-------------------------------------------------------*/
/* free oci handles and disconnect                       */
/*-------------------------------------------------------*/

void free_oci()
{

  sword  status = 0;

  /* End the session */
  if (status = OCISessionEnd((OCISvcCtx *)svchp, (OCIError *)errhp,
                             (OCISession *)sesshp, (ub4) OCI_DEFAULT))
  {
    if (envhp)
      OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
    return;
  }

  /* Detach from the server */
  if (status = OCIServerDetach((OCIServer *)srvhp, (OCIError *)errhp,
                               (ub4)OCI_DEFAULT))
  {
```

```
    if (envhp)
      OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
    return;
  }

  /* Free the handles */
  if (stmthp)
    OCIHandleFree((dvoid *)stmthp, (ub4) OCI_HTYPE_STMT);

  if (sesshp)
    OCIHandleFree((dvoid *)sesshp, (ub4) OCI_HTYPE_SESSION);


  if (svchp)
    OCIHandleFree((dvoid *)svchp, (ub4) OCI_HTYPE_SVCCTX);

  if (srvhp)
    OCIHandleFree((dvoid *)srvhp, (ub4) OCI_HTYPE_SERVER);

  if (errhp)
    OCIHandleFree((dvoid *)errhp, (ub4) OCI_HTYPE_ERROR);

  if (envhp)
    OCIHandleFree((dvoid *)envhp, (ub4) OCI_HTYPE_ENV);

  return;
}


void main()
{

    OCIType *xmltdo;

    xmldocnode  *doc;
    ocixmldbparam params[1];
    xmlerr       err;
    xmlctx  *xctx;

    oratext *ins_stmt;

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

    /* Get an xml context */
    params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
    params[0].value_ocixmldbparam = &dur;
    xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, 1);

    if (!(doc = XmlLoadDom(xctx, &err, "file", filename,
                           "schema_location", schemaloc, NULL)))
    {
       printf("Parse failed.\n");
       return;
    }
    else
       printf("Parse succeeded.\n");

    printf("The xml document is :\n");
```

```
    XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

    /* Insert the document to my_table */
    ins_stmt = (oratext *)"insert into PURCHASEORDER values (:1)";

    status = OCITypeByName(envhp, errhp, svchp, (const text *) "SYS",
                  (ub4) strlen((const char *)"SYS"), (const text *) "XMLTYPE",
                  (ub4) strlen((const char *)"XMLTYPE"), (CONST text *) 0,
                  (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                  (OCIType **) &xmltdo);

    if (status == OCI_SUCCESS)
    {
      status = exec_bind_xml(svchp, errhp, stmthp, (void *)doc,
                             xmltdo, ins_stmt);
    }

    if (status == OCI_SUCCESS)
      printf ("Insert successful\n");
    else
      printf ("Insert failed\n");

    /* free xml instances */
    if (doc)
      XmlFreeDocument((xmlctx *)xctx, (xmldocnode *)doc);
    /* free xml ctx */
    OCIXmlDbFreeXmlCtx(xctx);
    free_oci();
}

1 row selected.
```

> **See Also:** Appendix H, "Oracle XML DB-Supplied XML Schemas and Additional Examples" for a more detailed listing of this example

Loading Very Large XML Files of Smaller XML Documents into Oracle Database

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

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

The "SAX Loader Application", available on the Oracle Technology Network (OTN) site at `http://otn.oracle.com/sample_code/tech/xml/xmldb/content.html`, demonstrates how to do this.

Loading Large XML Files into Oracle Database Using SQL*Loader

Use SQL*Loader to load large amounts of XML data into Oracle Database. SQL*Loader loads in one of two modes, conventional or direct path. Table 3–2 compares these modes.

*Table 3–2 Comparing SQL*Loader Conventional and Direct Load Modes*

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

See Also:

- Chapter 27, "Loading XML Data into Oracle XML DB Using SQL*Loader"
- Example 27–1, "Loading Very Large XML Documents Into Oracle Database Using SQL*Loader" on page 27-4 for an example of direct loading of XML data.

Loading XML Documents into Oracle XML DB Repository

You can also store XML documents in Oracle XML DB repository and access these documents using path-based rather than table-based techniques. To load an XML document into Oracle XML DB repository under a given path, you can use PL/SQL package `DBMS_XDB`. This is illustrated by the following example.

Example 3–6 Inserting XML Content Into XML DB Repository Using PL/SQL DBMS_XDB

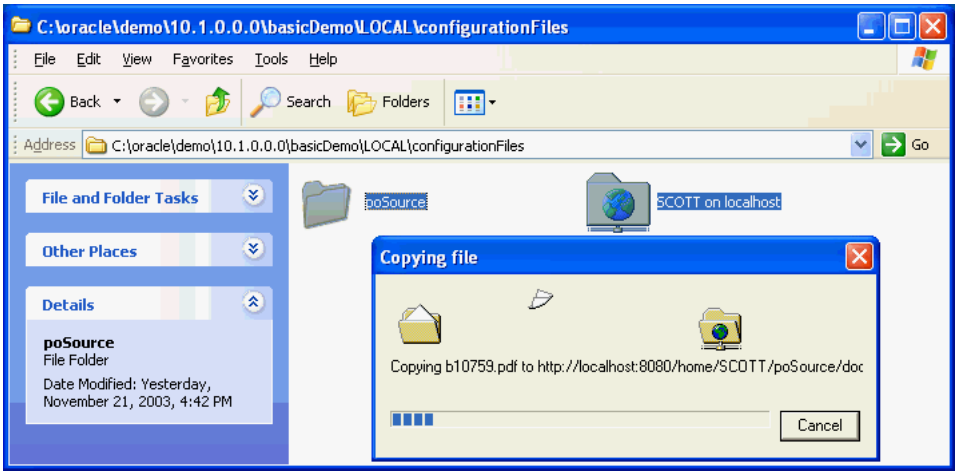
```
declare
  res boolean;
begin
  res := dbms_xdb.createResource('/home/SCOTT/purchaseOrder.xml',
                                 bfilename('XMLDIR','purchaseOrder.xml'),
                                 nls_charset_id('AL32UTF8'));
end;
/

PL/SQL procedure successfully completed.
```

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

Loading Documents into Oracle XML DB Repository Using Protocols

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

Figure 3–1 Using Windows Explorer to Load Content Into Oracle XML DB Repository



Here the folder tree contains an XML schema document, an HTML page, and a couple of XSLT style sheets.

Note: Oracle XML DB repository can also store non-XML content, such as HTML files, JPEG images, word documents, and so on, as well as both XML schema-based and non-XML schema-based XML documents.

Handling Non-ASCII XML Documents

This section describes how to load documents that are formatted in non-ASCII character sets.

XML Encoding Declaration

According to XML 1.0 Reference, each XML document is composed of units called entities. Each entity in an XML document may use a different encoding for its characters. Entities that are stored in an encoding other than UTF-8 or UTF-16 must begin with a declaration containing an encoding specification indicating which character encoding is in use. For example:

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

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

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

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

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

Loading Non-ASCII XML Documents

In releases prior to Oracle Database 10*g* Release 1, all XML documents are assumed to be in the database character set regardless of the document's encoding declaration. With Oracle Database 10*g* Release 1, the document encoding is detected from the encoding declaration when the document is loaded into the database. However, if the XML data is obtained from a `CLOB` or `VARCHAR` data type, then the encoding declaration is ignored because these two data types are always encoded in the database character set. In addition, when loading data to XML DB, either through programmatic APIs or transfer protocols, you can provide external encoding to override the document's internal encoding declaration. An error is raised if a schema-based XML document containing characters that are not legal in the determined encoding is loaded into XML DB.

The following examples show some ways which external encoding can be specified:

- Using the PL/SQL package `DBMS_XDB.CreateResource` to create a resource from a `BFILE`, you can specify the file encoding through the `csid` argument. If a zero `csid` is specified then the file encoding is auto-detected from the document's encoding declaration.

```
create directory XMLDIR as '/private/xmldir';
create or replace procedure loadXML(filename varchar2, file_csid number) is
   xbfile  bfile;
   ret     boolean;
begin
   xbfile := bfilename('XMLDIR', filename);
   ret := dbms_xdb.createResource('/public/mypo.xml', xbfile,
                                  file_csid);
end;
/
```

- When loading documents into XML DB through FTP protocol, you can specify the `quote set_charset` command to indicate the encoding of the files subsequently sent to the server.

```
FTP> quote set_charset  Shift_JIS
FTP> put mypo.xml
```

- When using the HTTP protocol, you can specify the encoding of the data transmitted to XML DB in the request header as follows:

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

Retrieving Non-ASCII XML Documents

XML documents stored in XML DB can be retrieved using transfer protocols programmatic APIs. In Oracle Database releases prior to 10*g* release 1, XML data is retrieved only in the database character set. Starting with 10*g* release 1, you can specify the encoding of the retrieved data.

The following examples show different ways to specify the output encoding:

- Using programmatic APIs, you can retrieve XML data into `VARCHAR`, `CLOB`, or `XMLType` datatypes. When using these techniques, you can control the encoding of the retrieved data by setting the `NLS_LANG` environment variable to an Oracle Database-supported character set. See the *Oracle Database Globalization Support Guide* for details on setting the `NLS_LANG` environment variable.

Also, methods are provided on the `XMLType` and `URIType` classes to retrieve XML data into a `BLOB` datatype. Using these methods, you can specify the desired character set of the returned `BLOB` value through the `csid` argument.

```
create or replace function getXML(pathname VARCHAR2, charset VARCHAR2)
                    return BLOB is
    xblob  blob;
begin
    select e.res.getBlobVal(nls_charset_id(charset)) into xblob
      from resource_view e where any_path = pathname;
   return xblob;
end;
/
```

- Using the FTP `quote set_nls_locale` command:

```
FTP> quote set_nls_locale EUC-JP
FTP> get mypo.xml
```

- Using the `Accept-Charset` parameter in the HTTP request:

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

See Also: For more information on specifying external coding:

- Controlling Character Sets for FTP on page 24-7
- Controlling Character Sets for HTTP on page 24-10
- Appendix F, "SQL and PL/SQL APIs: Quick Reference"
- *Oracle XML API Reference*

APIs Introduced in 10*g* Release 1 for Handling Non-ASCII Documents

A number of PL/SQL and Java APIs are introduced in 10*g* Release 1 to support non-ASCII documents.

See Also:

- New Java XMLType APIs on page E-9
- New PL/SQL APIs to Support XML Data in Different Character Sets on page F-22

Introducing the W3C XML Schema Recommendation

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

XML Instance Documents

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

schema. The most common use of an XML schema is to validate that a given instance document conforms to the rules defined by the XML schema.

The Schema for Schemas

The W3C Schema working group publishes an XML schema, often referred to as the "Schema for Schemas". This XML schema provides the definition, or vocabulary, of the XML Schema language. All valid XML schemas can be considered as members of the class defined by this XML schema. This means that an XML schema is an XML document that conforms to the class defined by the XML schema published at `http://www.w3.org/2001/XMLSchema`.

Editing XML Schemas

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

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

XML Schema Features

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

Text Representation of the PurchaseOrder XML Schema

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

Example 3–7 XML Schema, PurchaseOrder.xsd

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

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

```
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostCenterType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>

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

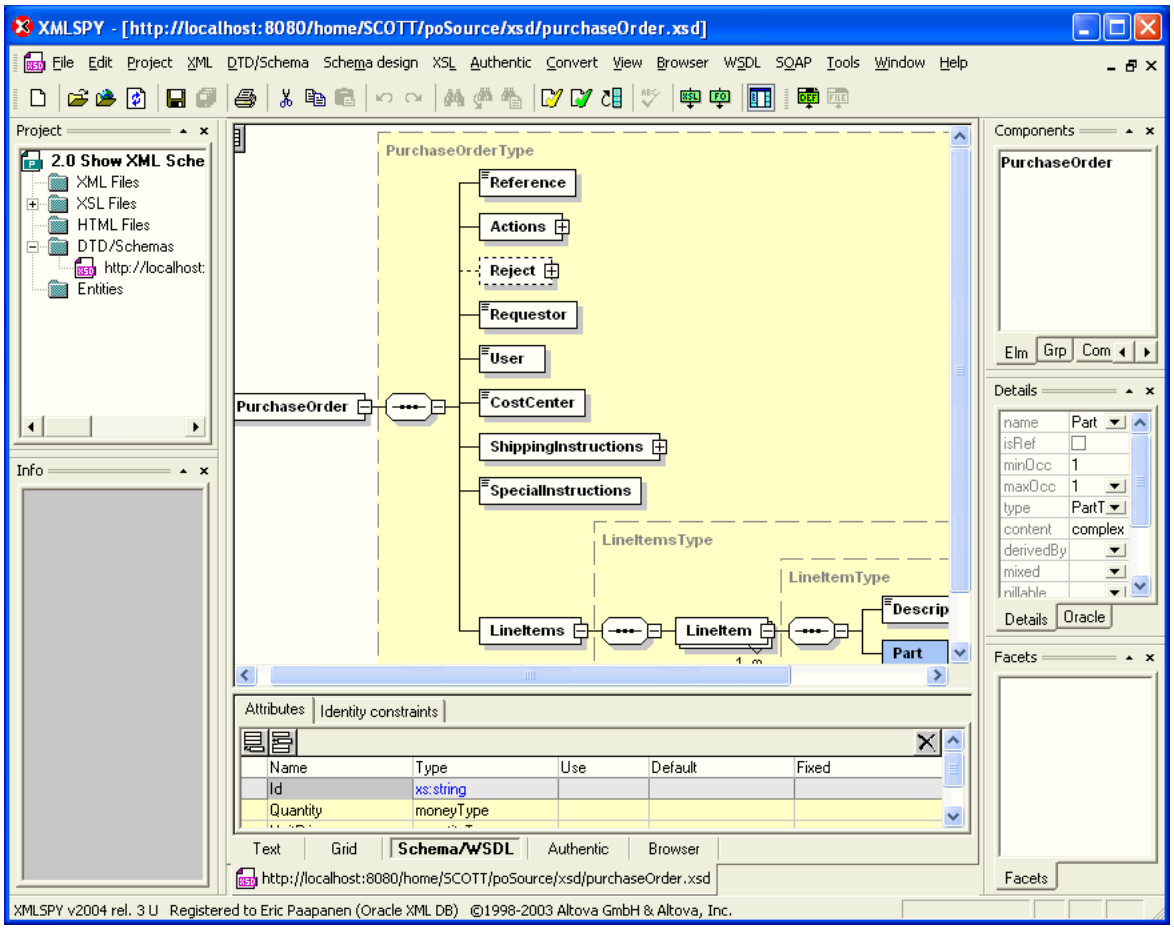
```
  <xs:simpleType name="CommentsType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DescriptionType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
 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


1 row selected.
```

> **See Also:** Appendix B, "XML Schema Primer" for a more detailed listing of `PurchaseOrder.xsd`

Graphical Representation of the PurchaseOrder XML Schema

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

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



The PurchaseOrder XML schema is a simple XML schema that demonstrates key features of a typical XML document. For example:

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

XML Schema and Oracle XML DB

XML schema are used with Oracle XML DB for a number of reasons.

Why Use XML Schema With Oracle XML DB?

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

Validating Instance Documents with XML Schema

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

Constraining Instance Documents for Business Rules or Format Compliance

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

Defining How XMLType Contents Must be Stored in the Database

Oracle XML DB also uses XML schema as a mechanism for defining how the contents of an `XMLType` should be stored inside the database. Currently Oracle XML DB provides two options:

- Unstructured storage. The content of the `XMLType` is persisted as XML text using a `CLOB` datatype. This option is available for non-schema-based and schema-based XML content**.** When the XML is to be stored and retrieved as complete documents, unstructured storage may be the best solution as it offers the fastest rates of throughput when storing and retrieving XML content.
- Structured storage. The content of the `XMLType` is persisted as a set of SQL objects. The structured storage option is only available when the `XMLType` table or column has been constrained to a global element defined by XML schema.

  If there is a need to extract or update sections of the document, perform XSL transformation on the document, or work through the DOM API, then structured storage may be the preferred storage type. Structured storage allows all these operations to take place more efficiently but at a greater overhead when storing and retrieving the entire document.

Structured Storage of XML Documents

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

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

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

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

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

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

Annotating an XML schema allows control over the naming of the SQL objects and attributes created. Annotations can also be used to override the *default* mapping between the XML schema data types and SQL data types and to specify which table should be used to store the data.

Controlling How XML Collections are Stored in the Database

Annotations are also used to control how collections in the XML are stored in the database. Currently there are four options:

- Character Large Object (CLOB). The entire set of elements is persisted as XML text stored in a CLOB column.
- VARRAY in LOB. Each element in the collection is converted into a SQL object. The collection of SQL objects is serialized and stored in a LOB column.
- VARRAY as a nested table. Each element in the collection is converted into a SQL object. The collection of SQL objects is stored as a set of rows in an Index Organized Nested Table (IOT).
- VARRAY as `XMLType`. Each element in the collection is treated as a separate `XMLType`. The collection of `XMLType` values is stored as a set of rows in an `XMLType` table.

These storage options allow you to tune the performance of applications that use `XMLType` datatypes to store XML in the database.

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

> **See Also:** Chapter 5, "XML Schema Storage and Query: The Basics"

Collections: Default Mapping

When no annotations are supplied by the user, XML DB stores collections as `VARRAY` values in a LOB.

Declaring the Oracle XML DB Namespace

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

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

The namespace is declared in the XML schema by adding a namespace declaration such as:

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

to the root element of the XML schema. Note the use of a namespace prefix. This makes it possible to abbreviate the namespace to `xdb` when adding annotations.

Example 3–8 shows the PurchaseOrder XML schema with annotations.

Example 3–8 Annotated XML Schema PurchaseOrder.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0" xdb:storeVarrayAsTable="true">
  <xs:element name="PurchaseOrder" type="PurchaseOrderType"
              xdb:defaultTable="PURCHASEORDER"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Reference" type="ReferenceType" minOccurs="1"
                  xdb:SQLName="REFERENCE"/>
      <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
      <xs:element name="Reject" type="RejectionType" minOccurs="0"
                  xdb:SQLName="REJECTION"/>
      <xs:element name="Requestor" type="RequestorType"
                  xdb:SQLName="REQUESTOR"/>
      <xs:element name="User" type="UserType" minOccurs="1"
                  xdb:SQLName="USERID"/>
      <xs:element name="CostCenter" type="CostCenterType"
                  xdb:SQLName="COST_CENTER"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                  xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                  xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
      <xs:element name="LineItems" type="LineItemsType"
                  xdb:SQLName="LINEITEMS"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
    <xs:sequence>
      <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"
                  xdb:SQLName="LINEITEM" xdb:SQLCollType="LINEITEM_V"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
    <xs:sequence>
      <xs:element name="Description" type="DescriptionType"
                  xdb:SQLName="DESCRIPTION"/>
      <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
    </xs:sequence>
    <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                  xdb:SQLType="NUMBER"/>
  </xs:complexType>
  <xs:complexType name="PartType" xdb:SQLType="PART_T">
    <xs:attribute name="Id" xdb:SQLName="PART_NUMBER" xdb:SQLType="VARCHAR2">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:minLength value="10"/>
          <xs:maxLength value="14"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="Quantity" type="moneyType" xdb:SQLName="QUANTITY"/>
    <xs:attribute name="UnitPrice" type="quantityType" xdb:SQLName="UNITPRICE"/>
  </xs:complexType>
  <xs:simpleType name="ReferenceType">
    <xs:restriction base="xs:string">
      <xs:minLength value="18"/>
      <xs:maxLength value="30"/>
    </xs:restriction>
```

```
</xs:simpleType>
<xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION"
                xdb:SQLCollType="ACTION_V">
      <xs:complexType xdb:SQLType="ACTION_T">
        <xs:sequence>
          <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
          <xs:element name="Date" type="DateType" minOccurs="0"
                      xdb:SQLName="DATE_ACTIONED"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0"
                xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0"
                xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0"
                xdb:SQLName="REASON_REJECTED"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType"
                xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
  <xs:sequence>
    <xs:element name="name" type="NameType" minOccurs="0"
                xdb:SQLName="SHIP_TO_NAME"/>
    <xs:element name="address" type="AddressType" minOccurs="0"
                xdb:SQLName="SHIP_TO_ADDRESS"/>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0"
                xdb:SQLName="SHIP_TO_PHONE"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
```

```
  <xs:simpleType name="CostCenterType">
    <xs:restriction base="xs:string">

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


1 row selected.
```

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

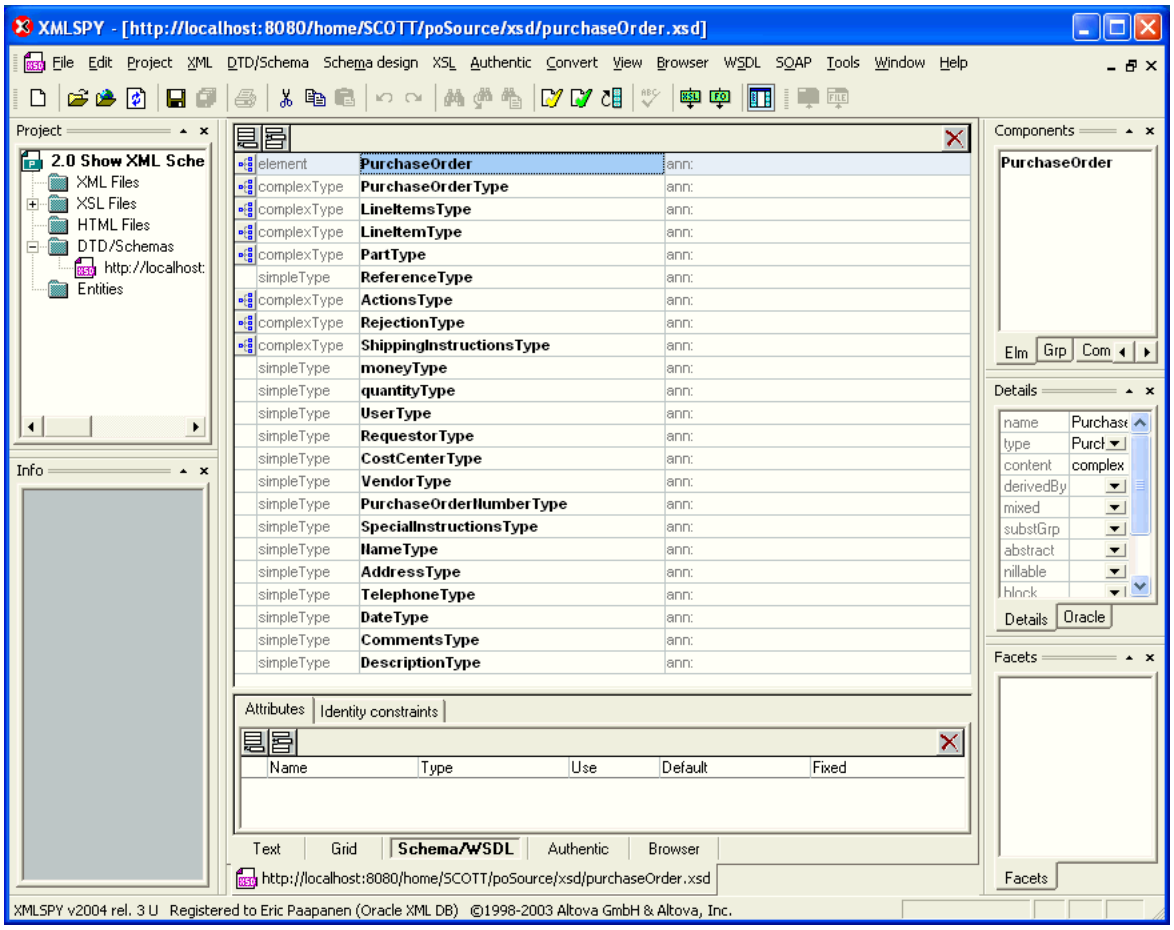
- `http://www.w3c.org/2001/XMLSchema`. This is reserved by W3C for the Schema for Schemas.
- `http://xmlns.oracle.com/xdb`. This is reserved by Oracle for the Oracle XML DB schema annotations.

The `PurchaseOrder` XML schema also uses the following annotations:

- `defaultTable` annotation in the `PurchaseOrder` element. This specifies that XML documents, compliant with this XML schema are stored in a table called `PURCHASEORDER`.
- `SQLType` annotation. The first occurrence of `SQLType` specifies that the name of the SQL type generated from complexType `PurchaseOrderType` is `PURCHASEORDER_T`.
- `SQLName` annotation. This provides an explicit name for the each SQL attribute of `PURCHASEORDER_T`.
- `SQLType` annotation. The second occurrence of `SQLType` specifies that the name of the SQL type generated from the complexType `LineItemType` is `LINEITEM_T` and the SQL type that manages the collection of `LineItem` elements is `LINEITEM_V`.

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

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



Registering an XML Schema with Oracle XML DB

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

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

- The XML schema document
- A string that can be used as a unique identifier for the XML schema, once it is registered with the database. Instance documents use this unique identifier to identify themselves as members of the class defined by the XML schema. The identifier is typically in the form of a URL, and often referred to as the *Schema Location Hint*.

XML schema registration is performed using a simple PL/SQL procedure, `dbms_xmlschema.registerschema()`. See Example 3–9. By default, when an XML schema is registered, Oracle XML DB automatically generates all the SQL object types and `XMLType` tables required to manage the instance documents.

XML schemas can be registered as global or local. See Chapter 5, "XML Schema Storage and Query: The Basics" for a discussion of the differences between global and local schemas.

Example 3–9 Using the DBMS_XMLSCHEMA Package to Register an XML Schema

```
begin
dbms_xmlschema.registerSchema
(
  'http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.xsd',
  xdbURIType('/home/SCOTT/poSource/xsd/purchaseOrder.xsd').getClob(),
  TRUE,TRUE,FALSE,TRUE
);
end;
/

PL/SQL procedure successfully completed.
```

In this example the unique identifier for the XML schema is: `http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.xsd`

The XML schema document was previously loaded into Oracle XML DB repository at the path: `/home/SCOTT/poSource/xsd/purchaseOrder.xsd`.

During XML schema registration, an `XDBUriType` accesses the content of the XML schema document, based on its location in the repository. Flags passed to the `registerSchema()` procedure specify that the XML schema must be registered as a local schema and that SQL objects and tables must be generated by the registration process.

`registerSchema()` performs the following operations:

- Parses and validates the XML schema
- Creates a set of entries in Oracle Data Dictionary that describe the XML schema
- Creates a set of SQL object definitions, based on `complexTypes` defined in the XML schema

  Creates an `XMLType` table for each global element defined by the XML schema

SQL Types and Tables Created During XML Schema Registration

Example 3–10 illustrates the creation of object types during XML schema registration with Oracle XML DB.

Example 3–10 Objects Created During XML Schema Registration

```
describe PURCHASEORDER_T
 PURCHASEORDER_T is NOT FINAL
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------------------------------------
 SYS_XDBPD$                                         XDB.XDB$RAW_LIST_T
 REFERENCE                                          VARCHAR2(30 CHAR)
 ACTIONS                                            ACTIONS_T
 REJECTION                                          REJECTION_T
 REQUESTOR                                          VARCHAR2(128 CHAR)
 USERID                                             VARCHAR2(10 CHAR)
 COST_CENTER                                        VARCHAR2(4 CHAR)
 SHIPPING_INSTRUCTIONS                              SHIPPING_INSTRUCTIONS_T
 SPECIAL_INSTRUCTIONS                               VARCHAR2(2048 CHAR)
 LINEITEMS                                          LINEITEMS_T

--
desc LINEITEMS_T
 LINEITEMS_T is NOT FINAL
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------------------------------------
```

```
 SYS_XDBPD$                                        XDB.XDB$RAW_LIST_T
 LINEITEM                                          LINEITEM_V

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

These examples show that SQL type definitions were created when the XML schema was registered with Oracle XML DB. These SQL type definitions include:

- `PURCHASEORDER_T`. This type is used to persist the SQL objects generated from a `PurchaseOrder` element. When an XML document containing a `PurchaseOrder` element is stored in Oracle XML DB the document is 'shredded' (or broken up) and the contents of the document are stored as an instance of `PURCHASEORDER_T`.
- `LINEITEMS_T`, `LINEITEM_V`, and `LINEITEM_T`. These types manage the collection of `LineItem` elements that may be present in a `PurchaseOrder` document. `LINEITEMS_T` consists of a single attribute `LINEITEM`, defined as an instance of `LINEITEM_V` type. `LINEITEM_V` is defined as a VARRAY of `LINTEITEM_T` objects. There is one instance of the `LINEITEM_T` object for each `LineItem` element in the document.

Working with Large XML Schemas

A number of issues can arise when working with large, complex XML schemas.

Sometimes the error `ORA-01792: maximum number of columns in a table or view is 1000` is encountered when registering an XML schema or creating a table based on a global element defined by an XML schema. This error occurs when an attempt is made to create an `XMLType` table or column based on a global element and the global element is defined as a `complexType` that contains a very large number of element and attribute definitions.

The error only occurs when creating an `XMLType` table or column that uses object-relational storage. When object-relational storage is selected the `XMLType` is persisted as a SQL type. When a table or column is based on a SQL type, each attribute defined by the type counts as a column in the underlying table. If the SQL type contains attributes that are based on other SQL types, the attributes defined by those types also count as columns in the underlying table. If the total number of attributes in all the SQL types exceeds the Oracle Database limit of 1000 columns in a table the storage table cannot be created.

This means that as the total number of elements and attributes defined by a `complexType` approaches 1000, it is no longer possible to create a single table that can manage the SQL objects generated when an instance of the type is stored in the database.

To resolve this you must reduce the total number of attributes in the SQL types that are used to create the storage tables. Looking at the schema there are two approaches for achieving this:

- Using a top-down technique with multiple `XMLType` tables that manage the XML documents. This technique reduces the number of SQL attributes in the SQL type

hierarchy for a given storage table. As long as none of the tables have to manage more than 1000 attributes, the problem is resolved.

- Using a bottom-up technique that reduces the number of SQL attributes in the SQL type hierarchy, collapsing some of elements and attributes defined by the XML schema so that they are stored as a single `CLOB` value.

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

For the top-down technique, annotations, `SQLInline`="false" and `defaultTable`, force some sub-elements in the XML document to be stored as rows in a separate `XMLType` table. Oracle XML DB maintains the relationship between the two tables using a REF of `XMLType`. Good candidates for this approach are XML schemas that define a choice where each element within the choice is defined as a `complexType`, or where the XML schema defines an element based on a `complexType` that contains a very large number of element and attribute definitions.

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

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

Working with Global Elements

When an XML schema is registered with the database Oracle XML DB generates a default table for each global element defined by the XML schema. If an XML schema contains a large number of global element definitions it can cause significant overhead in processor time and space used. There are two ways to avoid this:

- Add the annotation `xdb:defaultTable=""` to every global element that does not appear as the root element of an instance document.
- Set the `genTables` parameter to FALSE when registering the XML schema and then manually create the default table for each global element that can legally appear as the root element of an instance document.

Creating XML Schema-Based XMLType Columns and Tables

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

Example 3–11 shows how to manually create the `PurchaseOrder` table, the default table for `PurchaseOrder` elements, as defined by the `PurchaseOrder` XML schema.

Example 3–11 Creating an XMLType Table that Conforms to an XML Schema

```
CREATE TABLE PurchaseOrder of XMLType
XMLSCHEMA "http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.xsd"
ELEMENT "PurchaseOrder"
varray "XMLDATA"."ACTIONS"."ACTION"
STORE AS table ACTION_TABLE
(

(primary key (NESTED_TABLE_ID, ARRAY_INDEX))
 organization index overflow
```

```
)

varray "XMLDATA"."LINEITEMS"."LINEITEM"
store as table LINEITEM_TABLE
(

(primary key (NESTED_TABLE_ID, ARRAY_INDEX))
 organization index overflow
);

Table created.
```

In this example each member of the VARRAY that manages the collection of `LineItem` elements is stored as a row in nested table `LINEITEM_TABLE`. Each member of the VARRAY that manages the collection of `Action` elements is stored in the nested table `ACTION_TABLE`. The nested tables are index organized and automatically contain the `NESTED_TABLE_ID` and `ARRAY_INDEX` columns required to link them back to the parent column.

The `CREATE TABLE` statement is equivalent to the `CREATE TABLE` statement automatically generated by Oracle XML DB if the schema annotation `storeVarrayAsTable="true"` was included in the root element of the `PurchaseOrder` XML schema. Note that when this annotation is used to create nested tables, the nested tables are given system-generated names. Since these names are somewhat difficult to work with, nested tables generated by the XML schema registration process can be given more meaningful names using the SQL statement, `RENAME TABLE`.

A SQL*Plus `DESCRIBE` statement, abbreviated to `desc`, can be used to view information about an `XMLType` table.

Example 3–12 Using DESCRIBE for an XML Schema-Based XMLType Table

```
desc PURCHASEORDER
 Name                          Null?    Type
 ----------------------------- -------------------------------------------------
TABLE of SYS.XMLTYPE(XMLSchema
 "http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.xsd" Element
 "PurchaseOrder") STORAGE Object-relational TYPE "PURCHASEORDER_T"
```

The output of the `DESCRIBE` statement shows the following information about the `PurchaseOrder` table:

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

Default Tables

The XML schema in Example 3–11 specifies that the `PurchaseOrder` table is the default table for `PurchaseOrder` elements. This means that when an XML document compliant with the XML schema, is inserted into Oracle XML DB repository using protocols or PL/SQL, the content of the XML document is stored as a row in the `PurchaseOrder` table.

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

Identifying Instance Documents

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

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

The advantage of the `XMLSchema-instance` mechanism is that it allows the Oracle XML DB protocol servers to recognize that an XML document inserted into Oracle XML DB repository is an instance of a registered XML schema. This means that the content of the instance document is automatically stored in the default table defined by that XML schema.

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

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

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

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

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

noNamespaceSchemaLocation Attribute

If the target XML schema does not declare a target namespace, the `noNamespaceSchemaLocation` attribute is used to identify the XML schema. The value of the attribute is called the *Schema Location Hint*. This is the unique identifier passed to `dbms_xmlschema.registerSchema()` when the XML schema is registered with the database.

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

```
<PurchaseOrder
```

```
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:noNamespaceSchemaLocation="http://localhost:8080/home/SCOTT/poSource/xsd/purch
aseOrder.xsd">
```

schemaLocation Attribute

If the target XML schema declares a target namespace then the `schemaLocation` attribute is used to identify the XML schema. The value of the attribute is a pair of values separated by a space. The left hand side of the pair is the value of the target namespace declared in the XML schema. The right hand side of the pair is the *Schema Location Hint*, the unique identifier passed to `dbms_xmlschema.registerSchema()` when the XML schema is registered with the database.

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

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

and in this case the correct form of the root element of the instance document would read as follows:

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

Dealing with Multiple Namespaces

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

Using the Database to Enforce XML Data Integrity

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

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

that table or column. Any attempt to store or insert any other kind of XML document in an XML schema-based `XMLType` causes an ORA-19007 error. Example 3–13 illustrates this.

Example 3–13 ORA-19007 Error From Attempting to Insert an Incorrect XML Document

```
INSERT INTO PURCHASEORDER
       VALUES
       (
         XMLType
          (
             bfilename('XMLDIR','Invoice.xml'),
             nls_charset_id('AL32UTF8')
          )
       );
INSERT INTO PURCHASEORDER
            *
ERROR at line 1:
ORA-19007: Schema - does not match expected
 http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.xsd.
```

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

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

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

> **Note:** XML constraints are enforced within XML documents whereas database (SQL) constraints are enforced across sets of XML documents.

Comparing Partial to Full XML Schema Validation

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

Partial Validation

When an XML document is inserted into an XML schema-based `XMLType` table or column Oracle XML DB performs a partial validation of the document. A partial validation ensures that all the mandatory elements and attributes are present and that there are no unexpected elements or attributes in the document. It ensures that the structure of the XML document conforms to the SQL type definitions that were derived from the XML schema. However, it does not ensure that the instance document is fully compliant with the XML schema. Example 3–14 provides an example of failing a partial validation while inserting an XML document into table `PurchaseOrder`:

Example 3–14 ORA-19007 When Inserting Incorrect XML Document (Partial Validation)

```
INSERT INTO PURCHASEORDER
       VALUES
       (
         XMLType
          (
            bfilename('XMLDIR','InvalidElement.xml'),
            nls_charset_id('AL32UTF8')
          )
        );
         XMLType
         *
ERROR at line 4:
ORA-30937: No schema definition for 'UserName' (namespace '##local') in parent
 'PurchaseOrder'
ORA-06512: at "SYS.XMLTYPE", line 259
ORA-06512: at "SYS.XMLTYPE", line 284
ORA-06512: at line 1
```

Full Validation

When full validation of the instance document against the XML schema is required, you can enable XML schema validation using either of the following:

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

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

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

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

Full XML Schema Validation Costs CPU and Memory Usage Full XML Schema validation costs CPU and memory. By leaving the decision on whether or not to force a full XML schema validation to you, Oracle XML DB lets you perform full XML schema validation only when necessary. If you can rely on the application validating the XML document, you can obtain higher overall throughput by avoiding overhead associated with a full validation. If you cannot be sure about the validity of the incoming XML documents, you can rely on the database to ensure that the `XMLType` table or column only contains schema-valid XML documents.

In Example 3–15 the XML document, `InvalidReference`, is a not a valid XML document according to the XML schema. The XML schema defines a minimum length of 18 characters for the text node associated with the `Reference` element. In this document the node contains the value `SBELL-20021009`, which is only 14 characters long. Partial validation would not catch this error. Unless the constraint or trigger are present, attempts to insert this document into the database would succeed.

Example 3–15 shows how to force a full XML schema validation by adding a CHECK constraint to an XMLType table.

Example 3–15 Using CHECK Constraint to Force Full XML Schema Validation

Here, a CHECK constraint is added to PurchaseOrder table. Any attempt to insert an invalid document, namely one that does not pass the CHECK constraint, into the table fails:

```
ALTER TABLE PURCHASEORDER
ADD constraint VALIDATE_PURCHASEORDER
CHECK (XMLIsValid(object_value)=1);

Table altered.

--
INSERT INTO PURCHASEORDER
       VALUES
       (
         XMLType
          (
            bfilename('XMLDIR','InvalidReference.xml'),
            nls_charset_id('AL32UTF8')
          )
       );
INSERT INTO PURCHASEORDER
*

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

Note that the pseudo column name object_value can be used to access the content of an XMLType table from within a trigger.

Example 3–16 shows how to use a BEFORE INSERT trigger to validate that the data being inserted into the XMLType table conforms to the specified XML schema.

Example 3–16 Using BEFORE INSERT Trigger to Enforce Full XML Schema Validation

```
CREATE OR REPLACE TRIGGER VALIDATE_PURCHASEORDER
   BEFORE insert on PURCHASEORDER
   FOR each row
begin
  if (:new.object_value is not null) then
    :new.object_value.schemavalidate();
  end if;
end;
/


Trigger created.

INSERT INTO PURCHASEORDER
   VALUES (xmltype(getFileContent('InvalidReference.xml')));
   VALUES (xmltype(getFileContent('InvalidReference.xml')))
           *
ERROR at line 2:
ORA-31154: invalid XML document
ORA-19202: Error occurred in XML processing
LSX-00221: "SBELL-20021009" is too short (minimum length is 18)
ORA-06512: at "SYS.XMLTYPE", line 333
```

```
ORA-06512: at "SCOTT.VALIDATE_PURCHASEORDER", line 3
ORA-04088: error during execution of trigger 'SCOTT.VALIDATE_PURCHASEORDER'
```

Using SQL Constraints to Enforce Referential Integrity

The W3C XML Schema Recommendation defines a powerful language for defining the contents of an XML document. However there are a number of simple data management concepts not currently addressed by the W3C XML Schema Recommendation. These include the following:

- The ability to define that the value of an element or attribute has to be unique across a set of XML documents (a `UNIQUE` constraint)
- That the value of an element or attribute must exist in some data source outside the current document (a `FOREIGN KEY` constraint)

The mechanisms used to enforce integrity on XML are the same mechanisms used to enforce integrity on conventional relational data. In other words, simple rules such as uniqueness and foreign-key relationships, are enforced by specifying constraints. More complex rules are enforced by specifying database triggers. Example 3–17 and Example 3–18 illustrate how you can use SQL constraints to enforce referential integrity.

Oracle XML DB makes it possible to implement database-enforced business rules on XML content, in addition to rules that can be specified using the XML schema constructs. The database enforces these business rules regardless of whether XML is inserted directly into a table, or uploaded using one of the protocols supported by Oracle XML DB repository.

Example 3–17 Applying Database Integrity Constraints and Triggers to an XMLType Table

```
CREATE OR REPLACE TRIGGER VALIDATE_PURCHASEORDER
BEFORE insert on PURCHASEORDER
FOR each row
begin
  if (:new.object_value is not null) then
      :new.object_value.schemavalidate();
  end if;
end;
/


Trigger created.

--
INSERT INTO PURCHASEORDER
       VALUES
       (
         xmltype
         (
           bfilename('XMLDIR','InvalidReference.xml'),
           nls_charset_id('AL32UTF8')
     )
    );
INSERT INTO PURCHASEORDER
            *
ERROR at line 1:
ORA-31154: invalid XML document
ORA-19202: Error occurred in XML processing
```

```
LSX-00221: "SBELL-20021009" is too short (minimum length is 18)
ORA-06512: at "SYS.XMLTYPE", line 333
ORA-06512: at "SCOTT.VALIDATE_PURCHASEORDER", line 3
ORA-04088: error during execution of trigger 'SCOTT.VALIDATE_PURCHASEORDER'


--
ALTER TABLE PURCHASEORDER
ADD constraint REFERENCE_IS_UNIQUE
UNIQUE (xmldata."REFERENCE");

Table altered.

--
ALTER TABLE PURCHASEORDER
ADD constraint USER_IS_VALID
foreign key (xmldata."USERID") references HR.EMPLOYEES(EMAIL);

Table altered.

--
INSERT INTO PURCHASEORDER
       VALUES
       (
         xmltype
          (
            bfilename('XMLDIR','purchaseOrder.xml'),
            nls_charset_id('AL32UTF8')
          )
       );

1 row created.

--
INSERT INTO PURCHASEORDER
       VALUES
       (
         xmltype
          (
             bfilename('XMLDIR','DuplicateReference.xml'),
             nls_charset_id('AL32UTF8')
          )
       );
INSERT INTO PURCHASEORDER
*

ERROR at line 1:
ORA-00001: unique constraint (SCOTT.REFERENCE_IS_UNIQUE) violated


--
INSERT INTO PURCHASEORDER
       VALUES
       (
         xmltype
          (
            bfilename('XMLDIR','InvalidUser.xml'),
            nls_charset_id('AL32UTF8')
          )
       );
```

```
INSERT INTO PURCHASEORDER
*

ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.USER_IS_VALID) violated - parent key not
 found
```

The unique constraint `REFERENCE_IS_UNIQUE` enforces the rule that the value of the node `/PurchaseOrder/Reference/text()` is unique across all documents stored in the `PURCHASEORDER` table. The foreign key constraint `USER_IS_VALID` enforces the rule that the value of the node `/PurchaseOrder/User/text()` corresponds to one of the values in the EMAIL column in the `EMPLOYEES` table.

Oracle XML DB constraints must be specified in terms of attributes of the SQL types used to manage the XML content.

The following examples show how database-enforced data integrity ensures that only XML documents that do not violate the database-enforced referential constraints can be stored in the `PURCHASEORDER` table.

The text node associated with the `Reference` element in the XML document `DuplicateRefernce.xml`, contains the same value as the corresponding node in XML document `PurchaseOrder.xml`. This means that attempting to store both documents in Oracle XML DB results in the constraint `REFERENCE_IS_UNIQUE` being violated.

The text node associated with the `User` element in XML document `InvalidUser.xml`, contains the value `HACKER`. There is no entry in the `EMPLOYEES` table where the value of the `EMAIL` column is `HACKER`. This means attempting to store this document in Oracle XML DB results in the constraint `USER_IS_VALID` being violated.

```
INSERT INTO PURCHASEORDER
   VALUES (xmltype(getFileContent('PurchaseOrder.xml')));

1 row created.

INSERT INTO PURCHASEORDER
   VALUES (xmltype(getFileContent('DuplicateReference.xml')));

insert into PURCHASEORDER
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.REFERENCE_IS_UNIQUE) violated


INSERT INTO PURCHASEORDER
   VALUES (xmltype(getFileContent('InvalidUser.xml')));

insert into PURCHASEORDER
*
ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.USER_IS_VALID) violated - parent key not
found
```

Integrity rules defined using constraints and triggers are also enforced when XML schema-based XML content is loaded into Oracle XML DB repository.

Example 3–18 demonstrates that database integrity is also enforced when a protocol, such as FTP is used to upload XML schema-based XML content into Oracle XML DB repository.

Example 3–18 Enforcing Database Integrity When Loading XML Using FTP

```
$ ftp localhost 2100
Connected to localhost.
220 mdrake-sun FTP Server (Oracle XML DB/Oracle Database 10g Enterprise Edition
Release 10.1.0.0.0 - Beta) ready.
Name (localhost:oracle10): SCOTT
331 pass required for SCOTT
Password:
230 SCOTT logged in
ftp> cd /home/SCOTT
250 CWD Command successful
ftp> put InvalidReference.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-31154: invalid XML document
ORA-19202: Error occurred in XML processing
LSX-00221: "SBELL-20021009" is too short (minimum length is 18)
ORA-06512: at "SYS.XMLTYPE", line 333
ORA-06512: at "SCOTT.VALIDATE_PURCHASEORDER", line 3
ORA-04088: error during execution of trigger 'SCOTT.VALIDATE_PURCHASEORDER'
550 End Error Response
ftp> put InvalidElement.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-30937: No schema definition for 'UserName' (namespace '##local') in parent
'PurchaseOrder'
550 End Error Response
ftp> put DuplicateReference.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-00001: unique constraint (SCOTT.REFERENCE_IS_UNIQUE) violated
550 End Error Response
ftp> put InvalidUser.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-02291: integrity constraint (SCOTT.USER_IS_VALID) violated - parent key not
 found
550 End Error Response
```

Full SQL Error Trace

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

See also: *Oracle Database Error Messages*

DML Operations on XML Content Using Oracle XML DB

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

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

XPath and Oracle XML

Oracle XML DB includes new `XMLType` methods and XML-specific SQL functions. WIth these you can query and update XML content stored in Oracle Database. They use the W3C XPath Recommendation to identify the required node or nodes. Every node in an XML document can be uniquely identified by an XPath expression. An XPath expression consists of a slash-separated list of element names, attributes names, and XPath functions. XPath expressions may contain indexes and conditions that determine which branch of the tree is traversed in determining the target nodes.

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

Querying XML Content Stored in Oracle XML DB

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

A 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://localhost:8080/home/SCOTT
/poSource/xsd/purchaseOrder.xsd">
  <Reference>SBELL-20021009123336O1PDT</Reference>
  <Actions>
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
```

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


1 row selected.
```

Retrieving the Content of an XML Document Using Object_Value

The `object_value` keyword can be used as an alias for the value of an object table. For an `XMLType` table that consists of a single column of `XMLType`, the entire XML document is retrieved. `object_value` replaces the `value(x)` and `sys_nc_rowinfo$` aliases used in prior releases.

The SQL*Plus settings `PAGESIZE` and `LONG` ensure that the entire document is printed correctly without line breaks.

Example 3–19 Using object_value to Retrieve an Entire XML Document

```
set long 10000
set pagesize 100
set linesize 132
--
SELECT object_value
FROM PURCHASEORDER;

OBJECT_VALUE
--------------------------------------------------------------------------------
--------------------------------------------------
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://localhost:8080/home/SCOTT
/poSource/xsd/purchaseOrder.xsd">
  <Reference>SBELL-20021009123336O1PDT</Reference>
  <Actions>
```

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


1 row selected.
```

Accessing Fragments or Nodes of an XML Document Using extract()

The `extract()` function returns the node or nodes that match the XPath expression. Nodes are returned as an instance of `XMLType`. The results of `extract()` can be either a `document` or `DocumentFragment`. The functionality of `extract()` is also available through the `XMLType` datatype, `extract()` method.

Example 3–20 Accessing XML Fragments Using extract()

The following SQL statement returns an `XMLType` containing the `Reference` element that matches the XPath expression.

```
set pages 100
set linesize 132
set long 10000
--
SELECT extract(object_value,'/PurchaseOrder/Reference')
FROM PURCHASEORDER;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/REFERENCE')
--------------------------------------------------------------------------------
<Reference>SBELL-20020100912333601PDT</Reference>
```

```
1 row selected.
```

The following statement returns an `XMLType` containing the first `LineItem` element in the `LineItems` collection:

```
SELECT extract(object_value,'/PurchaseOrder/LineItems/LineItem[1]')
FROM PURCHASEORDER;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[1]')
--------------------------------------------------------------------------------
<LineItem ItemNumber="1">
  <Description>A Night to Remember</Description>
  <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
</LineItem>

1 row selected.
```

The following SQL statement returns an `XMLType` containing the three `Description` elements that match the XPath expression. The three `Description` elements are returned as nodes in a single `XMLType`. This means that the `XMLType` does not have a single root node. Consequently it is treated as an XML `DocumentFragment`.

```
SELECT extract(object_value,'/PurchaseOrder/LineItems/LineItem/Description')
FROM PURCHASEORDER;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM/DESCRIPTION')
--------------------------------------------------------------------------------
<Description>A Night to Remember</Description>
<Description>The Unbearable Lightness Of Being</Description>
<Description>Sisters</Description>


1 row selected.
```

Accessing Text Nodes and Attribute Values Using extractValue()

The `extractValue()` function returns the value of the text node or attribute value that matches the supplied XPath expression. The value is returned as a SQL scalar datatype. This means that the XPath expression passed to `extractValue()` must uniquely identify a single text node or attribute value within the document.

Example 3–21 Accessing a Text Node Value Matching an XPath Expression Using extractValue()

The following SQL statement returns the value of the text node associated with the `Reference` element that matches the XPath expression. The value is returned as a `VARCHAR2` datatype.

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference')
FROM PURCHASEORDER;

EXTRACTVALUE(OBJECT_VALUE,'/PU
------------------------------
SBELL-20021009123336O1PDT

1 row selected.
```

The following SQL statement returns the value of the text node associated with the `Description` element associated with the first `LineItem` element. The value is

returned as VARCHAR2 datatype. Note the use of the Index to identify which of the `LineItem` nodes should be processed.

```
SELECT extractValue(object_value,
'/PurchaseOrder/LineItems/LineItem[1]/Description')
FROM PURCHASEORDER;

EXTRACTVALUE(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[1]/DESCRIPTION')
--------------------------------------------------------------------------------
A Night to Remember

1 row selected.
```

The following SQL statement returns the value of the text node associated with the `Description` element, in turn associated with the `LineItem` element. The `LineItem` element contains an `Id` attribute with the specified value. The value is returned as VARCHAR2 datatype. Note how the predicate that identifies which `LineItem` to process is enclosed in Square Brackets (`[]`). The at-sign character (@) specifies that `Id` is an attribute rather than an element.

```
SELECT extractValue(object_value,
'/PurchaseOrder/LineItems/LineItem[Part/@Id="715515011020"]/Description')
FROM PURCHASEORDER;

EXTRACTVALUE(OBJECT
_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[PART/@ID="715515011020"]/DESCRIPTION')
--------------------------------------------------------------------------------
Sisters

1 row selected.
```

Invalid Use of extractValue()

The following examples show invalid uses of `extractValue()`. In the first example the XPath expression matches three nodes in the document. In the second example the XPath expression identifies a node tree, not a text node or attribute value.

Example 3–22 Invalid Uses of extractValue()

```
SELECT extractValue(object
_value,'/PurchaseOrder/LineItems/LineItem/Description')
FROM PURCHASEORDER;
SELECT extractValue(object
_value,'/PurchaseOrder/LineItems/LineItem/Description')
 *
ERROR at line 1:
ORA-01427: single-row subquery returns more than one row


--
SELECT extractValue(object_value,'/PurchaseOrder/LineItems/LineItem[1]')
FROM PURCHASEORDER;
FROM PURCHASEORDER
     *
ERROR at line 2:
ORA-19026: EXTRACTVALUE can only retrieve value of leaf node


--
SELECT extractValue(object
_value,'/PurchaseOrder/LineItems/LineItem/Description/text()')
```

```
FROM PURCHASEORDER;
SELECT extractValue(object
_value,'/PurchaseOrder/LineItems/LineItem/Description/text()')
 *
```

Note that depending on whether or not XPath rewrite takes place, the two preceding statements can also result in the following error being reported:

```
ORA-01427: single-row subquery returns more than one row
```

Searching the Content of an XML Document Using existsNode()

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 and false (0) if it does not. Since XPath expressions can contain predicates `existsNode()` can determine whether or not a given node exists in the document, or whether or not a node with the specified value exists in the document. The functionality provided by the `existsNode()` function is also available through the `XMLType` datatype `existsNode()` method.

Example 3–23 Searching XML Content Using the existsNode() Function

This example checks if the XML document contains a root element named `Reference` that is a child of the root element `PurchaseOrder`:

```
SELECT COUNT(*)
FROM PURCHASEORDER
WHERE existsNode(object_value,'/PurchaseOrder/Reference') = 1;
  COUNT(*)
----------
       132
```

The following example checks if the value of the text node associated with the `Reference` element is `SBELL-2002100912333601PDT`:

```
SELECT count(*)
FROM PURCHASEORDER
WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;
  COUNT(*)
----------
         1
1 row selected.
```

The following example checks if the value of the text node associated with the `Reference` element is `SBELL-XXXXXXXXXXXXXXXXXX`:

```
SELECT count(*)
FROM PURCHASEORDER
WHERE existsNode(object_
value,'/PurchaseOrder/Reference[Reference="SBELL-XXXXXXXXXXXXXXXXXX"]') = 1;

  COUNT(*)
----------
         0

1 row selected.
```

The following example checks if the XML document contains a root element `PurchaseOrder` that contains a `LineItems` element containing a `LineItem` element, which in turn contains a `Part` element with an `Id` attribute:

```
SELECT count(*)
FROM PURCHASEORDER
WHERE existsNode(object_value,'/PurchaseOrder/LineItems/LineItem/Part/@Id') = 1;

  COUNT(*)
----------
       132

1 row selected.
```

The following checks if the XML document contains a root element `PurchaseOrder` that contains a `LineItems` element, contain a `LineItem` element which contains a `Part` element where the value of the `Id` attribute is `715515009058`:

```
SELECT count(*)
FROM PURCHASEORDER
WHERE existsNode(object_
value,'/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]') = 1;

  COUNT(*)
----------
        21
```

The following checks if the XML document contains a root element `PurchaseOrder` that contains `LineItems` element, where the third `LineItem` element contains a `Part` element where the value of the `Id` attribute is `715515009058`:

```
SELECT count(*)
FROM PURCHASEORDER
WHERE existsNode(object_
value,'/PurchaseOrder/LineItems/LineItem[3]/Part[@Id="715515009058"]') = 1;

  COUNT(*)
----------
         1
1 row selected.
```

The following query shows how to use `extractValue()` to limit the results of the SELECT statement to those rows where the text node associated with the `User` element starts with the letter `S`. XPath 1.0 does not include support for `LIKE`-based queries:

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference') "Reference"
FROM PURCHASEORDER
WHERE extractValue(object_value,'/PurchaseOrder/User') LIKE 'S%';

Reference
------------------------------
SBELL-20021009123336231PDT
SBELL-20021009123336331PDT
SKING-20021009123336321PDT
...
36 rows selected.
```

The following query shows how to use `extractValue()` to perform a join based on the values of a node in an XML document and data in another table.

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference') "Reference"
FROM PURCHASEORDER, HR.EMPLOYEES e
WHERE extractValue(object_value,'/PurchaseOrder/User') = e.EMAIL
AND e.EMPLOYEE_ID = 100;

Reference
------------------------------
SKING-20021009123336321PDT
SKING-20021009123337153PDT
SKING-20021009123335560PDT
SKING-20021009123336952PDT
SKING-20021009123336622PDT
SKING-20021009123336822PDT
SKING-20021009123336131PDT
SKING-20021009123336392PDT
SKING-20021009123337974PDT
SKING-20021009123338294PDT
SKING-20021009123337703PDT
SKING-20021009123337383PDT
SKING-20021009123337503PDT

13 rows selected.
```

Using extractValue() and existsNode() in the WHERE Clause

The preceding examples demonstrated how `extractValue()` can be used in the `SELECT` list to return information contained in an XML document. You can also use these functions in the `WHERE` clause to determine whether or not a document must be included in the resultset of a `SELECT`, `UPDATE`, or `DELETE` statement.

You can use `existsNode()` to restrict the resultset to those documents containing nodes that match an XPath expression. You can use `extractValue()` when joining across multiple tables based on the value of one or more nodes in the XML document. Also use `existsNode()` when specifying the condition in SQL is easier than specifying it with XPath.

Example 3–24 Limiting the Results of a SELECT Using existsNode() and extractValue() in the WHERE Clause

The following query shows how to use `existsNode()` to limit the results of the `SELECT` statement to rows where the text node associated with the `User` element contains the value `SBELL`.

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference') "Reference"
FROM PURCHASEORDER
WHERE existsNode(object_value,'/PurchaseOrder[User="SBELL"]') = 1;

Reference
------------------------------
SBELL-20021009123336231PDT
SBELL-20021009123336331PDT
SBELL-20021009123337353PDT
SBELL-20021009123338304PDT
SBELL-20021009123338505PDT
SBELL-20021009123335771PDT
SBELL-20021009123335280PDT
SBELL-20021009123333763PDT
SBELL-20021009123333601PDT
SBELL-20021009123336362PDT
SBELL-20021009123336532PDT
```

```
SBELL-20021009123338204PDT
SBELL-20021009123337673PDT

13 rows selected.
```

Example 3–25 Finding the Reference for any PurchaseOrder Using extractValue() and existsNode()

This example combines `extractValue()` and `existsNode()` to find the `Reference` for any PurchaseOrder where the first `LineItem` element contains an order for the item with the `Id` `715515009058`. In this example the `existsNode()` function is used in the `WHERE` clause to determine which rows are selected, and the `extractValue()` function is used in the `SELECT` list to control which part of the selected documents appear in the result.

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference') "Reference"
FROM PURCHASEORDER
WHERE existsNode(object_
value,'/PurchaseOrder/LineItems/LineItem[1]/Part[@Id="715515009058"]') = 1;

Reference
------------------------------
SBELL-20021009123333601PDT

1 row selected.
```

Using XMLSequence() to Perform SQL Operations on XMLType Fragments

Example 3–20 demonstrated how the `extract()` function returns an `XMLType` containing the node or nodes that matched the supplied XPath expression. When the document contains multiple nodes that match the supplied XPath expression, `extract()` returns a document fragment containing all of the matching nodes. A fragment differs from a document in that it may contain multiple root elements which may be unrelated.

This kind of result is very common when the `extract()` function is used to retrieve the set of elements contained in a collection (in this case each node in the fragment will be of the same type), or when the XPath expression terminates in a wildcard (where the nodes in the fragment will be of different types).

The `XMLSequence()` function makes it possible to take an `XMLType` containing a fragment and perform SQL operations on it. It generates a collection of `XMLType` objects from an `XMLType` containing a fragment. The collection contains one `XMLType` for each of the root elements in the fragment. This collection of `XMLType` objects can then be converted into a virtual table using the SQL `table()` function. Converting the fragment into a virtual table makes it easier to use SQL to process the results of an `extract()` function that returned multiple nodes.

Example 3–26 Using XMLSequence() and Table() to view Description Nodes

The following example demonstrates how to access the text nodes for each `Description` element in the `PurchaseOrder` document.

The initial approach, based on using `extractValue()`, fails as there is more then one `Description` element in the document.

```
SELECT extractValue(p.object_
value,'/PurchaseOrder/LineItems/LineItem/Description')
FROM purchaseorder p
WHERE
```

```
existsNode(p.object_value,'/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]')
= 1;
SELECT extractValue(p.object_
value,'/PurchaseOrder/LineItems/LineItem/Description')
 *
ERROR at line 1:
ORA-01427: single-row subquery returns more than one row
```

Next use `extract()` to access the required values. This returns the set of `Description` nodes as a single `XMLType` object containing a fragment consisting of the three `Description` nodes. This is better but not ideal because the objective is to perform further SQL-based processing on the values in the text nodes.

```
SELECT extract(p.object_value, '/PurchaseOrder/LineItems/LineItem/Description')
FROM purchaseorder p
WHERE
existsNode(p.object
_value,'/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;

EXTRACT(P.OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM/DESCRIPTION')
--------------------------------------------------------------------------------
<Description>A Night to Remember</Description>
<Description>The Unbearable Lightness Of Being</Description>
<Description>Sisters</Description>

1 row selected.
```

To use SQL to process the contents of the text nodes you must convert the collection of `Description` nodes into a virtual table using the `XMLSequence()` and `table()` functions. These functions convert the three `Description` nodes retuned by `extract()` into a virtual table consisting of three `XMLType` objects, each of which contains a single `Description` element.

```
SELECT value(d)
FROM purchaseorder p,
table (xmlsequence(extract(p.object
_value,'/PurchaseOrder/LineItems/LineItem/Description'))) d
WHERE existsNode(p.object
_value,'/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;

VALUE(D)
--------------------------------------------------------------------------------
<Description>A Night to Remember</Description>
<Description>The Unbearable Lightness Of Being</Description>
<Description>Sisters</Description>

3 rows selected.
```

Since each `XMLType` in the virtual table contains a single `Description` element, `extractValue()` function can be used to access the value of the text node associated with the each `Description` element.

```
SELECT extractValue(value(d),'/Description')
FROM purchaseorder p,
table (xmlsequence(extract(p.object
_value,'/PurchaseOrder/LineItems/LineItem/Description'))) d
WHERE existsNode(p.object
_value,'/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;

EXTRACTVALUE(VALUE(D),'/DESCRIPTION')
--------------------------------------------------------------------------------
A Night to Remember
```

```
The Unbearable Lightness Of Being
Sisters

3 rows selected.
```

> **Note:** There is a correlated join between the results of the `table()` function and the row operated on by the `extract()` function. This means that the table that provides input to the `extract()` function must appear *before* the `table()` operator in the `FROM` list. The correlated join ensures a 1 : N relationship between the rows generated by the `table()` function and the row containing the value that was processed by the `extract()` function.

Example 3–27 Counting the Number of Elements in a Collection Using XMLSequence()

The following example demonstrates using `XMLSequence()` to count the number of elements in a collection. It also shows how SQL functionality such as `ORDER BY` and `GROUP BY` can be applied to results of the `extractValue()` operator.

In this case the query will first locate the set of the XML documents that match the XPath expression contained in the `existsNode()` function. It will then generate a virtual table containing the set of `LineItem` nodes for each document selected. Finally it counts the number of `LineItem` nodes for each `PurchaseOrder` document. The correlated join ensures that the `GROUP BY` correctly determines which `LineItems` belong to which `PurchaseOrder`.

```
SELECT extractValue(p.object_value,'/PurchaseOrder/Reference'), count(*)
FROM PURCHASEORDER p,
table (xmlsequence(extract(p.object_value,
'/PurchaseOrder/LineItems/LineItem'))) d
WHERE existsNode(p.object_value,'/PurchaseOrder[User="SBELL"]') = 1
GROUP BY extractValue(p.object_value,'/PurchaseOrder/Reference')
ORDER BY extractValue(p.object_value,'/PurchaseOrder/Reference');

EXTRACTVALUE(P.OBJECT_VALUE,'/   COUNT(*)
------------------------------ ----------
SBELL-20021009123335280PDT             20
SBELL-20021009123335771PDT             21
SBELL-20021009123336601PDT              3
SBELL-20021009123336231PDT             25
SBELL-20021009123336331PDT             10
SBELL-20021009123336362PDT             15
SBELL-20021009123336532PDT             14
SBELL-20021009123337353PDT             10
SBELL-20021009123337763PDT             21
SBELL-20021009123337673PDT             10
SBELL-20021009123338204PDT             14
SBELL-20021009123338304PDT             24
SBELL-20021009123338505PDT             20

13 rows selected.
```

Example 3–28 Counting the Number of Child Elements in an Element Using XMLSequence()

The following example demonstrates using `XMLSequence()` to count the number of child elements of a given element. The XPath expression passed to the `extract()` function contains a wildcard that matches the elements that are direct descendants of

the PurchaseOrder element. The XMLType returned by extract() will contain the set of nodes which match the XPath expression. The XMLSequence() function transforms each root element in the fragment into a separate XMLType object, and the table() function converts the collection returned by XMLSequence() into a virtual table. Counting the number of rows in the virtual table provides the number of child elements in the PurchaseOrder element.

```
SELECT count(*)
FROM PURCHASEORDER p,
TABLE (xmlSequence(extract(p.object_value,'/PurchaseOrder/*'))) n
WHERE existsNode(p.object_value,
'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

  COUNT(*)
----------
         9

1 row selected.
```

Accessing and Updating XML Content in Oracle XML DB Repository

These sections describe features for accessing and updating Oracle XML DB repository content.

Access XML Documents Using SQL

Another benefit of XML DB repository is that it can be queried from SQL. Content stored in Oracle XML DB repository can be accessed and updated from SQL and PL/SQL. You can interrogate the structure of the repository in complex ways.

For example, you can issue a query to determine how many documents with an .xsl extension are under a location other than /home/mystylesheetdir.

For document access, you can also mix path-based repository access with content-based access. For example, "how many documents not under /home/purchaseOrders have a node named /PurchaseOrder/User/text() with a value of DRAKE?

All the metadata for managing Oracle XML DB repository is stored in a database schema owned by the database user XDB. This user is created during Oracle XML DB installation. The primary table in this schema is an XMLType table called XDB$RESOURCE. This contains one row for each file or folder in Oracle XML DB repository. Documents in this table are referred to as *resource* documents. The XML schema that defines the structure of an Oracle XML DB *resource* document is registered under URL, "http://xmlns.oralce.com/xdb/XDBResource.xsd.

Repository Content is Exposed Through RESOURCE_VIEW and PATH_VIEW

XDB$RESOURCE table is not directly exposed to SQL programmers. Instead the contents of the repository are exposed through two public views, RESOURCE_VIEW and PATH_VIEW. Through these views you can access and update metadata and content of documents stored in Oracle XML DB repository.

Both views contain a virtual column, RES. Use RES to access and update resource documents with SQL statements based on a path notation. Operations on the views use underlying tables in Oracle XML DB repository.

Use exists_Path() and under_Path() Operators to Include Path-Based Predicates in the WHERE Clause

Oracle XML DB includes two repository specific SQL operators: `exists_path()` and `under_path()`. Use these operators to include path-based predicates in the `WHERE` clause of a SQL statement. This means that SQL operations can select repository content based on the location of the content in the folder hierarchy. The Hierarchical Index ensures that path-based queries are executed efficiently.

When **XML schema-based** XML documents are stored in Oracle XML DB repository the document content is stored as an object in the default table identified by the XML schema. The repository contains metadata about the document and a pointer (`REF` of `XMLType`) identifies the row in the default table that contains the content.

You Can Also Store Non-XML Documents in the Repository

It is also possible to store other kinds of documents in the repository. When non-XML and non-XML schema-based XML documents are stored in the repository, the documents' content is stored in a LOB along with the metadata about the document.

PL/SQL Packages Allow Creating, Deleting, Renaming, Moving, ... Folders and Documents

Since Oracle XML DB repository can be accessed and updated using SQL, any application capable of calling a PL/SQL procedure can work with Oracle XML DB repository. All SQL and PL/SQL repository operations are transactional, and access to the repository and its contents is subject to database security as well as XML DB repository Access Control Lists (ACLs).

With supplied PL/SQL packages `DBMS_XDB`, `DBMS_XDBZ`, and `DBMS_XDB_VERSION`, SQL programmers can perform common tasks on the repository itself. Methods provided by the packages make it possible to create, delete, and rename documents and folders, to move a file or folder within the folder hierarchy, to set and change the access permissions on a file or folder, and the ability to initiate and manage versioning.

The following example shows PL/SQL package `DBMS_XDB` used to create a set of subfolders beneath folder `/home/SCOTT`.

```
connect &1/&2@&3

DECLARE
  RESULT boolean;
BEGIN
  if (not xdb_utilities.ResourceExists('/home/' || USER || '/poSource')) then
    result := dbms_xdb.createFolder('/home/' || USER || '/poSource');
  end if;
  if (not xdb_utilities.ResourceExists('/home/' || USER || '/poSource/xsd')) then
    result := dbms_xdb.createFolder('/home/' || USER || '/poSource/xsd');
  end if;
  if (not xdb_utilities.ResourceExists('/home/' || USER || '/poSource/xsl')) then
    result := dbms_xdb.createFolder('/home/' || USER || '/poSource/xsl');
  end if;
  result := dbms_xdb.createFolder('/home/' || USER || '/purchaseOrders');
END;
/

--
-- Refresh the contents of WebDAV folder to show that new directories have been created.
--
PAUSE
--
-- The new directories were not visible from WebDAV as the transaction had not been committed.
-- Issue a COMMIT statement and then refresh the contents of the WebDAV folder.
-- The new directories should now be visible as the transaction that created them have been
```

```
-- committed.
--
COMMIT
/
```

Relational Access to XML Content Stored in Oracle XML DB Using Views

The XML-specific functions and methods provided by Oracle XML DB can be used to create conventional *relational views* that provide relational access to XML content. This allows programmers, tools, and applications that understand Oracle Database, but not XML, to work with XML content stored in the database.

The views use XPath expressions and functions such as `extractValue()` to define the mapping between columns in the view and nodes in the XML document. For performance reasons this approach is recommended when XML documents are stored as `XMLType`, that is, stored using object-relational storage techniques.

Example 3–29 Creating Relational Views On XML Content

This example shows how to create a simple relational view that exposes XML content in a relational manner:

```
CREATE OR REPLACE view PURCHASEORDER_MASTER_VIEW
(REFERENCE, REQUESTOR, USERID, COSTCENTER,
SHIP_TO_NAME,SHIP_TO_ADDRESS, SHIP_TO_PHONE,
INSTRUCTIONS)
AS
SELECT extractValue(value(p),'/PurchaseOrder/Reference'),
extractValue(value(p),'/PurchaseOrder/Requestor'),
extractValue(value(p),'/PurchaseOrder/User'),
extractValue(value(p),'/PurchaseOrder/CostCenter'),
extractValue(value(p),'/PurchaseOrder/ShippingInstructions/name'),
extractValue(value(p),'/PurchaseOrder/ShippingInstructions/address'),
extractValue(value(p),'/PurchaseOrder/ShippingInstructions/telephone'),
extractValue(value(p),'/PurchaseOrder/SpecialInstructions')
FROM PURCHASEORDER p;

View created.

--
describe PURCHASEORDER_MASTER_VIEW
 Name                                      Null?    Type
 ----------------------------------------------------------------------------
 REFERENCE                                          VARCHAR2(30 CHAR)
 REQUESTOR                                          VARCHAR2(128 CHAR)
 USERID                                             VARCHAR2(10 CHAR)
 COSTCENTER                                         VARCHAR2(4 CHAR)
 SHIP_TO_NAME                                       VARCHAR2(20 CHAR)
 SHIP_TO_ADDRESS                                    VARCHAR2(256 CHAR)
 SHIP_TO_PHONE                                      VARCHAR2(24 CHAR)
 INSTRUCTIONS                                       VARCHAR2(2048 CHAR)
```

This example created view `PURCHASEORDER_MASTER_VIEW`. There will be one row in the view for each row in table `PURCHASEORDER`.

The `CREATE VIEW` statement defines the set of columns that will make up the view. The `SELECT` statement uses XPath expressions and the `extractValue()` function to map between the nodes in the XML document and the columns defined by the view.

This technique can be used when there is a 1:1 relationship between documents in the XMLType table and the rows in the view.

Example 3–30 Using a View to Access Individual Members of a Collection

This example shows how to use extract() and xmlSequence() for a 1:many relationship between the documents in the XMLType table and rows in the view. This situation arises when the view must provide access to the individual members of a collection and expose the members of a collection as a set rows.

```
CREATE OR REPLACE VIEW PURCHASEORDER_DETAIL_VIEW
(REFERENCE, ITEMNO, DESCRIPTION,
PARTNO, QUANTITY, UNITPRICE)
AS
SELECT extractValue(value(p),'/PurchaseOrder/Reference'),
extractvalue(value(l),'/LineItem/@ItemNumber'),
extractvalue(value(l),'/LineItem/Description'),
extractvalue(value(l),'/LineItem/Part/@Id'),
extractvalue(value(l),'/LineItem/Part/@Quantity'),
extractvalue(value(l),'/LineItem/Part/@UnitPrice')
FROM PURCHASEORDER p,
TABLE (xmlsequence(extract(value(p),'/PurchaseOrder/LineItems/LineItem'))) l;

View created.

--
describe PURCHASEORDER_DETAIL_VIEW
 Name                                                  Null?    Type
 ----------------------------------------------------------------
 REFERENCE                                                      VARCHAR2(30 CHAR)
 ITEMNO                                                         NUMBER(38)
 DESCRIPTION                                                    VARCHAR2(1024)
 PARTNO                                                         VARCHAR2(56)
 QUANTITY                                                       NUMBER(12,2)
 UNITPRICE                                                      NUMBER(8,4)
```

This example creates a view called PURCHASEORDER_DETAIL_VIEW. There will be one row in the view for each LineItem element the occurs in the XML documents stored in table PURCHASEORDER.

The CREATE VIEW statement defines the set of columns that will make up the view. The SELECT statement uses extract() to access the set of LineItem elements in each PurchaseOrder document. It then uses xmlSequence() and TABLE() to create a virtual table that contains one XML document for each LineItem in the PURCHASEORDER table.

The XPath expressions passed to the extractValue() function are used to map between the nodes in the LineItem documents and the columns defined by the view. The Reference element included in the view to create a Foreign Key that can used to joins rows in PURCHASEORDER_DETAIL_VIEW to the corresponding row in PURCHASEORDER_MASTER_VIEW. The correlated join in the CREATE VIEW statement ensures that the 1:many relationship between the Reference element and the associated LineItem elements is maintained when the view is accessed.

As can be seen from the output of the DESCRIBE statement, both views appear to be a standard relational views. Since the XMLType table referenced in the CREATE VIEW statements is based on an XML schema, Oracle XML DB can determine the datatypes of the columns in the views from the information contained in the XML schema.

The following examples show some of the benefits provided by creating relational views over XMLType tables and columns.

Example 3–31 SQL queries on XML Content Using Views

This example uses a simple query against the master view. The query uses a conventional SQL SELECT statement to select rows where the USERID column starts with S.

```
column REFERENCE format A30
column DESCRIPTION format A40
--
SELECT REFERENCE, COSTCENTER, SHIP_TO_NAME
FROM PURCHASEORDER_MASTER_VIEW
WHERE USERID like 'S%';

REFERENCE                      COST SHIP_TO_NAME
------------------------------ ---- --------------------
SBELL-20021009123336231PDT     S30  Sarah J. Bell
SBELL-20021009123336331PDT     S30  Sarah J. Bell
SKING-20021009123336321PDT     A10  Steven A. King
...
36 rows selected.
```

The next query is based on a join between the master view and detail view. Again, a conventional SQL SELECT statement finds the PURCHASEORDER_DETAIL_VIEW rows where the value of the ITEMNO column is 1 and the corresponding PURCHASEORDER_MASTER_VIEW row contains a USERID column with the value SMITH.

```
SELECT d.REFERENCE, d.ITEMNO, d.PARTNO, d.DESCRIPTION
FROM PURCHASEORDER_DETAIL_VIEW d, PURCHASEORDER_MASTER_VIEW m
WHERE m.REFERENCE = d.REFERENCE
AND m.USERID = 'SBELL'
AND d.ITEMNO = 1;

REFERENCE                       ITEMNO PARTNO         DESCRIPTION
------------------------------ ------------------------------------------------
SBELL-20021009123336231PDT              1 37429165829    Juliet of the Spirits
SBELL-20021009123336331PDT              1 715515009225   Salo
SBELL-20021009123337353PDT              1 37429141625    The Third Man
SBELL-20021009123338304PDT              1 715515009829   Nanook of the North
SBELL-20021009123338505PDT              1 37429122228    The 400 Blows
SBELL-20021009123335771PDT              1 37429139028    And the Ship Sails on
SBELL-20021009123335280PDT              1 715515011426   All That Heaven Allows
SBELL-20021009123333763PDT              1 715515010320   Life of Brian - Python
SBELL-20021009123333601PDT              1 715515009058   A Night to Remember
SBELL-20021009123336362PDT              1 715515012928   In the Mood for Love
SBELL-20021009123336532PDT              1 37429162422    Wild Strawberries
SBELL-20021009123338204PDT              1 37429168820    Red Beard
SBELL-20021009123337673PDT              1 37429156322    Cries and Whispers

13 rows selected.
```

Since the views look and act like standard relational views they can be queried using standard relational syntax. No XML-specific syntax is required in either the query syntax or the generated result set.

By exposing XML content as relational data Oracle XML DB allows advanced features of Oracle Database, such as business intelligence and analytic capabilities, to be applied to XML content. Even though the business intelligence features themselves are

not XML-aware, the XML-SQL duality provided by Oracle XML DB allows these features to be applied to XML content.

Example 3–32 Querying XML Using Views of XML Content

This example demonstrates using relational views over XML content to perform business intelligence queries on XML documents. The query performs an analysis of `PurchaseOrder` documents that contain orders for titles identified by UPC codes `715515009058` and `715515009126`.

```
SELECT partno, count(*) "No of Orders", quantity "No of Copies"
FROM purchaseorder_detail_view
WHERE partno IN ( 715515009126, 715515009058 )
GROUP BY rollup(partno, quantity);

PARTNO         No of Orders No of Copies
-------------- ------------ ------------
715515009058              7            1
715515009058              9            2
715515009058              5            3
715515009058              2            4
715515009058             23
715515009126              4            1
715515009126              7            3
715515009126             11
                          34
9 rows selected.
```

The query determines the number of copies of each title that are being ordered on each `PurchaseOrder`. Looking at the results for the part number `715515009126`, the query shows that there are seven `PurchaseOrder` values where one copy of the item is ordered and two `PurchaseOrder` values where four copies of the item are ordered.

> **See Also:** Chapter 4, "Using XMLType" for a description of `XMLType` datatype and functions and Appendix C, "XPath and Namespace Primer" for an introduction to the W3C XPath Recommendation

Updating XML Content Stored in Oracle XML DB

Oracle XML DB allows update operations to take place on XML content. Update operations can either replace the entire contents or parts of a document. The ability to perform partial updates on XML documents is very powerful, particularly when trying to make small changes to large documents, as it can significantly reduce the amount of network traffic and disk input-output required to perform the update.

The `updateXML()` function enables partial update of an XML document stored as an `XMLType`. It allows multiple changes to be made to the document in a single operation. Each change consists of an XPath expression which identifies the node to be updated, and the new value for the node.

Example 3–33 Updating XML Content Using updateXML()

The following example shows an `updateXML()` function used to update the value of the text node associated with the `User` element.

```
SELECT extractValue(object_value,'/PurchaseOrder/User')
FROM PURCHASEORDER
WHERE
```

```
existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]')
= 1;

EXTRACTVAL
----------
SBELL

1 row selected.

--
UPDATE PURCHASEORDER
SET object_value = updateXML(object_value,'/PurchaseOrder/User/text()','SKING')
WHERE existsNode(object
_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

1 row updated.

--
SELECT extractValue(object_value,'/PurchaseOrder/User')
FROM PURCHASEORDER
WHERE existsNode(object_
value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVAL
----------
SKING

1 row selected.
```

Example 3–34 Replacing an Entire Element Using updateXML()

This example uses `updateXML()` to replace an entire element within the XML document. Here the XPath expression references the element, and the replacement value is passed as an `XMLType` object.

```
SELECT extract(object_value,'/PurchaseOrder/LineItems/LineItem[1]')
 FROM PURCHASEORDER
 WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[1]')
--------------------------------------------------------------------------------
<LineItem ItemNumber="1">
  <Description>A Night to Remember</Description>
  <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
</LineItem>


1 row selected.

--
UPDATE PURCHASEORDER
   SET object_value = updateXML
      (
        object_value,
        '/PurchaseOrder/LineItems/LineItem[1]',
        xmltype
         (
            '<LineItem ItemNumber="1">
            <Description>The Lady Vanishes</Description>
            <Part Id="37429122129" UnitPrice="39.95" Quantity="1"/>
```

```
           </LineItem>'
         )
      )
WHERE existsNode(object_value,
 '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;

1 row updated.

--
SELECT extract(object_value,'/PurchaseOrder/LineItems/LineItem[1]')
 FROM PURCHASEORDER
 WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[1]')
--------------------------------------------------------------------------------
<LineItem ItemNumber="1">
  <Description>The Lady Vanishes</Description>
  <Part Id="37429122129" UnitPrice="39.95" Quantity="1"/>
</LineItem>


1 row selected.
```

Example 3–35 Updating a Node Occurring Multiple Times Within a Collection Using updateXML(): Incorrect Usage

This example show a common error that occurs when using `updateXML()` to update a node occurring multiple times in a collection. The `UPDATE` statement sets the value of the `text` node belonging to a `Description` element to "The Wizard of Oz", where the current value of the `text` node is "Sisters". The statement includes an `existsNode()` term in the `WHERE` clause that identifies the set of nodes to be updated.

```
SELECT extractValue(value(l),'/Description')
  FROM purchaseorder p,
 table (xmlsequence(extract(p.object_value,
    '/PurchaseOrder/LineItems/LineItem/Description'))) l
 WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;

EXTRACTVALUE(VALUE(L),'/DESCRIPTION')
--------------------------------------------------------------------------------
The Lady Vanishes
The Unbearable Lightness Of Being
Sisters

3 rows selected.

--
UPDATE PURCHASEORDER
       SET object_value = updateXML
                        (
                          object_value,
                          '/PurchaseOrder/LineItems/LineItem/Description/text()',
                          'The Wizard of Oz')
 WHERE existsNode(object_value,
  '/PurchaseOrder/LineItems/LineItem[Description="Sisters"]') = 1
   AND existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;
```

```
1 row updated.

--
SELECT extractValue(value(l),'/Description')
  FROM purchaseorder p,
 table (xmlsequence(extract(p.object_value,
    '/PurchaseOrder/LineItems/LineItem/Description'))) l
 WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVALUE(VALUE(L),'/DESCRIPTION')
--------------------------------------------------------------------------------
The Wizard of Oz
The Wizard of Oz
The Wizard of Oz

3 rows selected.
```

As shown in the preceding example, instead of updating the required node, `updateXML()` updates the values of any `text` node that belongs to the `Description` element. This is actually the expected behavior. *The WHERE clause can only be used to identify which documents must be updated, not which nodes within the document must be updated.* Once the document has been selected the XPath expression passed to `updateXML()` determines which nodes within the document must be updated. In this case the XPath expression identified all three `Description` nodes, and so all three of the associated text nodes were updated. See Example 3–36 for the correct way to update the nodes.

Example 3–36 Updating a Node Occurring Multiple Times Within a Collection Using updateXML(): Correct Usage

To correctly use u`pdateXML()` to update a node occurring multiple times within a collection, use the XPath expression passed to `updateXML()` to identify which nodes in the XML document to update. By introducing the appropriate predicate into the XPath expression you can limit which nodes in the document are updated. The following statement shows the correct way of updating one node within a collection:

```
SELECT extractValue(value(l),'/Description')
  FROM purchaseorder p,
 table (xmlsequence(extract(p.object_value,
    '/PurchaseOrder/LineItems/LineItem/Description'))) l
 WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;
EXTRACTVALUE(VALUE(L),'/DESCRIPTION')
--------------------------------------------------------------------------------
A Night to Remember
The Unbearable Lightness Of Being
Sisters
3 rows selected.
--
UPDATE PURCHASEORDER
    SET object_value = updateXML
       (
       object_value,
       '/PurchaseOrder/LineItems/LineItem/Description[text()="Sisters"]/text()',
       'The Wizard of Oz'
       )
 WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;
```

```
1 row updated.
```

`updateXML()` allows multiple changes to be made to the document in one statement.

```
SELECT extractValue(value(l),'/Description')
  FROM purchaseorder p,
 table (xmlsequence(extract(p.object_value,
    '/PurchaseOrder/LineItems/LineItem/Description'))) l
 WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVALUE(VALUE(L),'/DESCRIPTION')
--------------------------------------------------------------------------------
A Night to Remember
The Unbearable Lightness Of Being
The Wizard of Oz

3 rows selected.
```

Example 3–37 Changing Text Node Values Using updateXML()

This example shows how to change the values of `text` nodes belonging to the `User` and `SpecialInstructions` elements in one statement.

```
column "Cost Center" format A12
column "Instructions" format A40
--
SELECT extractValue(object_value,'/PurchaseOrder/CostCenter') "Cost Center",
       extractValue(object_value,'/PurchaseOrder/SpecialInstructions')
"Instructions"
  FROM PURCHASEORDER
 WHERE existsNode(object_value,
          '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

Cost Center  Instructions
------------ ----------------------------------------
S30          Air Mail

1 row selected.
```

Here is the UPDATE statement that changes the `User` and `SpecialInstruct` element text node values:

```
UPDATE PURCHASEORDER
   SET object_value = updateXML
      (
        object_value,
        '/PurchaseOrder/CostCenter/text()',
        'B40',
        '/PurchaseOrder/SpecialInstructions/text()',
         'Priority Overnight Service'
      )
 WHERE existsNode(object_value,
  '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

1 row updated.
```

Use the following statement to check that the nodes have changed:

```
SELECT extractValue(object_value,'/PurchaseOrder/CostCenter') "Cost Center",
       extractValue(object_value,'/PurchaseOrder/SpecialInstructions')
 "Instructions"
  FROM PURCHASEORDER
 WHERE existsNode(object_value,
```

```
        '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

Cost Center  Instructions
------------ ----------------------------------------
B40          Priority Overnight Service

1 row selected.
```

Updating XML Schema-Based and Non-Schema-Based XML Documents

The way `updateXML()` updates XML documents is primarily determined by whether or not the XML document is XML schema-based or non-XML schema-based, and how the XML document is stored:

- **Storing XML documents in CLOBs.** When `updateXML()` updates a non-XML schema-based or XML schema-based XML document stored as a CLOB, Oracle XML DB performs the update by creating a Document Object Model (DOM) from the XML document and then uses DOM API methods, updates the specified nodes. When the updates have been applied, the updated DOM is returned back to the underlying CLOB.
- **Storing XML documents object-relationally.** When `updateXML()` updates a schema-based XML document stored object-relationally, Oracle XML DB can use XPath rewrite to perform an in-place update of the underlying option. This is a partial-update. Partial-updates translate the XPath expression passed to the `updateXML()` function to an equivalent SQL statement. The update is then performed by executing the SQL statement that directly updates the attributes of underlying objects. This partial-update can result in an `updateXML()` operation that executes many times faster than a DOM-based update. This can make a significant difference when executing a SQL statement that applies `updateXML()` to a large number of documents.

These updates techniques are explained further in the following section.

> **See Also:** Chapter 6, "XML Schema Storage and Query: Advanced Topics"

Namespace Support in Oracle XML DB

Namespace support is a key feature of the W3C XML Recommendations. Oracle XML DB fully supports the W3C Namespace Recommendation. All `XMLType` methods and XML-specific SQL functions work with XPath expressions that include namespace prefixes. All methods and functions accept an optional argument that provides the namespace declarations for correctly resolving namespace prefixes used in XPath expressions.

The `namespace` parameter is required whenever the provided XPath expression contains namespace prefixes. When the `namespace` parameter is not provided, Oracle XML DB makes the following assumptions about the XPath expression:

- If the content of the `XMLType` is *not* based on a registered XML schema any term in the XPath expression that does include a namespace prefix is assumed to be in the `noNamespace` namespace.
- If the content of the `XMLType` is based on a registered XML schema any term in the XPath expression that does not include a namespace prefix is assumed to be in the `targetNamespace` declared by the XML schema. If the XML schema does not declare a `targetnamespace`, this defaults to the `noNamespace` namespace.

- When the `namespace` parameter is provided the parameter must provide an explicit declaration for the default namespace in addition to the prefixed namespaces, unless the default namespace is the `noNamespace` namespace.

Failing to correctly define the namespaces required to resolve XPath expressions results in XPath-based operations not working as expected. When the namespace declarations are incorrect or missing, the result of the operation is normally null, rather than an error. To avoid confusion, Oracle Corporation strongly recommends that you always pass the set of namespace declarations, including the declaration for the default namespace, when any namespaces other than the `noNamespace` namespace are present in either the XPath expression or the target XML document.

Processing XMLType Methods and XML-Specific SQL Functions

Oracle XML DB processes `extract()`, `extractValue()`, `existsNode()`, and `updateXML()` functions and their equivalent `XMLType` methods using DOM-based or SQL-based techniques:

- **DOM-Based XMLType Processing (Functional Evaluation)**. Oracle XML DB performs the required processing by constructing a DOM from the contents of the `XMLType`. It uses methods provided by the DOM API to perform the required operation on the DOM. If the operation involves updating the DOM tree, then the entire XML document has to be written back to disc when the operation is completed. The process of using DOM-based operations on `XMLType` data is referred to as *functional evaluation*.

  The advantage of functional evaluation is that it can be used regardless of whether the `XMLType` is stored using structured or unstructured storage techniques The disadvantage of functional evaluation is that it much more expensive than XPath rewrite, and will not scale across large numbers of XML documents.

  SQL-Based XMLType Processing (XPath rewrite). Oracle XML DB constructs a SQL statement that performs the processing required to complete the function or method. The SQL statement works directly against the object-relational data structures that underly a schema-based `XMLType`. This process is referred to as XPath rewrite, but it can also occur with `updateXML()` operations.

  The advantage of XPath rewrite is that it allows Oracle XML DB to evaluate XPath-based SQL functions and methods at near relational speeds. This allows these operations to scale across large numbers of XML documents. The disadvantage of XPath rewrite is that since it relies on direct access and updating the objects used to store the XML document, it can only be used when the `XMLType` is stored using XML schema-based object-relational storage techniques.

Understanding and Optimizing XPath Rewrite

XPath rewrite improves the performance of SQL statements containing XPath-based functions, by converting the functions into conventional relational SQL statements. By translating XPath-based functions into conventional SQL statements, Oracle XML DB insulates the database optimizer from having to understand the XPath notation and the XML data model. The database optimizer processes the re-written SQL statement in the same manner as any other SQL statement. in this way it can derive an execution plan based on conventional relational algebra. This results in the execution of SQL statements with XPath-based functions with near-relational performance.

When Can XPath Rewrite Occur?

For XPath rewrite to take place the following conditions must be satisfied:

- The `XMLType` column or table containing the XML documents must be based on a registered XML schema.
- The `XMLType` column or table must be stored using structured (object-relational) storage techniques.
- It must be possible to map the nodes referenced by the XPath expression to attributes of the underlying SQL object model.

Understanding the concept of XPath rewrite, and conditions under which XPath rewrite takes place, is key to developing Oracle XML DB applications that deliver satisfactory levels of scalability and performance.

However, XPath rewrite on its own cannot guarantee scalable and performant applications. Like any other SQL statement, the performance of SQL statements generated by XPath rewrite is ultimately determined by the way data is stored on disk and available indexes. Also, as with any other SQL application, a DBA must monitor the database and optimize storage and indexes if the application is to perform well.

Using the EXPLAIN Plan to Tune XPath Rewrites

The good news, from a DBA perspective, is that this information is nothing new. The same skills are required to tune an XML application as for any other database application. All tools that DBAs typically use with SQL-based applications can be used with XML-based applications using Oracle XML DB functions.

Using Indexes to Tune Simple XPath-Based Operations

Example 3–38 shows how to use an `EXPLAIN PLAN` to look at the execution plan for selecting the set of PurchaseOrders created by user `SCOTT`.

Example 3–38 Using an EXPLAIN Plan to Analyze the Selection of PurchaseOrders

```
EXPLAIN PLAN FOR
SELECT extractValue(object_value,'/PurchaseOrder/Reference') "Reference"
  FROM PURCHASEORDER
  WHERE existsNode(object_value,'/PurchaseOrder[User="SBELL"]') = 1;

Explained.

set echo off
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 841749721

-------------------------------------------------------------------------------
| Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |     1 | 22207 |     4   (0)| 00:00:01|
|*  1 |  TABLE ACCESS FULL| PURCHASEORDER |     1 | 22207 |     4   (0)| 00:00:01|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("PURCHASEORDER"."SYS_NC00022$"='SBELL')

Note
-----
   - dynamic sampling used for this statement
```

```
17 rows selected.
```

Using Indexes to Improve Performance of XPath-Based Functions

Oracle XML DB supports the creation of three kinds of index on XML content:

- **Text-based indexes.** These can be created on any `XMLType` table or column.
- **Function-based indexes.** These can be created on any `XMLType` table or column.
- **Conventional B-Tree indexes**. When the `XMLType` table or column is based on structured storage techniques, conventional B-Tree indexes can be created on underlying SQL types.

Indexes are typically created by using the `extractValue()` function, although it is also possible to create indexes based on other `XMLType` functions such as `existsNode()`. During the index creation process Oracle XML DB uses XPath rewrite to determine whether it is possible to map between the nodes referenced in the XPath expression used in the `CREATE INDEX` statement and the attributes of the underlying SQL types. If the nodes in the XPath expression can be mapped to attributes of the SQL types, then the index is created as a conventional B-Tree index on the underlying SQL objects. If the XPath expression cannot be restated using object-relational SQL then a function-based index is created.

Example 3–39 Creating an Index on a Text Node

This example shows creating an index `PURCHASEORDER_USER_INDEX` on the value of the text node belonging to the `User` element.

```
CREATE INDEX PURCHASEORDER_USER_INDEX
   ON PURCHASEORDER
     (extractValue(object_value,'/PurchaseOrder/User'));
```

At first glance the index appears to be a function-based index. However, where the `XMLType` table or column being indexed is based on object-relational storage , XPath rewrite determines whether the index can be re-stated as an index on the underlying SQL types. In this example, the `CREATE INDEX` statement results in the index being created on the `USERID` attribute of the `PURCHASEORDER_T` object.

The following output shows the `EXPLAIN PLAN` output generated when the query is executed after the index has been created.

The `EXPLAIN PLAN` clearly shows that the query plan will make use of the newly created index. The new execution plan is much more scalable.

```
explain plan for
 SELECT extractValue(object_value,'/PurchaseOrder/Reference') "Reference"
   FROM PURCHASEORDER
  WHERE existsNode(object_value,'/PurchaseOrder[User="SBELL"]') = 1;

Explained.

--
set echo off

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------------
Plan hash value: 713050960

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT            |                           |     1 | 22207 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER             |     1 | 22207 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | PURCHASEORDER_USER_INDEX  |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("PURCHASEORDER"."SYS_NC00022$"='SBELL')

Note
-----
   - dynamic sampling used for this statement

18 rows selected.
```

One key benefit of the relational database is that you do not need to change your application logic when the indexes change. This is also true for XML applications that leverage Oracle XML DB capabilities. Once the index has been created the optimizer automatically uses it when appropriate.

Optimizing Operations on Collections

The majority of XML documents contain collections of repeating elements. For Oracle XML DB to be able to efficiently process the collection members it is important that the storage model for managing the collection provides an efficient way of accessing the individual members of the collection. Selecting the correct storage structure makes it possible to index elements within the collection and perform direct operations on individual elements within the collection.

Oracle XML DB offers four ways to manage members of the collection:

- When stored as a `CLOB` value, you cannot directly access members of the collection.
- When a `VARRAY` is stored as a LOB, you cannot directly access members of the collection.

  Storing the members as XML Text managed by a `CLOB` means that any operation on the collection would require parsing the contents of the `CLOB` and then using functional evaluation to perform the required operation.

  Converting the collection into a set of SQL objects that are serialized into a LOB removes the need to parse the documents. However any operations on the members of the collection still require that the collection be loaded from disk into memory before the necessary processing can take place.
- `VARRAY` stored as a nested table, allows direct access to members of the collection.
- `VARRAY` stored as `XMLType`, allows direct access to members of the collection

  In the latter two cases, each member of the `VARRAY` becomes a row in a table. Since each element is stored as a row in a table it can be access directly though SQL.

Using Indexes to Tune Queries on Collections Stored as Nested Tables

The following example shows the execution plan for the query to find the Reference from any document that contains an order for the part with an Id of `717951002372` .

Example 3–40 Generating the EXPLAIN Plan When Selecting a Collection of LineItem Elements from a Nested Table

In this example the collection of `LineItem` elements has been stored as rows in the Index organized, nested table `LINEITEM_TABLE`.

```
explain plan for
 SELECT extractValue(object_value,'/PurchaseOrder/Reference') "Reference"
   FROM PURCHASEORDER
  WHERE existsNode(object_value,
          '/PurchaseOrder/LineItems/LineItem/Part[@Id="717951002372"]') = 1;

Explained.

--
set echo off

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------
Plan hash value: 3281623413

-------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |    21 |   550K|   822   (1)| 00:00:10 |
|*  1 |  HASH JOIN RIGHT SEMI |                    |    21 |   550K|   822   (1)| 00:00:10 |
|*  2 |   INDEX FAST FULL SCAN| LINEITEM_TABLE_IOT |    22 |    99K|   817   (0)| 00:00:10 |
|   3 |   TABLE ACCESS FULL   | PURCHASEORDER      |   132 |  2863K|     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")
   2 - filter("SYS_NC00011$"='717951002372')

Note
-----
   - dynamic sampling used for this statement

20 rows selected.
```

The execution plan shows that the query will be resolved by performing a full scan of the index that contains the contents of the nested table. Each time an entry is found that matches the XPath expression passed to the `existsNode()` function the parent row is located using the value of the `NESTED_TABLE_ID` column. Since the nested table is an Indexed Organized Table (IOT) this plan effectively resolves the query by a full scan of `LINEITEM_TABLE`. This plan may be acceptable when there are only a few hundred documents in the `PURCHASEORDER` table, but would be unacceptable if there are 1000's or 1,000,000's of documents in the table.

To improve the performance of this query create an index that allows direct access to the `NESTED_TABLE_ID` column given the value of the `Id` attribute. Unfortunately, Oracle XML DB does not currently allow indexes on collections to be created using XPath expressions. To create the index you must understand the structure of the SQL object used to manage the `LineItem` elements. Given this information you can create the required index using conventional object-relational SQL.

Here the `LineItem` element is stored as an instance of the `LINEITEM_T` object. The `Part` element is stored as an instance of the SQL Type `PART_T`. The `Id` attribute is mapped to the `PART_NUMBER` attribute. Given this information, you can create a composite index on the `PART_NUMBER` attribute and the `NESTED_TABLE_ID` that will allow direct access to the `PURCHASEORDER` documents that contain `LineItem` elements that reference the required part.

Example 3–41 Creating an Index to Improve Query Performance by Allowing Direct Access to the Nested Table

The following example shows how to use object-relational SQL to create the required index:

```
explain plan for
 SELECT extractValue(object_value,'/PurchaseOrder/Reference') "Reference"
   FROM PURCHASEORDER
  WHERE existsNode(object_value,
    '/PurchaseOrder/LineItems/LineItem/Part[@Id="717951002372"]') = 1;

Explained.

--
set echo off

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------
Plan hash value: 1699938086

-------------------------------------------------------------------------------------------
| Id  | Operation            | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                     |    21 | 13587 |    11  (10)| 00:00:01 |
|*  1 |  HASH JOIN RIGHT SEMI|                     |    21 | 13587 |    11  (10)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN  | LINEITEM_TABLE_IOT  |    22 |  2640 |     6   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN  | LINEITEM_PART_INDEX |    17 |       |     2   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL  | PURCHASEORDER       |   132 | 69564 |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")
   2 - access("SYS_NC00011$"='717951002372')
   3 - access("SYS_NC00011$"='717951002372')

18 rows selected.
```

The plan clearly shows that query plan will make use of the newly created index. The query is now resolved by using `LINEITEM_PART_INDEX` to determine which documents in the `PURCHASEORDER` table satisfy the condition specified in the XPath expression specified in the `existsNode()` function. This query is clearly much more scalable.

In both cases the syntax used to define the query has not changed. XPath rewrite has allowed the optimizer to analyze the query and determine that the new indexes provide a more efficient way to resolve the queries.

EXPLAIN Plan Output with ACL-Based Security Enabled: SYS_CHECKACL() Filter

The `EXPLAIN PLAN` output for a query on an `XMLType` table created as a result of calling `DBMS_XMLSCHEMA.REGISTER_SCHEMA()` will contain a filter that looks similar to the following:

```
3 - filter(SYS_CHECKACL("ACLOID","OWNERID",xmltype(''<privilege
              xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
              http://xmlns.oracle.com/xdb/acl.xsd
              DAV:http://xmlns.oracle.com/xdb/dav.xsd">
                        <read-properties/><read-contents/></privilege>''))=1)
```

This shows that ACL-based security is implemented for this table. In this example the filter is checking that the user performing the SQL query has `read-contents` privilege on each of the documents accessed.

Oracle XML DB repository uses an ACL-based security mechanism that allows access to XML content to be controlled on a document by document basis, rather than a table by table basis. When XML content is accessed using a SQL statement, the `SYS_CHECKACL()` predicate is added to the `WHERE` clause to ensure that the security defined is enforced at the SQL level.

Enforcing ACL-based security does add overhead to the SQL query. If ACL-based security is not required the procedure `DISABLE_HIERARCHY` in the `DBMS_XDBZ` package must be used to turn ACL checking off. After calling this procedure the `SYS_CHECKACL()` filter should no longer appear in the output generated by `EXPLAIN PLAN`.

Example 3–42 Generating an EXPLAIN Plan When XPath Rewrite Does Not Occur

This example shows the kind of `EXPLAIN PLAN` output generated when Oracle XML DB cannot perform XPath rewrite. The key is in line 3. Since the `existsNode()` function appears in the `EXPLAIN` output the query was not re-written.

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("NESTED_TABLE_ID"=:B1)
   2 - access("NESTED_TABLE_ID"=:B1)
   3 - filter(EXISTSNODE(SYS_MAKEXML('C0A5497E8DCF110BE034080020E5
              CF39',3044,"SYS_ALIAS_4"."XMLEXTRA","SYS_ALIAS_4"."XMLDATA"),
              '/PurchaseOrder[User="SBELL"]')=1)
   5 - access("NESTED_TABLE_ID"=:B1)
   6 - access("NESTED_TABLE_ID"=:B1)
```

In this situation Oracle XML DB constructs a pre-filtered results set based on any other conditions specified in the `WHERE` clause of the SQL statement. It then filters all the rows in potential results set to determine which rows belong in the actual results set. The filtering is performed by constructing a DOM on each document and performing a functional evaluation (using the methods defined by the DOM API) to determine whether or not each document is a member of the actual results set, This can result in poor performance when there are a large number of documents in the potential results set. However when other predicates in the `WHERE` clause caused a small number of documents in the potential results set, this may be not be a problem.

`XMLType` and XPath abstractions make it possible for you to develop applications independently of the underlying storage technology. As in conventional relational applications, creating and dropping indexes makes it possible to tune the performance of an application without having to rewrite it.

Accessing Relational Database Content Using XML

Oracle XML DB provides a number of ways to generate XML from relational data.

The most powerful and flexible method is based on the SQL/XML standard. The SQL/XML standard defines a set of functions that allow XML to be generated directly from a SQL `SELECT` statement. These functions make it possible for a SQL statement to generate an XML document, or set of XML documents, rather than a traditional tabular result set. The set of functions defined by the SQL/XML standard are flexible, allowing all most any shape of XML to generated. These functions include the following:

- `xmlElement()` creates a element

- `xmlAttributes()` adds attributes to an element
- `xmlForest()` creates forest of elements
- `xmlAgg()` creates a single element from a collection of elements

See Also: Chapter 15, "Generating XML Data from the Database"

Example 3–43 Using SQL/XML Functions to Generate XML

The following `SELECT` statement generates an XML document containing information from the tables `DEPARTMENTS`, `LOCATIONS`, `COUNTRIES`, `EMPLOYEES`, and `JOBS`:

```
set long 100000
set pages 50
--
select xmlElement
       (
          "Department",
          xmlAttributes( d.DEPARTMENT_ID as "DepartmentId"),
          xmlForest
            (
              d.DEPARTMENT_NAME as "Name"
            ),
            xmlElement
             (
               "Location",
               xmlForest
                 (
                  STREET_ADDRESS as "Address",
                  CITY as "City",
                  STATE_PROVINCE as "State",
                  POSTAL_CODE as "Zip",
                  COUNTRY_NAME as "Country"
                 )
               ),
               xmlElement
               (
                 "EmployeeList",
                   (
                     select xmlAgg
                       (
                         xmlElement
                           (
                            "Employee",
                            xmlAttributes ( e.EMPLOYEE_ID as "employeeNumber" ),
                            xmlForest
                            (
                         e.FIRST_NAME as "FirstName", e.LAST_NAME as "LastName",
                         e.EMAIL as "EmailAddress",
                         e.PHONE_NUMBER as "PHONE_NUMBER",
                         e.HIRE_DATE as "StartDate",
                         j.JOB_TITLE as "JobTitle",
                         e.SALARY as "Salary",
                         m.FIRST_NAME || ' ' || m.LAST_NAME as "Manager"
                           ),
                           xmlElement ( "Commission", e.COMMISSION_PCT )
                         )
                      )
                 from HR.EMPLOYEES e, HR.EMPLOYEES m, HR.JOBS j
                 where e.DEPARTMENT_ID = d.DEPARTMENT_ID
                 and j.JOB_ID = e.JOB_ID
```

```
                and m.EMPLOYEE_ID = e.MANAGER_ID
            )
          )
      ) as XML
  from HR.DEPARTMENTS d, HR.COUNTRIES c, HR.LOCATIONS l
 where DEPARTMENT_NAME = 'Executive'
   and d.LOCATION_ID = l.LOCATION_ID
   and l.COUNTRY_ID  = c.COUNTRY_ID;
```

This returns the following XML:

```
XML
--------------------------------------------------------------------------------
<Department DepartmentId="90"><Name>Executive</Name><Location><Address>2004
 Charade Rd</Address><City>Seattle</City><State>Washingto
n</State><Zip>98199</Zip><Country>United States of
 America</Country></Location><EmployeeList><Employee
 employeeNumber="101"><FirstNa
me>Neena</FirstName><LastName>Kochhar</LastName><EmailAddress>NKOCHHAR</EmailAdd
ess><PHONE_NUMBER>515.123.4568</PHONE_NUMBER><Start
Date>21-SEP-89</StartDate><JobTitle>Administration Vice
 President</JobTitle><Salary>17000</Salary><Manager>Steven King</Manager><Com
mission></Commission></Employee><Employee
 employeeNumber="102"><FirstName>Lex</FirstName><LastName>De
 Haan</LastName><EmailAddress>L
DEHAAN</EmailAddress><PHONE_NUMBER>515.123.4569</PHONE
NUMBER><StartDate>13-JAN-93</StartDate><JobTitle>Administration Vice Presiden
t</JobTitle><Salary>17000</Salary><Manager>Steven
 King</Manager><Commission></Commission></Employee></EmployeeList></Department>
```

This query generates element `Department` for each row in the DEPARTMENTS table.

- Each `Department` element contains attribute `DepartmentID`. The value of `DepartmentID` comes from the `Department_Id` column. The `Department` element contains sub-elements `Name`, `Location`, and `EmployeeList`.
- The text node associated with the `Name` element will come from the NAME column in the `DEPARTMENTS` table.
- The `Location` element will have child elements `Address`, `City`, `State`, `Zip`, and `Country`. These elements are constructed by creating a Forest or named elements from columns in the `LOCATIONS` and `COUNTRIES` tables. The values in the columns become the text node for the named element.
- The `Employeelist` element will contain aggregation of `Employee` Elements. The content of the `EmployeeList` element is created by a sub-select that returns the set of rows in the `EMPLOYEES` table that in turn corresponds to the current department. Each `Employee` element will contain information about the employee. The contents of the elements and attributes for each Employee is taken from the `EMPLOYEES` and `JOBS` tables.

By default, the output generated by the SQL/XML functions is not pretty-printed. This allows the SQL/XML functions to avoid creating a full DOM when generating the required output. By avoiding pretty-printing, Oracle XML DB can avoid overheads associated with DOM and reduce the size of the generated document.

The lack of pretty-printing should not matter to most applications. However it can make it difficult to verify the generated output. When pretty-printing is required, the `extract()` function can force the generation of pretty-printed output. Invoking the `extract()` method on the generated document forces a DOM to be constructed. Printing the results of the `extract()` forces the generation of pretty-printed output.

Since invoking `extract()` forces a conventional DOM to be constructed, this technique should not be used when working with queries that create large documents.

Example 3–44 Forcing Pretty-Printing by Invoking extract() on the Result

Example 3–44 shows how to force pretty-printing by invoking the `extract()` method on the result generated by the `xmlElement()` SQL/XML function.

```
set long 100000
set pages 50
--
select xmlElement
       (
     "Department",
     xmlAttributes( d.DEPARTMENT_ID as "DepartmentId"),
     xmlForest
     (
       d.DEPARTMENT_NAME as "Name"
     ),
     xmlElement
     (
       "Location",
       xmlForest
       (
         STREET_ADDRESS as "Address",
         CITY as "City",
         STATE_PROVINCE as "State",
         POSTAL_CODE as "Zip",
         COUNTRY_NAME as "Country"
       )
     ),
     xmlElement
     (
       "EmployeeList",
       (
         select xmlAgg
            (
              xmlElement
              (
            "Employee",
            xmlAttributes ( e.EMPLOYEE_ID as "employeeNumber" ),
            xmlForest
            (
              e.FIRST_NAME as "FirstName", e.LAST_NAME as "LastName",
              e.EMAIL as "EmailAddress",
              e.PHONE_NUMBER as "PHONE_NUMBER",
              e.HIRE_DATE as "StartDate",
              j.JOB_TITLE as "JobTitle",
              e.SALARY as "Salary",
              m.FIRST_NAME || ' ' || m.LAST_NAME as "Manager"
            ),
            xmlElement ( "Commission", e.COMMISSION_PCT )
              )
            )
           from HR.EMPLOYEES e, HR.EMPLOYEES m, HR.JOBS j
          where e.DEPARTMENT_ID = d.DEPARTMENT_ID
        and j.JOB_ID = e.JOB_ID
        and m.EMPLOYEE_ID = e.MANAGER_ID
       )
     )
       ).extract('/*')  as XML
```

```
  from HR.DEPARTMENTS d, HR.COUNTRIES c, HR.LOCATIONS l
 where DEPARTMENT_NAME = 'Executive'
   and d.LOCATION_ID = l.LOCATION_ID
   and l.COUNTRY_ID  = c.COUNTRY_ID;

XML
--------------------------------------------------------------------------------
<Department DepartmentId="90">
  <Name>Executive</Name>
  <Location>
    <Address>2004 Charade Rd</Address>
    <City>Seattle</City>
    <State>Washington</State>
    <Zip>98199</Zip>
    <Country>United States of America</Country>
  </Location>
  <EmployeeList>
    <Employee employeeNumber="101">
      <FirstName>Neena</FirstName>
      <LastName>Kochhar</LastName>
      <EmailAddress>NKOCHHAR</EmailAddress>
      <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
      <StartDate>21-SEP-89</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
    <Employee employeeNumber="102">
      <FirstName>Lex</FirstName>
      <LastName>De Haan</LastName>
      <EmailAddress>LDEHAAN</EmailAddress>
      <PHONE_NUMBER>515.123.4569</PHONE_NUMBER>
      <StartDate>13-JAN-93</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
  </EmployeeList>
</Department>


1 row selected.
```

All SQL/XML functions return `XMLTypes`. This means that you can use the SQL/XML operators to create `XMLType` views over conventional relational tables. Example 3–45 illustrates this. `XMLType` views are object views. As such each row in the view has to be identified by an object id. The object id must be specified in the `CREATE VIEW` statement.

Example 3–45 Creating XMLType Views Over Conventional Relational Tables

```
CREATE OR REPLACE VIEW DEPARTMENT_XML of XMLType
WITH object id
(

substr(extractValue(object_value,'/Department/Name'),1,128)
)
```

```
AS
select xmlElement
       (
     "Department",
     xmlAttributes( d.DEPARTMENT_ID as "DepartmentId"),
     xmlForest
     (
       d.DEPARTMENT_NAME as "Name"
     ),
     xmlElement
     (
       "Location",
       xmlForest
       (
         STREET_ADDRESS as "Address",
         CITY as "City",
         STATE_PROVINCE as "State",
         POSTAL_CODE as "Zip",
         COUNTRY_NAME as "Country"
       )
     ),
     xmlElement
     (
       "EmployeeList",
       (
         select xmlAgg
            (
              xmlElement
              (
            "Employee",
            xmlAttributes ( e.EMPLOYEE_ID as "employeeNumber" ),
            xmlForest
            (
              e.FIRST_NAME as "FirstName", e.LAST_NAME as "LastName",
              e.EMAIL as "EmailAddress",
              e.PHONE_NUMBER as "PHONE_NUMBER",
              e.HIRE_DATE as "StartDate",
              j.JOB_TITLE as "JobTitle",
              e.SALARY as "Salary",
              m.FIRST_NAME || ' ' || m.LAST_NAME as "Manager"
            ),
            xmlElement ( "Commission", e.COMMISSION_PCT )
              )
            )
           from HR.EMPLOYEES e, HR.EMPLOYEES m, HR.JOBS j
          where e.DEPARTMENT_ID = d.DEPARTMENT_ID
         and j.JOB_ID = e.JOB_ID
         and m.EMPLOYEE_ID = e.MANAGER_ID
       )
     )
       ).extract('/*') as XML
  from HR.DEPARTMENTS d, HR.COUNTRIES c, HR.LOCATIONS l
 where d.LOCATION_ID = l.LOCATION_ID
   and l.COUNTRY_ID  = c.COUNTRY_ID;

View created.
```

The `XMLType` view allows relational data to be persisted as XML content. Rows in `XMLType` views can be persisted as documents in Oracle XML DB repository. The

contents of an XMLType view can be queried using SQL/XML functions. See Example 3–46.

Example 3–46 Querying XMLType Views

Example 3–46 shows a simple query against an XMLType view. The XPath expression passed to the existsNode() function restricts the resultset to the node that contains the information related to the Executive department.

```
SELECT object_value FROM DEPARTMENT_XML
WHERE existsNode(object_value,'/Department[Name="Executive"]') = 1;

OBJECT_VALUE
--------------------------------------------------------------------------------
<Department DepartmentId="90">
  <Name>Executive</Name>
  <Location>
    <Address>2004 Charade Rd</Address>
    <City>Seattle</City>
    <State>Washington</State>
    <Zip>98199</Zip>
    <Country>United States of America</Country>
  </Location>
  <EmployeeList>
    <Employee employeeNumber="101">
      <FirstName>Neena</FirstName>
      <LastName>Kochhar</LastName>
      <EmailAddress>NKOCHHAR</EmailAddress>
      <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
      <StartDate>21-SEP-89</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
    <Employee employeeNumber="102">
      <FirstName>Lex</FirstName>
      <LastName>De Haan</LastName>
      <EmailAddress>LDEHAAN</EmailAddress>
      <PHONE_NUMBER>515.123.4569</PHONE_NUMBER>
      <StartDate>13-JAN-93</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
  </EmployeeList>
</Department>


1 row selected.
```

As can be seen from the following EXPLAIN PLAN output, Oracle XML DB was able to correctly XPath rewrite the existsNode() function on the XMLType row in the XMLType view into a SELECT statement on the underlying relational tables .

```
explain plan for
SELECT object_value FROM DEPARTMENT_XML
WHERE existsNode(object_value,'/Department[Name="Executive"]') = 1;

Explained.
```

```
--
set echo off

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------------
Plan hash value: 1218413855

-------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                   |     1 |    80 |    12  (17)| 00:00:01 |
|   1 |  SORT AGGREGATE                    |                   |     1 |   114 |            |          |
|*  2 |   HASH JOIN                        |                   |    10 |  1140 |     7  (15)| 00:00:01 |
|*  3 |    HASH JOIN                       |                   |    10 |   950 |     5  (20)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID    | EMPLOYEES         |    10 |   680 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN              | EMP_DEPARTMENT_IX |    10 |       |     1   (0)| 00:00:01 |
|   6 |     TABLE ACCESS FULL              | JOBS              |    19 |   513 |     2   (0)| 00:00:01 |
|   7 |    TABLE ACCESS FULL               | EMPLOYEES         |   107 |  2033 |     2   (0)| 00:00:01 |
|*  8 |    FILTER                          |                   |       |       |            |          |
|*  9 |     HASH JOIN                      |                   |     1 |    80 |     5  (20)| 00:00:01 |
|  10 |      NESTED LOOPS                  |                   |    23 |  1403 |     2   (0)| 00:00:01 |
|  11 |       TABLE ACCESS FULL            | LOCATIONS         |    23 |  1127 |     2   (0)| 00:00:01 |
|* 12 |       INDEX UNIQUE SCAN            | COUNTRY_C_ID_PK   |     1 |    12 |            | 00:00:01 |
|  13 |      TABLE ACCESS FULL             | DEPARTMENTS       |    27 |   513 |     2   (0)| 00:00:01 |
|  14 |     SORT AGGREGATE                 |                   |     1 |   114 |            |          |
|* 15 |      HASH JOIN                     |                   |    10 |  1140 |     7  (15)| 00:00:01 |
|* 16 |       HASH JOIN                    |                   |    10 |   950 |     5  (20)| 00:00:01 |
|  17 |        TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |    10 |   680 |     2   (0)| 00:00:01 |
|* 18 |         INDEX RANGE SCAN           | EMP_DEPARTMENT_IX |    10 |       |     1   (0)| 00:00:01 |
|  19 |        TABLE ACCESS FULL           | JOBS              |    19 |   513 |     2   (0)| 00:00:01 |
|  20 |       TABLE ACCESS FULL            | EMPLOYEES         |   107 |  2033 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("M"."EMPLOYEE_ID"="E"."MANAGER_ID")
   3 - access("J"."JOB_ID"="E"."JOB_ID")
   5 - access("E"."DEPARTMENT_ID"=:B1)
   8 - filter(EXISTSNODE("XMLTYPE"."EXTRACT"(XMLELEMENT("Department",XMLATTRIBUTES(TO_CHAR("D".
              "DEPARTMENT_ID") AS "DepartmentId"),XMLELEMENT("Name","D"."DEPARTMENT_NAME"),XMLELEMENT("Locati
              on",CASE  WHEN "STREET_ADDRESS" IS NOT NULL THEN XMLELEMENT("Address","STREET_ADDRESS") ELSE
              NULL END ,XMLELEMENT("City","CITY"),CASE  WHEN "STATE_PROVINCE" IS NOT NULL THEN
              XMLELEMENT("State","STATE_PROVINCE") ELSE NULL END ,CASE  WHEN "POSTAL_CODE" IS NOT NULL THEN
              XMLELEMENT("Zip","POSTAL_CODE") ELSE NULL END ,CASE  WHEN "COUNTRY_NAME" IS NOT NULL THEN
              XMLELEMENT("Country","COUNTRY_NAME") ELSE NULL END ),XMLELEMENT("EmployeeList", (SELECT
              "XMLAGG"(XMLELEMENT("Employee",XMLATTRIBUTES(TO_CHAR("E"."EMPLOYEE_ID") AS
              "employeeNumber"),CASE  WHEN "E"."FIRST_NAME" IS NOT NULL THEN
              XMLELEMENT("FirstName","E"."FIRST_NAME") ELSE NULL END
              ,XMLELEMENT("LastName","E"."LAST_NAME"),XMLELEMENT("EmailAddress","E"."EMAIL"),CASE  WHEN
              "E"."PHONE_NUMBER" IS NOT NULL THEN XMLELEMENT("PHONE_NUMBER","E"."PHONE_NUMBER") ELSE NULL
              END ,XMLELEMENT("StartDate","E"."HIRE_DATE"),XMLELEMENT("JobTitle","J"."JOB_TITLE"),CASE  WHEN

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------------
--------------------
              "E"."SALARY" IS NOT NULL THEN XMLELEMENT("Salary",TO_CHAR("E"."SALARY")) ELSE NULL END ,CASE
              WHEN "M"."FIRST_NAME"||' '||"M"."LAST_NAME" IS NOT NULL THEN
              XMLELEMENT("Manager","M"."FIRST_NAME"||' '||"M"."LAST_NAME") ELSE NULL END
              ,XMLELEMENT("Commission",TO_CHAR("E"."COMMISSION_PCT")))) FROM "HR"."JOBS"
              "J","HR"."EMPLOYEES" "M","HR"."EMPLOYEES" "E" WHERE "E"."DEPARTMENT_ID"=:B1 AND
              "M"."EMPLOYEE_ID"="E"."MANAGER_ID" AND "J"."JOB_ID"="E"."JOB_ID"))),'/*'),'/Department[Name="Ex
              ecutive"]')=1)
   9 - access("D"."LOCATION_ID"="L"."LOCATION_ID")
  12 - access("L"."COUNTRY_ID"="C"."COUNTRY_ID")
```

```
  15 - access("M"."EMPLOYEE_ID"="E"."MANAGER_ID")
  16 - access("J"."JOB_ID"="E"."JOB_ID")
  18 - access("E"."DEPARTMENT_ID"=:B1)

Note
-----
   - warning: inconsistencies found in estimated optimizer costs

63 rows selected.
```

In the current release of Oracle XML DB, XPath rewrites on XML functions that operate on `XMLType` views are only supported when nodes referenced in the XPath expression are *not* descendants of an element created using `xmlAgg()` function.

Generating XML From Relational Tables Using DBUriType

Another way to generate XML from relational data is with the `DBUriType` datatype. `DBUriType` exposes one or more rows in a given table as a single XML document. The name of the root element is derived from the name of the table. The root element contains a set of `ROW` elements. There will be one `ROW` element for each row in the table. The sub-elements of each `ROW` element are derived from the columns in the table or view. Each sub-element will contain a text node that contains the value of the column for the given row.

Example 3–47 shows how to use `DBUriType()` to access the contents of the DEPTARTMENTS table in the HR schema. The example uses the `getXML()` method to return the resulting document as an `XMLType` instance.

Example 3–47 Accessing DEPARTMENTS Table XML Content Using DBUriType() and getXML()

```
set pagesize 100
set linesize 132
set long 10000
--
select dbURIType('/HR/DEPARTMENTS').getXML()
from dual;

DBURITYPE('/HR/DEPARTMENTS').GETXML()
--------------------------------------------------------------------------------
------------------------------------------------
<?xml version="1.0"?>
<DEPARTMENTS>
 <ROW>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Administration</DEPARTMENT_NAME>
  <MANAGER_ID>200</MANAGER_ID>
  <LOCATION_ID>1700</LOCATION_ID>
 </ROW>
 <ROW>
  <DEPARTMENT_ID>20</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Marketing</DEPARTMENT_NAME>
  <MANAGER_ID>201</MANAGER_ID>
  <LOCATION_ID>1800</LOCATION_ID>
 </ROW>
...
```

`DBUriType()` allows XPath notations to be used to control how much of the data in the table is returned when the table is accessed using the `DBUriType()`. Predicates in

the XPath expression allow control over which of the rows in the table are included in the generated document.

Example 3–48 Using a Predicate in the XPath Expression to Restrict Which Rows Are Included

This example demonstrates how to use a predicate in the XPath expression to restrict which rows are included in the generated document. Here the XPath expression restricts the document to those `DEPARTMENT_ID` columns containing the values 10.

```
SELECT dbURIType('/HR/DEPARTMENTS/ROW[DEPARTMENT_ID="10"]').getXML()
FROM dual;

DBURITYPE('/HR/DEPARTMENTS/ROW[DEPARTMENT_ID="10"]').GETXML()
--------------------------------------------------------------------
<?xml version="1.0"?>
 <ROW>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Administration</DEPARTMENT_NAME>
  <MANAGER_ID>200</MANAGER_ID>
  <LOCATION_ID>1700</LOCATION_ID>
 </ROW>


1 row selected.
```

As can be seen from the examples in this section `DBUriType()` provide a simple way to expose some or all rows in a relational table as an XML document(s). The URL passed to `DBUriType()` can be extended to return a single column from the view or table, but in this case the URL must also include predicates that identify a single row in the target table or view. For example, the following URI would return just the value of the DEPARTMENT_NAME column for the DEPARTMENTS row where the DEPARTMENT_ID columns contain the value 10.

```
SELECT dbURIType('/HR/DEPARTMENTS/ROW[DEPARTMENT_ID="10"]/DEPARTMENT
_NAME').getXML()
FROM dual;

DBURITYPE('/HR/DEPARTMENTS/ROW[DEPARTMENT_ID="10"]/DEPARTMENT_NAME').GETXML()
--------------------------------------------------------------------------------
<?xml version="1.0"?>
 <DEPARTMENT_NAME>Administration</DEPARTMENT_NAME>


1 row selected.
```

`DBUriType()` does not provide the flexibility of the SQL/XML operators. Unlike the SQL/XML operators, `DBUriType()` has no way to control the shape of the generated document. The data can only come from a single table or view. The generated document will consist of a set of `ROW` elements or single column from a single row. Each `ROW` element will contain a sub-element for each column in the target table. The names of the sub-elements will be derived from names of the columns.

To control the names of the elements, include columns from more than one table, or control which columns from a table appear in the generated document, it is necessary to create a relational view that exposes the desired set of columns as a single row and then use `DBUriType()` to generate an XML document from the contents of the view.

> **See Also:** Appendix D, "XSLT Primer" for an introduction to the W3C XSL and XSLT recommendations

XSL Transformation

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. Oracle XML DB includes an XSLT processor that allows XSL transformations to be performed inside the database.

Using XSLT with Oracle XML DB

XSL transformation is typically expensive in terms of the amount of memory and processing required. Both the source document and style sheet have to be parsed and loaded into in-memory structures that allow random access to different parts of the documents. Most XSL processors use DOM to provide the in-memory representation of both documents. The XSL processor then applies the style sheet to the source document, generating a third document.

By performing XSL transformation inside the database, alongside the data, Oracle XML DB can provide XML-specific memory optimizations that significantly reduces the memory required to perform the transformation. It can also eliminate overhead associated with parsing the documents. These optimizations are only available when the source for the transformation is a schema-based XML document.

Oracle XML provides three options for invoking the XSL processor.

- `XMLTransform()` SQL function
- `transform()` `XMLType` datatype method
- `DBMS_XSLPROCESSOR` PL/SQL package

All three options expect the source document and XSL style sheet to be provided as an `XMLType`. The result of the transformation is also expected to be a valid XML document. This means that any HTML generated by the transformation must be XHTML, that is valid XML and valid HTML

Example 3–49 XSLT Style Sheet Example: PurchaseOrder.xsl

The following example, `PurchaseOrder.xsl`, is a fragment of an XSLT style sheet:

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xdb="http://xmlns.oracle.com/xdb"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsl:template match="/">
    <html>
      <head/>
      <body bgcolor="#003333" text="#FFFFCC" link="#FFCC00" vlink="#66CC99" alink="#669999">
        <FONT FACE="Arial, Helvetica, sans-serif">
          <xsl:for-each select="PurchaseOrder"/>
          <xsl:for-each select="PurchaseOrder">
            <center>
              <span style="font-family:Arial; font-weight:bold">
                <FONT COLOR="#FF0000">
                  <B>PurchaseOrder </B>
                </FONT>
              </span>
            </center>
            <br/>
            <center>
              <xsl:for-each select="Reference">
                <span style="font-family:Arial; font-weight:bold">
                  <xsl:apply-templates/>
```

```
      </span>
    </xsl:for-each>
  </center>
</xsl:for-each>
<P>
  <xsl:for-each select="PurchaseOrder">
    <br/>
  </xsl:for-each>
  <P/>
  <P>
    <xsl:for-each select="PurchaseOrder">
      <br/>
    </xsl:for-each>
  </P>
</P>
<xsl:for-each select="PurchaseOrder"/>
<xsl:for-each select="PurchaseOrder">
  <table border="0" width="100%" BGCOLOR="#000000">
    <tbody>
      <tr>
        <td WIDTH="296">
          <P>
            <B>
              <FONT SIZE="+1" COLOR="#FF0000" FACE="Arial, Helvetica, sans-serif">Internal</FONT>
            </B>
          </P>
          <table border="0" width="98%" BGCOLOR="#000099">
            <tbody>
              <tr>
                <td WIDTH="49%">
                  <B>
                    <FONT COLOR="#FFFF00">Actions</FONT>
                  </B>
                </td>
                <td WIDTH="51%">
                  <xsl:for-each select="Actions">
                    <xsl:for-each select="Action">
                      <table border="1" WIDTH="143">
                        <xsl:if test="position()=1">
                          <thead>
                            <tr>
                              <td HEIGHT="21">
                                <FONT COLOR="#FFFF00">User</FONT>
                              </td>
                              <td HEIGHT="21">
                                <FONT COLOR="#FFFF00">Date</FONT>
                              </td>
                            </tr>
                          </thead>
                        </xsl:if>
                        <tbody>
                          <tr>
                            <td>
                              <xsl:for-each select="User">
                                <xsl:apply-templates/>
                              </xsl:for-each>
                            </td>
                            <td>
                              <xsl:for-each select="Date">
                                <xsl:apply-templates/>
                              </xsl:for-each>
                            </td>
                          </tr>
                        </tbody>
                      </table>
                    </xsl:for-each>
```

```
          </xsl:for-each>
        </td>
      </tr>
      <tr>
        <td WIDTH="49%">
          <B>
            <FONT COLOR="#FFFF00">Requestor</FONT>
          </B>
        </td>
        <td WIDTH="51%">

          <xsl:for-each select="Requestor">
            <xsl:apply-templates/>
          </xsl:for-each>
        </td>
      </tr>
      <tr>
        <td WIDTH="49%">
          <B>
            <FONT COLOR="#FFFF00">User</FONT>
          </B>
        </td>
        <td WIDTH="51%">
          <xsl:for-each select="User">
            <xsl:apply-templates/>
          </xsl:for-each>
        </td>
      </tr>
      <tr>
        <td WIDTH="49%">
          <B>
            <FONT COLOR="#FFFF00">Cost Center</FONT>
          </B>
        </td>
        <td WIDTH="51%">
          <xsl:for-each select="CostCenter">
            <xsl:apply-templates/>
          </xsl:for-each>
        </td>
      </tr>
    </tbody>
  </table>
</td>
<td width="93"/>
<td valign="top" WIDTH="340">
  <B>
    <FONT COLOR="#FF0000">
      <FONT SIZE="+1">Ship To</FONT>
    </FONT>
  </B>
  <xsl:for-each select="ShippingInstructions">
    <xsl:if test="position()=1"/>
  </xsl:for-each>
  <xsl:for-each select="ShippingInstructions">
    <xsl:if test="position()=1">
      <table border="0" BGCOLOR="#999900">
        <tbody>
          <tr>
            <td WIDTH="126" HEIGHT="24">
              <B>Name</B>
            </td>
            <xsl:for-each select="../ShippingInstructions">
              <td WIDTH="218" HEIGHT="24">
                <xsl:for-each select="name">
                  <xsl:apply-templates/>
                </xsl:for-each>
```

```
                    </td>
                  </xsl:for-each>
                </tr>
                <tr>
                  <td WIDTH="126" HEIGHT="34">
                    <B>Address</B>
                  </td>
                  <xsl:for-each select="../ShippingInstructions">
                    <td WIDTH="218" HEIGHT="34">
                      <xsl:for-each select="address">
                        <span style="white-space:pre">
                          <xsl:apply-templates/>
                        </span>
                      </xsl:for-each>
                    </td>
                  </xsl:for-each>
                </tr>
                <tr>
                  <td WIDTH="126" HEIGHT="32">
                    <B>Telephone</B>
                  </td>
                  <xsl:for-each select="../ShippingInstructions">
                    <td WIDTH="218" HEIGHT="32">
                      <xsl:for-each select="telephone">
                        <xsl:apply-templates/>
                      </xsl:for-each>
                    </td>
                  </xsl:for-each>
                </tr>
              </tbody>
            </table>
          </xsl:if>
        </xsl:for-each>
      </td>
    </tr>
  </tbody>
</table>
<br/>
<B>
  <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
</B>
<br/>
<br/>
<table border="0">

  <xsl:for-each select="LineItems">
    <xsl:for-each select="LineItem">
      <xsl:if test="position()=1">
        <thead>
          <tr bgcolor="#C0C0C0">
            <td>
              <FONT COLOR="#FF0000">
                <B>ItemNumber</B>
              </FONT>
            </td>
            <td>
              <FONT COLOR="#FF0000">
                <B>Description</B>
              </FONT>
            </td>
            <td>
              <FONT COLOR="#FF0000">
                <B>PartId</B>
              </FONT>
            </td>
            <td>
```

```
        <FONT COLOR="#FF0000">
          <B>Quantity</B>
        </FONT>
      </td>
      <td>
        <FONT COLOR="#FF0000">
          <B>UnitPrice</B>
        </FONT>
      </td>
      <td>
        <FONT COLOR="#FF0000">
          <B>Total Price</B>
        </FONT>
      </td>
    </tr>
  </thead>
</xsl:if>
<tbody>
  <tr bgcolor="#DADADA">
    <td>
      <FONT COLOR="#000

1 row selected.
```

The style sheet is a standard XSL style sheet. These is nothing Oracle XML DB- specific about the style sheet. The style sheet can be stored in an `XMLType` table or column or stored as non-schema based XML inside Oracle XML DB repository.

Performing transformations inside the database allows Oracle XML DB to optimize features such as memory usage, I/O operations, and network traffic. These optimizations are particularly effective when the transform operates on a small subset of the nodes in the source document.

In traditional XSL processors the entire source document must be parsed and loaded into memory before XSL processing can begin. This process requires significant amounts of memory and processor. When only a small part of the document is processed this is inefficient.

When Oracle XML DB performs XSL transformations on a schema-based XML document there is no need to parse the document before processing can begin. The lazily loaded virtual DOM eliminates the need to parse the document by loading content directly from disk as the nodes are accessed. The lazy load also reduces the amount of memory required to perform the transformation as only the parts of the document that are processed are loaded into memory.

Example 3–50 Using transform() to Apply an XSL to an XML Document Stored in an XMLType Table

This example shows how to use `XMLType transform()` method to apply an XSL style sheet to a document stored in an `XMLType` table. `XDBUriType()` reads the XSL style sheet from Oracle XML DB repository:

```
set long 10000
set pagesize 100
set linesize 132
--
SELECT XMLTRANSFORM(object
_value,xdbUriType('/home/SCOTT/poSource/xsl/purchaseOrder.xsl').getXML())
  FROM PURCHASEORDER
 WHERE existsNode(object_value,
          '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

XMLTRANSFORM(OBJECT
```

```
_VALUE,XDBURITYPE('/HOME/SCOTT/POSOURCE/XSL/PURCHASEORDER.XSL').GETXML())
--------------------------------------------------------------------------------
<html>
  <head/>
  <body bgcolor="#003333" text="#FFFFCC" link="#FFCC00" vlink="#66CC99" alink="#669999">
    <FONT FACE="Arial, Helvetica, sans-serif">
      <center>
        <span style="font-family:Arial; font-weight:bold">
          <FONT COLOR="#FF0000">
            <B>PurchaseOrder </B>
          </FONT>
        </span>
      </center>
      <br/>
      <center>
        <span style="font-family:Arial; font-weight:bold">SBELL-2002100912333601PDT</span>
      </center>
      <P>
        <br/>
        <P/>
        <P>
          <br/>
        </P>
      </P>
      <table border="0" width="100%" BGCOLOR="#000000">
        <tbody>
          <tr>
            <td WIDTH="296">
              <P>
                <B>
                  <FONT SIZE="+1" COLOR="#FF0000" FACE="Arial, Helvetica,
                        sans-serif">Internal</FONT>
                </B>
              </P>
              <table border="0" width="98%" BGCOLOR="#000099">
                <tbody>
                  <tr>
                    <td WIDTH="49%">
                      <B>
                        <FONT COLOR="#FFFF00">Actions</FONT>
                      </B>
                    </td>
                    <td WIDTH="51%">
                      <table border="1" WIDTH="143">
                        <thead>
                          <tr>
                            <td HEIGHT="21">
                              <FONT COLOR="#FFFF00">User</FONT>
                            </td>
                            <td HEIGHT="21">
                              <FONT COLOR="#FFFF00">Date</FONT>
                            </td>
                          </tr>
                        </thead>
                        <tbody>
                          <tr>
                            <td>SVOLLMAN</td>
                            <td/>
                          </tr>
                        </tbody>
                      </table>
                    </td>
                  </tr>
                  <tr>
                    <td WIDTH="49%">
                      <B>
```

```
                        <FONT COLOR="#FFFF00">Requestor</FONT>
                      </B>
                    </td>
                    <td WIDTH="51%">Sarah J. Bell</td>
                  </tr>
                  <tr>
                    <td WIDTH="49%">
                      <B>
                        <FONT COLOR="#FFFF00">User</FONT>
                      </B>
                    </td>
                    <td WIDTH="51%">SBELL</td>
                  </tr>
                  <tr>
                    <td WIDTH="49%">
                      <B>
                        <FONT COLOR="#FFFF00">Cost Center</FONT>
                      </B>
                    </td>
                    <td WIDTH="51%">B40</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>
              <table border="0" BGCOLOR="#999900">
                <tbody>

XMLTRANSFORM(OBJECT
_VALUE,XDBURITYPE('/HOME/SCOTT/POSOURCE/XSL/PURCHASEORDER.XSL').GETXML())
--------------------------------------------------------------------------------
                  <tr>
                    <td WIDTH="126" HEIGHT="24">
                      <B>Name</B>
                    </td>
                    <td WIDTH="218" HEIGHT="24">Sarah J. Bell</td>
                  </tr>
                  <tr>
                    <td WIDTH="126" HEIGHT="34">
                      <B>Address</B>
                    </td>
                    <td WIDTH="218" HEIGHT="34">
                      <span style="white-space:pre">400 Oracle Parkway
Redwood Shores
CA
94065
USA</span>
                    </td>
                  </tr>
                  <tr>
                    <td WIDTH="126" HEIGHT="32">
                      <B>Telephone</B>
                    </td>
                    <td WIDTH="218" HEIGHT="32">650 506 7400</td>
                  </tr>
                </tbody>
              </table>
            </td>
          </tr>
        </tbody>
```

```
      </table>
      <br/>
      <B>
        <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
      </B>
      <br/>
      <br/>
      <table border="0">
        <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>UnitPrice</B>
              </FONT>
            </td>
            <td>
              <FONT COLOR="#FF0000">
                <B>Total Price</B>
              </FONT>
            </td>
          </tr>
        </thead>
        <tbody>
          <tr bgcolor="#DADADA">
            <td>
              <FONT COLOR="#000000">1</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">A Night to Remember</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">715515009058</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">2</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">39.95</FONT>
            </td>
            <td>
              <FONT FACE="Arial, Helvetica, sans-serif"
COLOR="#000000">79.900000000000006</FONT>
            </td>
          </tr>
        </tbody>
        <tbody>
```

```
          <tr bgcolor="#DADADA">
            <td>

XMLTRANSFORM(OBJECT
_VALUE,XDBURITYPE('/HOME/SCOTT/POSOURCE/XSL/PURCHASEORDER.XSL').GETXML())
--------------------------------------------------------------------------------
              <FONT COLOR="#000000">2</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">The Unbearable Lightness Of Being</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">37429140222</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">2</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">29.95</FONT>
            </td>
            <td>
              <FONT FACE="Arial, Helvetica, sans-serif"
                    COLOR="#000000">59.899999999999999</FONT>
            </td>
          </tr>
        </tbody>
        <tbody>
          <tr bgcolor="#DADADA">
            <td>
              <FONT COLOR="#000000">3</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">The Wizard of Oz</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">715515011020</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">4</FONT>
            </td>
            <td>
              <FONT COLOR="#000000">29.95</FONT>
            </td>
            <td>
              <FONT FACE="Arial, Helvetica, sans-serif"
                    COLOR="#000000">119.799999999999997</FONT>
            </td>
          </tr>
        </tbody>
      </table>
    </FONT>
  </body>
</html>


1 row selected.
```

> **See Also:** Chapter 8, "Transforming and Validating XMLType Data"

Using Oracle XML DB Repository

Oracle XML DB repository makes it possible to organize XML content using a file - folder metaphor. This lets you use a URL to uniquely identify XML documents stored

in the database. This approach appeals to XML developers used to using constructs such as URLs and XPath expressions to identify content.

Oracle XML DB repository is modelled on the DAV standard. The DAV standard uses the term *resource* to describe any file or folder managed by a WebDAV server. A resource consists of a combination of metadata and content. The DAV specification defines the set of metadata properties that a WebDAV server is expected to maintain for each resource and the set of XML documents that a DAV server and DAV-enabled client uses to exchange metadata.

Although Oracle XML DB repository can manage any kind of content, it provides specialized capabilities and optimizations related to managing resources where the content is XML.

Installing and Uninstalling Oracle XML DB Repository

All the metadata and content managed by the Oracle XML DB repository is stored using a set of tables in the database schema owned by database user `XDB`. User `XDB` is a locked account installed with DBCA or by running the script `catqm.sql`. Script `catqm.sql` is located in the directory *`ORACLE_HOME/rdbms/admin`*. The repository can be uninstalled using DBCA or by running the script `catnoqm.sql`. Great care should be taken when running `catnoqm.sql` as this will drop all content stored in the Oracle XML DB repository and invalidate any `XMLType` tables or columns associated with registered XML schemas.

Oracle XML DB Provides Name-Level Not Folder-Level Locking

When using a relational database to maintain hierarchical folder structures, ensuring a high degree of concurrency when adding and removing items in a folder is a challenge. In conventional file system there is no concept of a transaction. Each operation (add a file, create a subfolder, rename a file, delete a file, and so on) is treated as an atomic transaction. Once the operation has completed the change is immediately available to all other users of the file system.

Note: Concurrency: As a consequence of transactional semantics enforced by the database, folders created using SQL statements will not be visible to other database users until the transaction is committed. Concurrent access to the Oracle XML DB repository is controlled by the same mechanism used to control concurrency in Oracle Database. The integration of the repository with Oracle Database provides *strong management options for XML content*.

One key advantage of Oracle XML DB repository is the ability to use SQL for repository operations in the context of a logical transaction. Applications can create long-running transactions that include updates to one or more folders. In this situation a conventional locking strategy that takes an exclusive lock on each updated folder or directory tree would quickly result in significant concurrency problems.

Queued Folder Modifications are Locked Until Committed

Oracle XML DB solves this by providing for name-level locking rather than folder-level locking. Repository operations such as creating, renaming, moving, or deleting a sub-folder or file do not require that your operation be granted an exclusive write lock on the target folder. The repository manages concurrent folder operations by locking the name within the folder rather than the folder itself. The name and the modification type are put on a queue.

Only when the transaction is committed is the folder locked and its contents modified. Hence Oracle XML DB allows multiple applications to perform concurrent updates on the contents of a folder. The queue is also used to manage folder concurrency by preventing two applications from creating objects with the same name.

Queuing folder modifications until commit time also minimizes I/O when a number of changes are made to a single folder in the same transaction.

This is useful when several applications generate files quickly in the same directory, for example when generating trace or log files, or when maintaining a spool directory for printing or email delivery.

Use Protocols or SQL to Access and Process Repository Content

There are two ways to work with content stored in Oracle XML DB repository:

- Using industry standard protocols such as HTTP, WebDAV, or FTP to perform document level operations such as insert, update and delete.
- By directly accessing Oracle XML DB repository content at the table or row level using SQL.

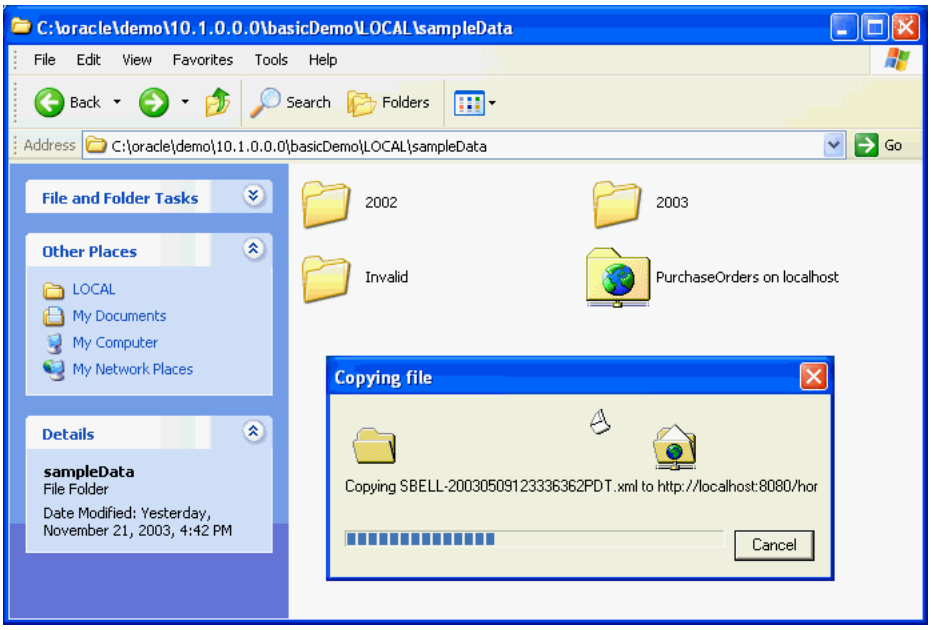
Using Standard Protocols to Store and Retrieve Content

Oracle XML DB supports industry-standard internet protocols such as HTTP, WebDav, and FTP. The combination of protocol support and URL-based access makes it possible to insert, retrieve, update, and delete content stored in Oracle Database from standard desktop applications such as Windows Explorer, Microsoft Word, and XMLSpy.

Figure 3–4 shows Windows Explorer used to insert a folder from the local hard drive into Oracle Database. Windows Explorer includes support for the WebDAV protocol. WebDAV extends the HTTP standard, adding additional verbs that allow an HTTP server to act as a file server.

When a Windows Explorer copy operation or FTP input command is used to transfer a number of documents into Oracle XML DB repository, each `put` or `post` command is treated as a separate atomic operation. This ensures that the client does not get confused if one of the file transfers fails. It also means that changes made to a document through a protocol are visible to other users as soon as the request has been processed.

Figure 3–4 Copying Files into Oracle XML DB Repository



Uploading Content Into Oracle XML DB Using FTP

The following example shows commands issued and output generated when a standard command line FTP tool loads documents into Oracle XML DB repository:

Example 3–51 Uploading Content into Oracle XML DB Repository Using FTP

```
$ ftp mdrake-sun 2100
Connected to mdrake-sun.
220 mdrake-sun FTP Server (Oracle XML DB/Oracle Database 10g Enterprise Edition
Release 10.1.0.1.0 - Beta) ready.
Name (mdrake-sun:oracle10): SCOTT
331 pass required for SCOTT
Password:
230 SCOTT logged in
ftp> cd /home/SCOTT
250 CWD Command successful
ftp> mkdir PurchaseOrders
257 MKD Command successful
ftp> cd PurchaseOrders
250 CWD Command successful
ftp> mkdir 2002
257 MKD Command successful
ftp> cd 2002
250 CWD Command successful
ftp> mkdir "Apr"
257 MKD Command successful
ftp> put "Apr/AMCEWEN-20021009123336171PDT.xml"
"Apr/AMCEWEN-20021009123336171PDT.xml"
200 PORT Command successful
150 ASCII Data Connection
226 ASCII Transfer Complete
local: Apr/AMCEWEN-20021009123336171PDT.xml remote:
Apr/AMCEWEN-20021009123336171PDT.xml
4718 bytes sent in 0.0017 seconds (2683.41 Kbytes/s)
ftp> put "Apr/AMCEWEN-20021009123336271PDT.xml"
```

```
"Apr/AMCEWEN-20021009123336271PDT.xml"
200 PORT Command successful
150 ASCII Data Connection
226 ASCII Transfer Complete
local: Apr/AMCEWEN-20021009123336271PDT.xml remote:
Apr/AMCEWEN-20021009123336271PDT.xml
4800 bytes sent in 0.0014 seconds (3357.81 Kbytes/s)
.....
ftp> cd "Apr"
250 CWD Command successful
ftp> ls -l
200 PORT Command successful
150 ASCII Data Connection
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 AMCEWEN-20021009123336171PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 AMCEWEN-20021009123336271PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 EABEL-20021009123336251PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 PTUCKER-20021009123336191PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 PTUCKER-20021009123336291PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 SBELL-20021009123336231PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 SBELL-20021009123336331PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 SKING-20021009123336321PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 SMCCAIN-20021009123336151PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 SMCCAIN-20021009123336341PDT.xml
-rw-r--r1 SCOTT oracle 0 JUN 24 15:41 VJONES-20021009123336301PDT.xml
226 ASCII Transfer Complete
remote: -l
959 bytes received in 0.0027 seconds (349.45 Kbytes/s)
ftp> cd ".."
250 CWD Command successful
....
ftp> quit
221 QUIT Goodbye.
$
```

The key point demonstrated by both these examples is that neither Windows Explorer nor the FTP tool are aware that they are working with Oracle XML DB. Since the tools and Oracle XML DB both support open Internet protocols they simply work with each other out of the box.

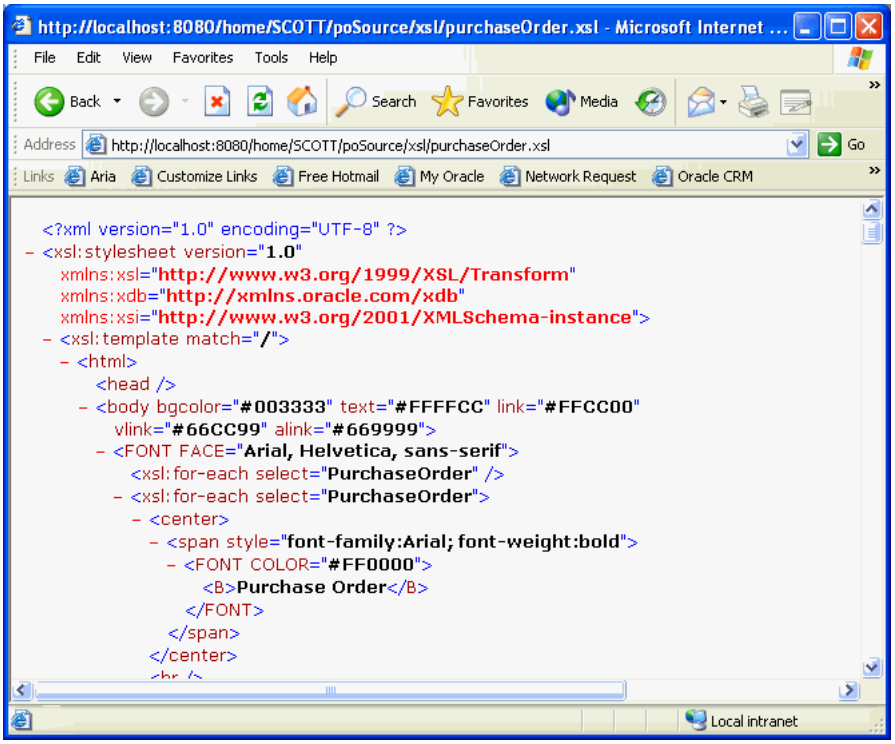
Any tool that understands the WebDAV or FTP protocol can be used to create content managed by Oracle XML DB repository. No additional software has to installed on the client or the mid-tier.

When the contents of the folders are viewed using a tool such as Windows Explorer or FTP, the length of any schema-based XML documents contained in the folder is shown as 0 bytes. This was designed as such for two reasons:

- Firstly, it is not clear what the size of the document should be. Is it the size of the CLOB generated by printing the document, or the number of bytes required to store the objects used to persist the document inside the database?
- Secondly, regardless of which definition is chosen, calculating and maintaining this information is costly.

Figure 3–5 shows Internet Explorer using a URL and the HTTP protocol to view an XML document stored in the database.

Figure 3–5 Path-Based Access Using HTTP and a URL



Accessing Oracle XML DB Repository Programmatically

Oracle XML DB repository can be accessed and updated directly from SQL. This means that any application or programming language that can use SQL to interact with Oracle Database can also access and update content stored in Oracle XML DB repository. Oracle XML DB includes PL/SQL package, `DBMS_XDB`, that provides methods that allow resources to be created, modified, and deleted in a programmatically.

Example 3–52 Creating a Text Document Resource Using DBMS_XDB

This example shows how to create a resource using `DBMS_XDB`. Here the resource will be a simple text document containing the supplied text.

```
declare
  res boolean;
begin
  res := dbms_xdb.createResource('/home/SCOTT/NurseryRhyme.txt',
                bfilename('XMLDIR','DocExample01.txt'),
                nls_charset_id('AL32UTF8'));
end;
/


PL/SQL procedure successfully completed.
```

Accessing the Content of Documents Using SQL

You can access the content of documents stored in Oracle XML DB repository in several ways. The easiest way is to use `XDBUriType`. `XDBUriType` uses a URL to

specify which resource to access. The URL passed to the XDBUriType is assumed to start at the root of XML DB repository. XDBUriType provides methods getBLOB(), getCLOB(), and getXML() to access the different kinds of content that can be associated with a resource.

Example 3–53 Using XDBUriType to Access a Text Document in the Repository

This example shows how to use XDBUriType to access the content of the text document:

```
SELECT xdburitype('/home/SCOTT/NurseryRhyme.txt').getClob()
  FROM dual;

XDBURITYPE('/HOME/SCOTT/NURSERYRHYME.TXT').GETCLOB()
--------------------------------------------------------------------------------
Mary had a little lamb
It's fleece was white as snow
and every where that Mary went
that lamb was sure to go


1 row selected.
```

Example 3–54 Using XDBUriType and a Repository Resource to Access Content

The contents of a document can also be accessed using the resource document. This example shows how to access the content of a text document:

```
SELECT dbms_xmlgen.convert
       (
     extract
     (
       res,
       '/Resource/Contents/text/text()',
       'xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"'
     ).getClobVal(),
     1
       )
  FROM RESOURCE_VIEW r
 WHERE equals_path(res,'/home/SCOTT/NurseryRhyme.txt') = 1;

DBMS_XMLGEN.CONVERT(EXTRACT(RES,'/RESOURCE/CONTENTS/TEXT/TEXT()','XMLNS="HTTP://
--------------------------------------------------------------------------------
Mary had a little lamb
It's fleece was white as snow
and every where that Mary went
that lamb was sure to go


1 row selected.
```

extract() rather than extractValue() is used to access the text() node. This returns the content of the text() node as an XMLType, which makes it possible to access the content of the node using getCLOBVal(). Hence you can access the content of documents larger than 4K. Here DBMS_XMLGEN.convert removes any entity escaping from the text.

Example 3–55 Accessing Schema-Based XML Documents Using the Resource and Namespace Prefixes

The content of non-schema-based and schema-based XML documents can also be accessed through the resource. This example shows how to use an XPath expression that includes nodes from the resource document and nodes from the XML document to access the contents of a `PurchaseOrder` document using the resource.

```
SELECT extractValue(value(l),'/Description')
  FROM RESOURCE_VIEW r,
 table (
     xmlsequence
     (
       extract
       (
         res,
         '/r:Resource/r:Contents/PurchaseOrder/LineItems/LineItem/Description',
         'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"'
       )
     )
       ) l
 WHERE equals_path(res,
      '/home/SCOTT/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml') = 1;

EXTRACTVALUE(VALUE(L),'/DESCRIPTION')
-----------------------------------------
A Night to Remember
The Unbearable Lightness Of Being
The Wizard of Oz

3 rows selected.
```

In this case a namespace prefix was used to identify which nodes in the XPath expression are members of the resource namespace. This was necessary as the `PurchaseOrder` XML schema does not define a namespace and it was not possible to apply a namespace prefix to nodes in the PurchaseOrder document.

Accessing the Content of XML Schema-Based Documents

The content of a schema-based XML document can be accessed in two ways.

- In the same manner as for non-schema-based XML documents, by using the resource document. This allows the `RESOURCE_VIEW` to be used to query different types of schema-based XML documents with a single SQL statement.
- As a row in the default table that was defined when the XML schema was registered with Oracle XML DB.

Using the XMLRef Element in Joins to Access Resource Content in the Repository

The `XMLRef` element in the resource document provides the join key required when a SQL statement needs to access or update metadata and content as part of a single operation.

The following queries use joins based on the value of the `XMLRef` to access resource content.

Example 3–56 Querying Repository Resource Data Using Ref() and the XMLRef Element

This example locates a row in the defaultTable based on a path in Oracle XML DB repository. SQL `ref()` function locates the target row in the default table based on value of the `XMLRef` element contained in the resource document.

```
SELECT extractValue(value(l),'/Description')
  FROM RESOURCE_VIEW r, PURCHASEORDER p,
 TABLE (
     xmlsequence
     (
       extract
       (
         object_value,
         '/PurchaseOrder/LineItems/LineItem/Description'
       )
     )
       ) l
 WHERE equals_path(res,
        '/home/SCOTT/PurchaseOrders/2002/Mar/SBELL-20020100912333601PDT.xml') = 1
   AND ref(p) = extractValue(res,'/Resource/XMLRef');

EXTRACTVALUE(VALUE(L),'/DESCRIPTION')
--------------------------------------------------------------------------------
A Night to Remember
The Unbearable Lightness Of Being
The Wizard of Oz

3 rows selected.
```

Example 3–57 Selecting XML Document Fragments Based on Metadata, Path, and Content

This example shows how this technique makes it possible to select fragments from XML documents based on metadata, path, and content. The statement returns the value of the `Reference` element for documents foldered under the path `/home/SCOTT/PurchaseOrders/2002/Mar` and contain orders for part `715515009058`.

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference')
  FROM RESOURCE_VIEW r, PURCHASEORDER p
 WHERE under_path(res,'/home/SCOTT/PurchaseOrders/2002/Mar') = 1
   AND ref(p) = extractValue(res,'/Resource/XMLRef')
   AND existsNode(object_value,
          '/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]') = 1;

EXTRACTVALUE(OBJECT_VALUE,'/PU
------------------------------
CJOHNSON-20021009123335851PDT
LSMITH-20020100912333661PDT
SBELL-20020100912333601PDT

3 rows selected.
```

In general when accessing the content of schema-based XML documents, joining `RESOURCE_VIEW` or `PATH_VIEW` with the default table is more efficient than using the `RESOURCE_VIEW` or `PATH_VIEW` on their own. The explicit join between the resource document and the default table tells Oracle XML DB that the SQL statement will only work on one type of XML document. This allows XPath rewrite to be used to optimize the operation on the default table as well as the operation on the resource.

Updating the Content of Documents Stored in Oracle XML DB Repository

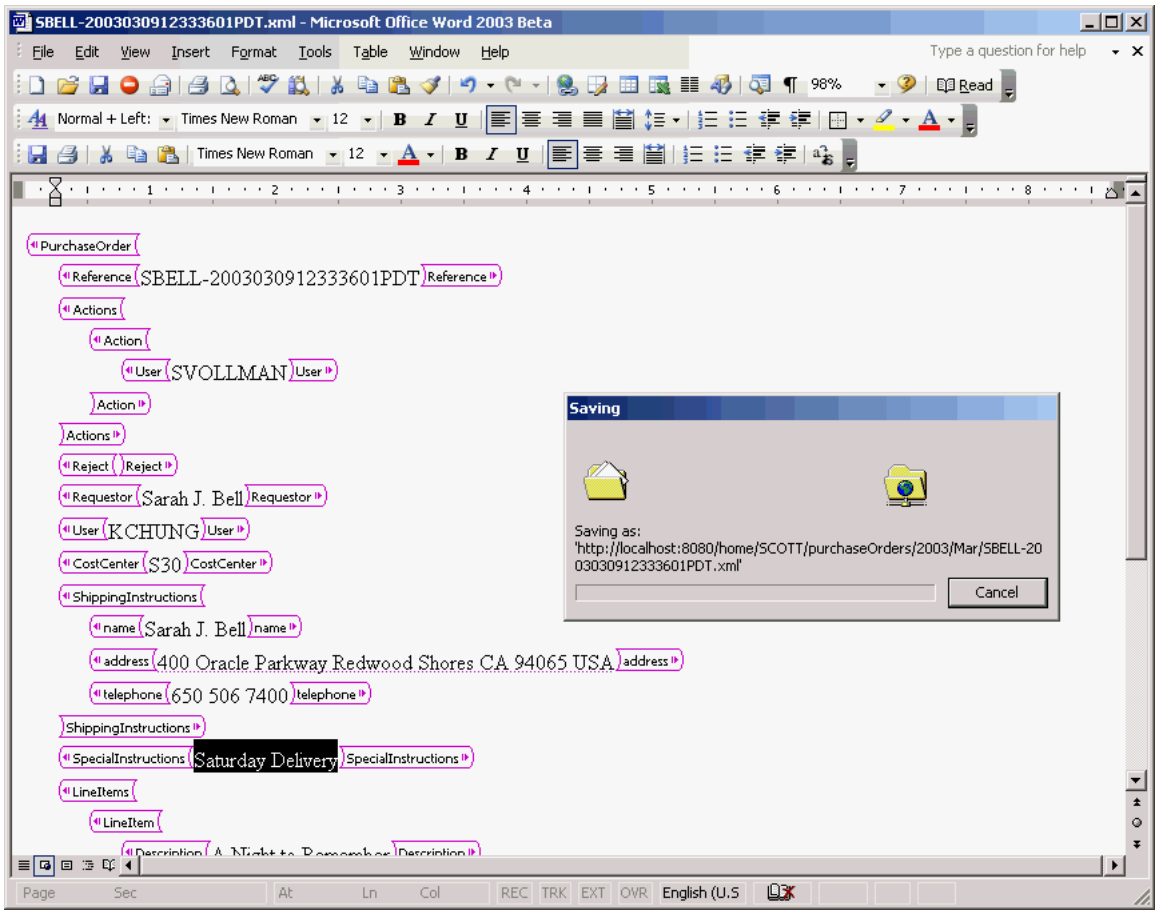
You can also update the content of documents stored in the Oracle XML DB repository using protocols or SQL.

Updating Repository Content Using Protocols

The most popular content authoring tools now support HTTP, FTP, and WebDAV protocols. These tools can use a URL and the HTTP `get` verb to access the content of a document, and the HTTP `put` verb to save the contents of a document. Hence, given the appropriate access permissions, a simple URL is all you need to access and edit content stored in Oracle XML DB repository.

Figure 3–6 shows how with the WebDAV support, included in Microsoft Word, you can use Microsoft Word to update and edit a document stored in Oracle XML DB repository.

Figure 3–6 Using Microsoft Word to Update and Edit Content Stored in Oracle XML DB



When an editor like Microsoft Word updates an XML document stored in Oracle XML DB the database receives an input stream containing the new content of the document. Unfortunately products such as Word do not provide Oracle XML DB with any way of identifying what changes have taken place in the document. This means that partial-updates are not possible and it is necessary to re-parse the entire document, replacing all the objects derived from the original document with objects derived from the new content.

Updating Repository Content Using SQL

The `updatexml()` function can be used to update the content of any document stored in Oracle XML DB repository. The content of the document can be updated by updating the resource document, or in the case of schema-based XML documents, by updating the default table that contains the content of the document.

Example 3–58 Updating the Contents of a Text Document Using UPDATE and updateXML() on the Resource

This example shows how to update the contents of a simple text document using the SQL `UPDATE` statement `and updateXML()` on the resource document. XPath expression is passed to `updateXML()` to identify the text node belonging to the element `text` contained in element `/Resource/Contents` as the target of the update operation.

```
declare
  file bfile;
  contents clob;

  dest_offset     number := 1;
  src_offset      number := 1;
  lang_context    number := 0;
  conv_warning    number := 0;

begin
  file := bfilename('XMLDIR','DocExample02.txt');
  DBMS_LOB.createTemporary(contents,true,DBMS_LOB.SESSION);
  DBMS_LOB.fileopen(file, DBMS_LOB.file_readonly);
  DBMS_LOB.loadClobfromFile
  (
     contents,
     file,
     DBMS_LOB.getLength(file),
     dest_offset,
     src_offset,
     nls_charset_id('AL32UTF8'),
     lang_context,
     conv_warning
  );
  DBMS_LOB.fileclose(file);
  UPDATE RESOURCE_VIEW
   SET res = updateXML
         (
           res,
           '/Resource/Contents/text/text()',
           contents,
           'xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"'
         )
 WHERE equals_path(res,'/home/SCOTT/NurseryRhyme.txt') = 1;
 dbms_lob.freeTemporary(contents);
end;
/


PL/SQL procedure successfully completed.
```

The technique for updating the content of a document by updating the associated resource has the advantage that it can be used to update any kind of document stored in XML DB repository.

Example 3–59 Updating a Node in the XML Document Using UPDATE and updateXML()

This example shows how to update a node in an XML document by performing an update on the resource document. Here `updateXML()` changes the value of the text node associated with the `User` element.

```
UPDATE RESOURCE_VIEW
   SET res = updateXML
         (
           res,
           '/r:Resource/r:Contents/PurchaseOrder/User/text()',
           'SKING',
           'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"'
         )
 WHERE equals_path(res,
       '/home/SCOTT/PurchaseOrders/2002/Mar/SBELL-20021009123336601PDT.xml') = 1;

1 row updated.
```

Updating XML Schema-Based Documents in the Repository

You can update XML schema-based XML documents by performing the update operation directly on the default table used to manage the content of the document. If the document must be located by a `WHERE` clause that includes a path or conditions based on metadata, then the `UPDATE` statement must use a join between the resource and the default table.

In general when updating the contents of XML schema-based XML documents, joining the `RESOURCE_VIEW` or `PATH_VIEW` with the default table is more efficient than using the `RESOURCE_VIEW` or `PATH_VIEW` on their own. The explicit join between the resource document and the default table tells Oracle XML DB that the SQL statement will only work on one type of XML document. This allows a partial-update to be used on the default table and resource.

Example 3–60 Updating XML Schema-Based Documents in the Repository

Here `updateXML()` operates on the default table with the target row identified by a path. The row to be updated is identified by a `Ref`. The value of the row is obtained from the resource document identified by the `equals_path()` function. This effectively limits the update to the row corresponding to the resource identified by the specified path.

```
UPDATE PURCHASEORDER p
   SET object_value = updateXML
              (
             object_value,
             '/PurchaseOrder/User/text()','SBELL')
 WHERE ref(p) =
       (
     SELECT extractValue(res,'/Resource/XMLRef')
       FROM RESOURCE_VIEW
      WHERE equals_path(res,
       '/home/SCOTT/PurchaseOrders/2002/Mar/SBELL-20021009123336601PDT.xml') = 1
       );

1 row updated.
```

Controlling Access to Repository Data

You can control access to the resources in the XML DB repository by using Access Control Lists (ACLs). An ACL is a list of access control entries, each of which grants or

denies a set of privileges to a specific principal. The principal can be a database user, a database role, an LDAP user, an LDAP group or the special principal 'dav:owner' that refers to the owner of the resource. Each resource in the repository is protected by an ACL. The ACL determines what privileges, such as 'read-properties' and 'update', a user has on the resource. Each repository operation includes a check of the ACL to determine if the current user is allowed to perform the operation.

By default, a new resource inherits the ACL of its parent folder. But you can set the ACL of a resource using the `DBMS_XDB.setACL()` procedure. For more details on XML DB resource security, see Chapter 23, "Oracle XML DB Resource Security".

In the following example, the current user is `SCOTT`. The query gives the number of resources in the folder `/public`. Assume that there are only 2 resources in this folder: f1 and f2. Also assume that the ACL on f1 grants the `read-properties` privilege to SCOTT while the ACL on f2 does not grant `SCOTT` any privileges. A user needs the 'read-properties' privilege on a resource for it to be visible to the user. The result of the query is 1 since only f1 is visible to `SCOTT`.

```
select count(*) from resource_view r
  where under_path(r.res, '/public') = 1;

  COUNT(*)
----------
         1
```

XML DB Transactional Semantics

When working from SQL, normal transactional behavior is enforced. Multiple `updatexml()` statements can be used within a single logical unit of work. Changes made through `updatexml()` are not visible to other database users until the transaction is committed. At any point, `rollback` can be used to back out the set of changes made since the last commit.

Querying Metadata and the Folder Hierarchy

In Oracle XML DB the metadata for each resource is preserved as an XML document. The structure of these documents is defined by the `XDBResource.xsd` XML schema. This schema is registered as a global XML schema at URL `http://xmlns.oracle.com/xdb/XDBResource.xsd`.

Oracle XML DB allows you access to metadata and information about the folder hierarchy using two public views, `RESOURCE_VIEW` and `PATH_VIEW`.

RESOURCE_VIEW

`RESOURCE_VIEW` contains one entry for each file or folder stored in XML DB repository. The view consists of two columns. The RES column contains the resource document that manages the metadata properties associated with the document. The `ANY_PATH` column contains a valid URL that the current user can pass to `XDBUriType` in order to access the content the document. In the cases of non-binary content the resource document will also contain the content of the document.

Oracle XML DB supports the concept of linking. Linking makes it possible to define multiple paths to a given document. A separate XML document, called the link-properties document, maintains metadata properties that are specific to the link, rather than to the resource. Whenever a resource is created an initial link is also created.

PATH_VIEW

PATH_VIEW exposes the link-properties documents. There is one entry it PATH_VIEW for each possible path to a document. The PATH_VIEW consists of three columns. The RES columns contains the resource document that this link points at. The PATH column contains the Path that the link allows to be used to access the resource. The LINK column contains the link-properties document for this PATH.

> **See Also:** Chapter 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"

Example 3–61 Viewing RESOURCE_VIEW and PATH_VIEW Structures

The following example shows the description of the public views RESOURCE_VIEW and PATH_VIEW:

```
desc RESOURCE_VIEW
 Name                           Null?    Type
 ----------------------------------------------------------------------
 RES                                     SYS.XMLTYPE(XMLSchema
                                           "http://xmlns.oracle.com/xd
                                         b/XDBResource.xsd" Element "Resource")
 ANY_PATH                                VARCHAR2(4000)
 RESID                                   RAW(16)

--
desc PATH_VIEW
 Name                           Null?    Type
 ----------------------------------------------------------------
 PATH                                    VARCHAR2(1024)
 RES                                     SYS.XMLTYPE(XMLSchema
                                          "http://xmlns.oracle.com/xd
                                         b/XDBResource.xsd" Element "Resource")
 LINK                                    SYS.XMLTYPE
 RESID                                   RAW(16)
```

Oracle XML DB provides two new functions, equals_path() and under_path(), that can be used to perform folder-restricted queries. Folder-restricted queries limit SQL statements that operate on the RESOURCE_VIEW or PATH_VIEW to documents that are at a particular location in Oracle XML DB folder hierarchy. equals_path() restricts the statement to a single document identified by the specified path. under_path() restricts the statement to those documents that exist beneath a certain point in the hierarchy.

Example 3–62 Accessing Resources Using equals_path() and RESOURCE_VIEW

The following query uses the equals_path() function and RESOURCE_VIEW to access the resource created in Example 3–61.

```
SELECT r.res.getClobVal()
  FROM RESOURCE_VIEW r
 WHERE equals_path(res,'/home/SCOTT/NurseryRhyme.txt') = 1;

R.RES.GETCLOBVAL()
--------------------------------------------------------------------------------
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd" Hidden="false" Inv
alid="false" Container="false" CustomRslv="false" VersionHistory="false" StickyR
ef="true">
  <CreationDate>2003-12-08T19:03:06.584000</CreationDate>
  <ModificationDate>2003-12-08T19:03:07.456000</ModificationDate>
  <DisplayName>NurseryRhyme.txt</DisplayName>
```

```
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>text/plain</ContentType>
  <RefCount>1</RefCount>
  <ACL>
    <acl description="Private:All privileges to OWNER only and not accessible to
 others" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.ora
cle.com/xdb/acl.xsd                          http://xmlns.oracle.com/xdb/acl.xs
d">
      <ace>
        <principal>dav:owner</principal>
        <grant>true</grant>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>SCOTT</Owner>
  <Creator>SCOTT</Creator>
  <LastModifier>SCOTT</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#text</SchemaElement>
  <Contents>
    <text>Hickory Dickory Dock
The Mouse ran up the clock
The clock struck one
The Mouse ran down
Hickory Dickory Dock

</text>
  </Contents>
</Resource>


1 row selected.
```

As this example shows, a resource document is an XML document that captures the set of metadata defined by the DAV standard. The metadata includes information such as Creation Date, Creator, Owner, Last Modification Date, and Display Name. The content of the resource document can be queried and updated just like any other XML document, using functions such as `extract()`, `extractValue()`, `existsNode()`, and `updateXML()`.

Querying Resources Stored in RESOURCE_VIEW and PATH_VIEW

The following examples demonstrate simple folder-restricted queries against resource documents stored in the `RESOURCE_VIEW` and `PATH_VIEW`.

Example 3–63 Determining the Path to XSL Style Sheets Stored in the Repository

The first query finds a path to each of XSL style sheet stored in Oracle XML DB repository. It performs a search based on the `DisplayName` ending in `.xsl`. Unlike a conventional file system, Oracle XML DB can use the power of Oracle Database to resolve this query.

```
SELECT any_path
 FROM RESOURCE_VIEW
 WHERE extractValue(RES,'/Resource/DisplayName') like '%.xsl';
```

```
ANY_PATH
--------------------------------------------------------------------------------
/home/SCOTT/poSource/xsl/empdept.xsl
/home/SCOTT/poSource/xsl/purchaseOrder.xsl

2 rows selected.
```

Example 3–64 Counting Resources Under a Path

This example counts the number of resources (files and folders) under the path `/home/SCOTT/PurchaseOrders`. Using `RESOURCE_VIEW` rather than `PATH_VIEW` ensures that resources that any resources that are the target of multiple links are only counted once. The `under_path()` function restricts the resultset to documents that can be accessed using a path where the path starts from `/home/SCOTT/PurchaseOrders`.

```
SELECT count(*)
   FROM RESOURCE_VIEW
   WHERE under_path (RES,'/home/SCOTT/PurchaseOrders') = 1;

  COUNT(*)
----------
       145
1 row selected.
```

Example 3–65 Listing the Folder Contents in a Path

This query lists the contents of the folder identified by path `/home/SCOTT/PurchaseOrders/2002/Apr`. This is effectively a directory listing of the folder.

```
SELECT PATH
FROM PATH_VIEW
WHERE under_path(RES,'/home/SCOTT/PurchaseOrders/2002/Apr') = 1;

PATH
--------------------------------------------------------------------------------
/home/SCOTT/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/PTUCKER-20021009123336191PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/PTUCKER-20021009123336291PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SBELL-20021009123336231PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SBELL-20021009123336331PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SKING-20021009123336321PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336151PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/VJONES-20021009123336301PDT.xml

11 rows selected.
```

Example 3–66 Listing the Links Contained in a Folder

This query lists the set of links contained in the folder identified by the path `/home/SCOTT/PurchaseOrders/2002/Apr` where the `DisplayName` element in the associated resource starts with an `S`.

```
SELECT PATH
  FROM PATH_VIEW
 WHERE extractValue(RES,'/Resource/DisplayName') like 'S%'
```

```
  AND under_path(RES,'/home/SCOTT/PurchaseOrders/2002/Apr') = 1;

PATH
--------------------------------------------------------------------------------
/home/SCOTT/PurchaseOrders/2002/Apr/SBELL-20021009123336231PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SBELL-20021009123336331PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SKING-20021009123336321PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336151PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml

5 rows selected.
```

Example 3–67 Finding the Path to Resources in the Repository Containing a PO XML Document

This query finds a path to each of the resources in the repository that contain a PurchaseOrder XML document. The documents are identified based on the metadata property `SchemaElement` that identifies the XML schema URL and global element for schema-based XML stored in Oracle XML DB repository.

```
SELECT ANY_PATH
  FROM RESOURCE_VIEW
 WHERE existsNode(RES,

'/Resource[SchemaElement="http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOr
der.xsd#PurchaseOrder"]') = 1;
```

This returns the following paths each of which contain a PurchaseOrder XML document:

```
ANY_PATH
--------------------------------------------------------------------------------
/home/SCOTT/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
/home/SCOTT/PurchaseOrders/2002/Apr/PTUCKER-20021009123336191PDT.xml
```

The Oracle XML DB Hierarchical Index

In a conventional relational database, path-based access and folder-restricted queries would have to be implemented using `CONNECT BY` operations. Such queries are expensive and path-based access and folder-restricted queries would become very inefficient as the number of documents and depth of the folder hierarchy increases.

To address this issue, Oracle XML DB introduces a new index, the *hierarchical index*. The hierarchical index allows the database to resolve folder-restricted queries without relying on a `CONNECT BY` operation. Hence Oracle XML DB can execute path-based and folder-restricted queries efficiently. The hierarchical index is implemented as an Oracle domain index. This is the same technique used to add Oracle Text indexing support and many other advanced index types to the database.

Example 3–68 EXPLAIN Plan Output for a Folder-Restricted Query

This example shows the `EXPLAIN PLAN` output generated for a folder-restricted query. As shown, the hierarchical index (`XDBHI_IDX`) will be used to resolve the query.

```
explain plan for
 SELECT PATH
   FROM PATH_VIEW
```

```
  WHERE extractValue(RES,'/Resource/DisplayName') like 'S%'
    AND under_path(RES,'/home/SCOTT/PurchaseOrders/2002/Apr') = 1;

Explained.

--
set echo off

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
Plan hash value: 2568289845

------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |   300 | 62100 |    28   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                       |               |   300 | 62100 |    28   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                      |               |   300 | 57000 |    28   (0)| 00:00:01 |
|   3 |    NESTED LOOPS                     |               |   300 | 44400 |    28   (0)| 00:00:01 |
|*  4 |     TABLE ACCESS BY INDEX ROWID     | XDB$RESOURCE  |     1 |   146 |     4   (0)| 00:00:01 |
|   5 |      DOMAIN INDEX                   | XDBHI_IDX     |       |       |            |          |
|   6 |     COLLECTION ITERATOR PICKLER FETCH|               |       |       |            |          |
|*  7 |    INDEX UNIQUE SCAN                | XDB_PK_H_LINK |     1 |    42 |            | 00:00:01 |
|*  8 |   INDEX UNIQUE SCAN                 | SYS_C002901   |     1 |    17 |            | 00:00:01 |
------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter("P"."SYS_NC00011$" LIKE 'S%')
   7 - access("H"."PARENT_OID"=SYS_OP_ATG(VALUE(KOKBF$),3,4,2) AND
              "H"."NAME"=SYS_OP_ATG(VALUE(KOKBF$),2,3,2))
   8 - access("R2"."SYS_NC_OID$"=SYS_OP_ATG(VALUE(KOKBF$),3,4,2))

Note
-----
   - warning: inconsistencies found in estimated optimizer costs

27 rows selected.
```

How Documents are Stored in Oracle XML DB Repository

Oracle XML DB provides special handling for XML documents. The rules for storing the contents of schema-based XML document are defined by the XML schema. The content of the document is stored in the default table associated with the global element definition.

Oracle XML DB repository also stores the content of non-XML files, such as JPEG images or Word documents. The XML schema for each resource defines which elements are allowed and specifies whether the content of these files is to be stored as BLOBs or CLOBs. The contents of non-schema-based XML documents are stored as a CLOB in the repository.

There is one resource and one link-properties document for every file or folder in Oracle XML DB repository. If there are multiple access paths to a given document there will be a link-properties document for each possible link. Both the resource document and the link-properties are stored as XML documents. All these documents are stored in tables in Oracle XML DB repository.

When an XML file is loaded into Oracle XML DB repository the following sequence of events that takes place:

1. Oracle XML DB examines the root element of the XML document to see if it is associated with a known (registered) XML schema. This involves looking to see if

the document includes a namespace declaration for the `XMLSchema-instance` namespace, and then looking for a `schemaLocation` or `noNamespaceSchemaLocation` attribute that identifies which XML schema the document is associated with.

2. If the document is based on a known XML schema, then the metadata for the XML schema is loaded from the XML schema cache.
3. The XML document is parsed and decomposed into a set the SQL objects derived from the XML schema.
4. The SQL objects created from the XML file are stored in the default table defined when the XML schema was registered with the database.
5. A resource document is created for each document processed. This allows the content of the document to be accessed using Oracle XML DB repository. The resource document for a schema-based `XMLType` includes an element `XMLRef`. This contents of this element is a REF of `XMLType` that can be used to locate the row in the default table containing the content associated with the resource.

Viewing Relational Data as XML From a Browser

The HTTP server built into Oracle XML DB makes it possible to use a browser to access any document stored in the Oracle XML DB repository. Since a resource can include a REF to a row in an `XMLType` table or view it is possible to use path-based access to access this type of content.

Using DBUri Servlet to Access Any Table or View From a Browser

Oracle XML DB includes the `DBUri` servlet that makes it possible to access the content of any table or view directly from a browser. `DBUri` servlet uses the facilities of the `DBUriType` to generate a simple XML document from the contents of the table. The servlet is C- based and installed in the Oracle XML DB HTTP server. By default the servlet is installed under the virtual directory `/oradb`.

The URL passed to the `DBUri` Servlet is an extension of the URL passed to the `DBUriType`. The URL is simply extended with the address and port number of the Oracle XML DB HTTP server and the virtual root that directs HTTP requests to the `DBUri` servlet. The default configuration for this is `/oradb`.

This means that the URL: `http://localhost:8080/oradb/HR/DEPTARTMENTS`,

would return an XML document containing the contents of the `DEPARTMENTS` table in the HR database schema, assuming that the Oracle XML DB HTTP server is running on port 8080, the virtual root for the `DBUri servlet` is `/oradb`, and that the user making the request has access to the `HR` database schema.

`DBUri` servlet accepts parameters that allow you to specify the name of the `ROW` tag and MIME-type of the document that is returned to the client.

Content in `XMLType` table or view can also be accessed through the `DBUri servlet`. When the URL passed to the `DBUri servlet` references an `XMLType` table or `XMLType` view the URL can be extended with an XPath expression that can determine which documents in the table or row are returned. The XPath expression appended to the URL can reference any node in the document.

XML generated by DBUri servlet can be transformed using the XSLT processor built into Oracle XML DB. This allows XML generated by `DBUri servlet` to be presented in a more legible format such as HTML.

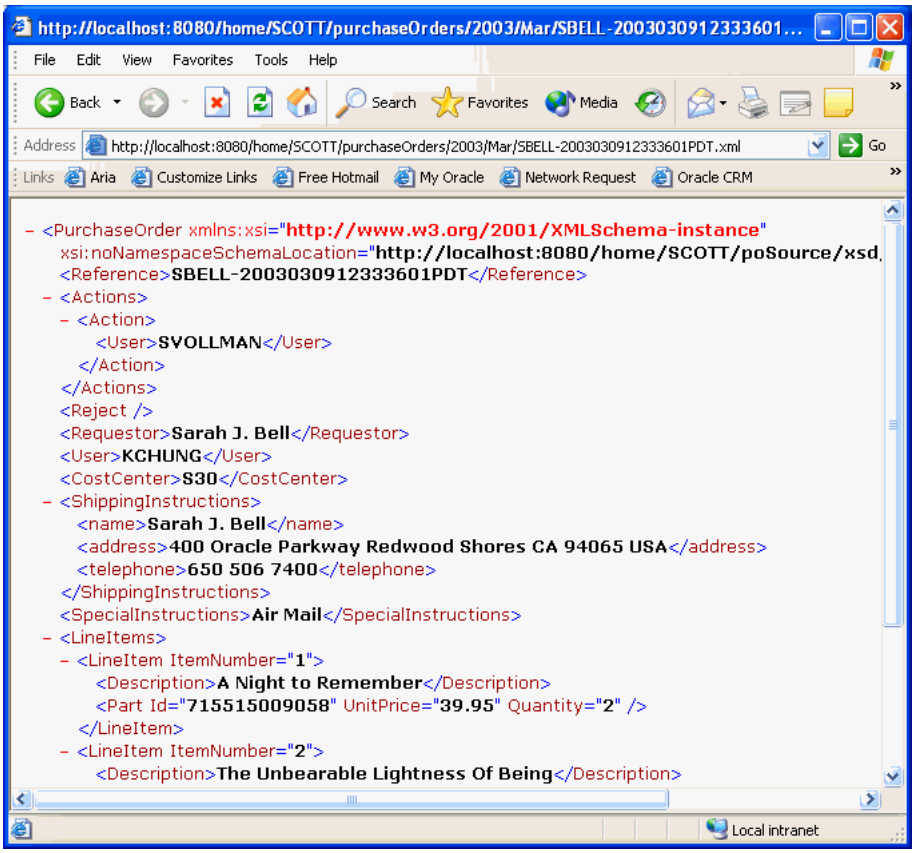
See Also: Chapter 17, "Creating and Accessing Data Through URLs" under "Turning a URL into a Database Query with DBUri Servlet" on page 17-25

Style-sheet processing is initiated by specifying a transform parameter as part of the URL passed to `DBUri servlet`. The style sheet is specified using a URI that references the location of the style sheet within database. The URI can either be a `DBUriType` value that identifies a `XMLType` column in a table or view, or a path to a document stored in the Oracle XML DB repository. The style sheet is applied directly to the generated XML before it is returned to the client. When using DBUri servlet for XSLT processing it is good practice to use the `contenttype` parameter to explicitly specify the MIME type of the generated output.

If the XML document being transformed is stored as schema-based `XMLType`, then Oracle XML DB can reduce the overhead associated with XSL transformation by leveraging the capabilities of the lazily loaded virtual DOM.

Example 3–7 shows how `DBUri` can access a row in the `PURCHASEORDER` table.

Figure 3–7 Using DBUri Servlet to Access XML Content



Note that the root of the URL is `/oradb`. This means that the URL will be passed to the DBUri servlet that accesses the `PURCHASEORDER` table in the `SCOTT` database schema, rather than as a resource in Oracle XML DB repository. The URL includes an XPath expression that restricts the result set to those documents where node `/PurchaseOrder/Reference/text()` contains the value specified in the predicate. The `contenttype` parameter sets the MIME type of the generated document to `text/xml`.

XSL Transformation Using DBUri Servlet

Figure 3–8 shows how an XSL transformation can be applied to XML content generated by the DBUri servlet. In this example the URL passed to the DBUri includes the transform parameter. This causes the DBUri servlet to use the `XMLTransform()` function to apply the style sheet `/home/SCOTT/xsl/purchaseOrder.xsl` to the PurchaseOrder document identified by the main URL, before returning the document to the browser. This style sheet transforms the XML document to a more user-friendly HTML page. The URL also uses `contentType` parameter to specify that the MIME-type of the final document will be `text/html`.

Figure 3–8 Database XSL Transformation of a PurchaseOrder Using DBUri Servlet

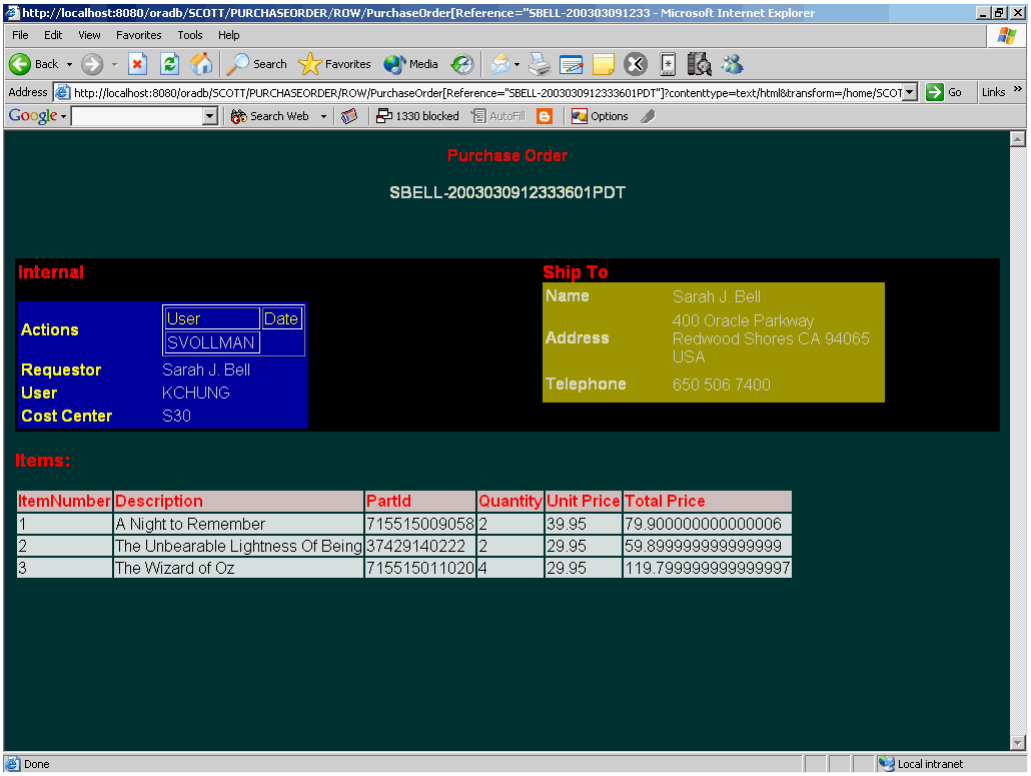
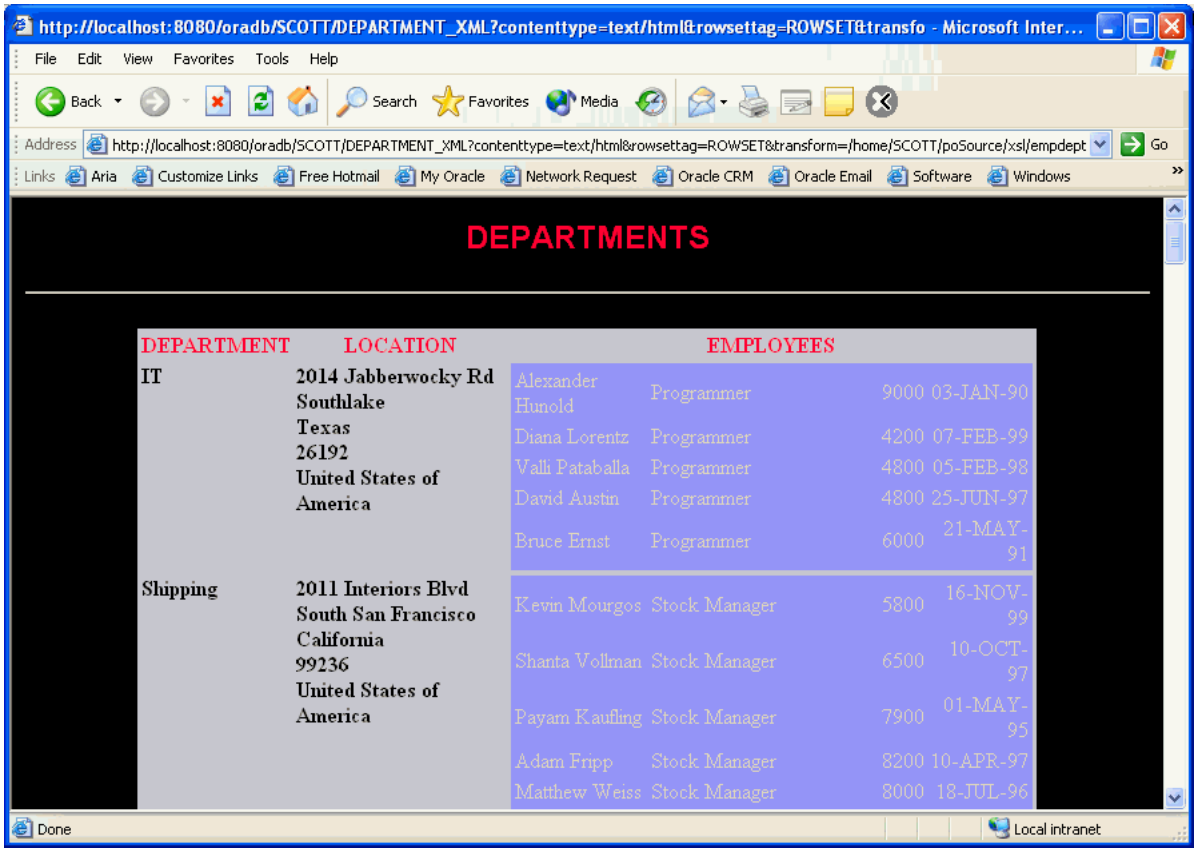


Figure 3–9 shows the `DEPARTMENTS` table displayed as an HTML document. You need no code to achieve this, you only need an `XMLType` view, based on SQL/XML functions, an industry-standard XSL style sheet, and `DBUri` servlet.

Figure 3–9 Database XSL Transformation of Departments Table Using DBUri Servlet



Part II

Storing and Retrieving XML Data in Oracle XML DB

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

4

XMLType Operations

This chapter describes `XMLType` operations and indexing for XML schema-based and non-schema-based applications. It includes guidelines for creating, manipulating, updating, querying, and indexing `XMLType` columns and tables.

This chapter contains these topics:

- Manipulating XML Data With SQL Member Functions
- Selecting and Querying XML Data
- Updating XML Instances and XML Data in Tables
- Indexing XMLType Columns

Note:

- ***Non-schema-based:*** `XMLType` tables and columns described in this chapter are not based on W3C XML Schema 1.0 Recommendation. 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 more storage recommendations.
- ***XML schema-based:*** Appendix B, "XML Schema Primer" and Chapter 5, "XML Schema Storage and Query: The Basics" describe how to work with XML schema-based `XMLType` tables and columns.
- Throughout this chapter, XML schema refers to XML Schema 1.0 recommendation. See also:

  `http://www.w3.org/XML/Schema`

Manipulating XML Data With SQL Member Functions

SQL functions such as `existsNode()`, `extract()`, `XMLTransform()`, and `updateXML()` operate on XML data inside SQL. `XMLType` datatype supports most of these as member functions.

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 4-26 and Chapter 9, "Full Text Search Over XML".

Searching XML Documents With XPath Expressions

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 that walk the tree of nodes and also apply predicates and node test functions. Applying an XPath expression to an XML document can result in a set of nodes. For example, /PO/PONO selects all PONO child elements under the PO root element of the document.

See Also: Appendix C, "XPath and Namespace Primer"

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

Table 4–1 Common XPath Constructs

XPath Construct	Description
/	Denotes the root of the tree in an XPath expression. For example, /PO refers to the child of the root node whose name is PO.
/	Also used as a path separator to identify the children node of any given node. For example, /PurchaseOrder/Reference identifies the purchase order name element Reference, a child of the root element.
//	Used to identify all descendants of the current node. For example, PurchaseOrder//ShippingInstructions matches any ShippingInstructions element under the PurchaseOrder element.
*	Used as a wildcard to match any child node. For example, /PO/*/STREET matches any street element that is a grandchild of the PO element.
[ ]	Used to denote predicate expressions. XPath supports a rich list of binary operators such as OR, AND, and NOT. For example, /PO[PONO=20 and PNAME="PO_2"]/SHIPADDR 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 to denote an index into a list. For example, /PO/PONO[2] identifies the second purchase order number element under the PO root element.
Functions	XPath supports a set of built-in functions such as substring(), round(), and not(). In addition, XPath allows extension functions through the use of namespaces. In the Oracle namespace, http://xmlns.oracle.com/xdb, XML DB additionally supports the function ora:contains(). This functions behave just like the equivalent SQL function.

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

Oracle Extension XPath Function Support

Oracle supports the XPath extension function ora:contains(). This function provides text searching functionality with XPath.

See Also: Chapter 9, "Full Text Search Over XML"

Selecting XML Data Using XMLType Member Functions

You can select XMLType data using PL/SQL, C, or Java. You can also use the SQL functions getClobVal(), getStringVal(), getNumberVal(), or

`getBlobVal(csid)` functions to retrieve XML as a `CLOB`, `VARCHAR`, `NUMBER`, or `BLOB`, respectively.

Example 4–1 Selecting XMLType Columns using getClobVal()

This example shows how to select an `XMLType` column using `getClobVal()` in SQL*Plus:

```
set long 500
set pagesize 100
set linesize 132
--
create table XML_TABLE of XMLType;

Table created.

--
create table TABLE_WITH_XML_COLUMN
(

  FILENAME     varchar2(64),
  XML_DOCUMENT XMLType
);

Table created.

--
INSERT INTO XML_TABLE
       VALUES
       (
 xmltype
 (
    bfilename('XMLDIR','purchaseOrder.xml'),
    nls_charset_id('AL32UTF8')
 )
       );

1 row created.

--
INSERT INTO TABLE_WITH_XML_COLUMN (FILENAME, XML_DOCUMENT)
       VALUES
       (
 'purchaseOrder.xml',
 xmltype
 (
    bfilename('XMLDIR','purchaseOrder.xml'),
    nls_charset_id('AL32UTF8')
 )
       );

1 row created.

--
select x.object_value.getCLOBVal()
from XML_TABLE x;

X.OBJECT_VALUE.GETCLOBVAL()
--------------------------------------------------------------------------------
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNames
paceSchemaLocation="http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.
```

```
xsd">
  <Reference>SBELL-20021009123336O1PDT</Reference>
  <Actions>
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Sarah J. Bell</Requestor>
  <User>SBELL</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Sarah J. Bell</name>
    <address>400 Oracle Parkway
Redwood Shor


1 row selected.

--
select x.XML_DOCUMENT.getCLOBVal()
from TABLE_WITH_XML_COLUMN x;

X.XML_DOCUMENT.GETCLOBVAL()
--------------------------------------------------------------------------------
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNames
paceSchemaLocation="http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.
xsd">
  <Reference>SBELL-20021009123336O1PDT</Reference>
  <Actions>
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Sarah J. Bell</Requestor>
  <User>SBELL</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Sarah J. Bell</name>
    <address>400 Oracle Parkway
Redwood Shor


1 row selected.
```

Querying XML Data Using XMLType Functions

You can query `XMLType` data and extract portions of it using the `existsNode()`, `extract()`, or `extractValue()` functions. These functions use a subset of the W3C XPath recommendation to navigate the document.

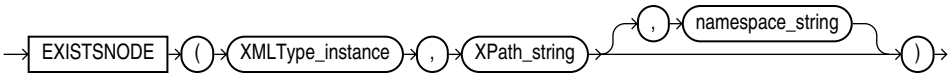
- existsNode() XMLType Function
- extract() XMLType Function
- extractValue() XMLType Function

existsNode() XMLType Function

Figure 4–1 and the following describes the syntax for the `existsNode()` `XMLType` function:

```
existsNode(XMLType_instance IN XMLType,
   XPath_string IN VARCHAR2, namespace_string IN varchar2 := null) RETURN NUMBER
```

Figure 4–1 existsNode() Syntax



The `existsNode()` `XMLType` function 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. The `namespace` parameter can be used to identify the mapping of prefix(es) specified in the `XPath_string` to the corresponding namespace(s).

Example 4–2 Using existsNode() on XMLType

The following example demonstrates how to use `existsNode()` on an `XMLType` instance in a query.

```
SELECT extract(object_value,'/PurchaseOrder/Reference') "REFERENCE"
  FROM PURCHASEORDER
 WHERE
   existsNode(object_value,'/PurchaseOrder[SpecialInstructions="Expidite"]') = 1
```

An XPath expression such as `/PurchaseOrder/Reference` results in a single node. Therefore, `existsNode()` will return 1 for that XPath. This is the same with `/PurchaseOrder/Reference/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.

The following example uses `existsNode()` to select rows with `SpecialInstructions` set to `Expedite`.

> **Note:** When using the `existsNode()` function in a query, always specify `existsNode()` in the `WHERE` clause as shown in this example, never in the `SELECT` list.

Example 4–3 Using existsNode() to Find a node

```
SELECT object_value
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[SpecialInstructions="Expidite"]') =
1;

OBJECT_VALUE
--------------------------------------------------------------------------------
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

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

13 rows selected.
```

Using Indexes to Evaluate existsNode()

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

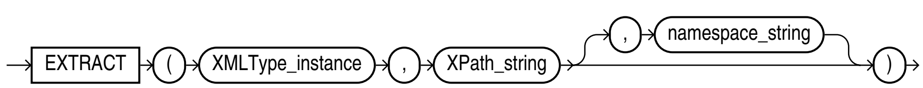
See Also: "Creating CTXXPATH Indexes" on page 4-33

extract() XMLType Function

The `extract()` `XMLType` 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 4–2 and follows:

```
extract(XMLType_instance IN XMLType, XPath_string IN VARCHAR2,
        namespace_string In varchar2 := null) RETURN XMLType;
```

Figure 4–2 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. If multiple text nodes are referenced in the XPath, the 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. You can use the `getStringVal()` or `getNumberVal()` methods on `XMLType` to extract the scalar data.

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

Note: A text node is considered an XMLType. In other words,

```
extract(object_value,
        '/PurchaseOrder/Reference/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:

- /PurchaseOrder/Reference identifies the fragment <Reference> ... </Reference>
- /PurchaseOrder/Reference/text() identifies the value of the text node of the Reference element.

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:

```
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.xsd">
  <Reference>SBELL-20021000912333601PDT</Reference>
  <Actions>
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Sarah J. Bell</Requestor>
  <User>SBELL</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Sarah J. Bell</name>
    <address>400 Oracle Parkway
      Redwood Shores
      CA
      94065
      USA</address>
    <telephone>650 506 7400</telephone>
  </ShippingInstructions>
  <SpecialInstructions>Air Mail</SpecialInstructions>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="2">
      <Description>The Unbearable Lightness Of Being</Description>
      <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>Sisters</Description>
      <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
    </LineItem>
  </LineItems>
```

```
</PurchaseOrder>
```

then you can use:

- `//LineItem[1]` to identify the first `LineItem` element.
- `//LineItem[2]` to identify the second `LineItem` element.

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

To summarize, the `extract()` member function can be used in a number of ways. For example, to extract:

- Numerical values on which function-based indexes can be created to speed up processing
- Collection expressions for use in the `FROM` clause of SQL statements
- Fragments for later aggregation to produce different documents

This example extracts the value of node, `/Warehouse/Docks`, of column, `warehouse_spec` in table `oe.warehouses`:

The following example uses `extract()` to query the value of the `Reference` column for orders with `SpecialInstructions` set to `Expedite`.

Example 4–4 Using extract() to Extract the Value of a Node

```
SELECT extract(object_value,'/PurchaseOrder/Reference') "REFERENCE"
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[SpecialInstructions="Expidite"]') =
1;

REFERENCE
--------------------------------------------------------------
<Reference>AMCEWEN-20021009123336271PDT</Reference>
<Reference>SKING-20021009123336321PDT</Reference>
<Reference>AWALSH-20021009123337303PDT</Reference>
<Reference>JCHEN-20021009123337123PDT</Reference>
<Reference>AWALSH-20021009123336642PDT</Reference>
<Reference>SKING-20021009123336622PDT</Reference>
<Reference>SKING-20021009123336822PDT</Reference>
<Reference>AWALSH-20021009123336101PDT</Reference>
<Reference>WSMITH-20021009123336412PDT</Reference>
<Reference>AWALSH-20021009123337954PDT</Reference>
<Reference>SKING-20021009123338294PDT</Reference>
<Reference>WSMITH-20021009123338154PDT</Reference>
<Reference>TFOX-20021009123337463PDT</Reference>

13 rows selected.
```

Note: Functions `extract().getStringVal()` and `extractValue()` differ in their treatment of entity encoding. Function `extractValue()` unescapes any encoded entities; `extract().getStringVal()` returns the data with entity encoding intact.

extractValue() XMLType Function

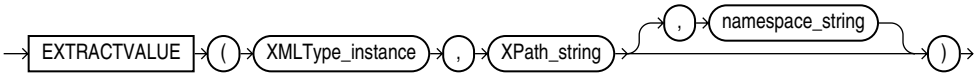
The `extractvalue()` `XMLType` function takes as arguments an `XMLType` instance and an XPath expression. It returns a scalar value corresponding to the result of the XPath evaluation on the `XMLType` instance. Figure 4–3 describes the `extractValue()` syntax.

- **XML schema-based documents**. For documents based on XML schema, if Oracle Database 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.** If the `extractValue()` query can potentially be re-written, such as when the query is over a SQL/XML view, then a scalar value of the appropriate type is returned. Otherwise, the result is of type `VARCHAR2`.

The `extractValue()` function attempts to determine the proper return type from the XML schema of the document, or from other information such as the SQL/XML view. If the proper return type cannot be determined, then Oracle XML DB returns a VARCHAR2. With XML schema-based content, `extractValue()` returns the underlying datatype in most cases. For CLOB datatypes, it will return the CLOB directly.

If a specific datatype is desired, conversion functions such as `to_char` or `to_date` can be put around the `extractValue()` function call or around an `extract.getStringVal()`. This can help maintain consistency between different queries regardless of whether the queries can be rewritten.

Figure 4–3 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|number)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()` has the same syntax 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 value is bigger than 4K, a runtime error occurs.
- 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 extractValue() is on top of a SQL/XML view and the type can be determined at compile time, the appropriate type is returned.
- If the XPath identifies a node, then it automatically gets the scalar content from its text child. The node must have exactly one text child. For example:

  ```
  extractValue(xmlinstance, '/PurchaseOrder/Reference')
  ```

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

  ```
  extract(xmlinstance, '/PurchaseOrder/Reference/text()').getstringval()
  ```

Example 4–5 demonstrates usage of the `extractValue()` function. This query extracts the scalar value of the Reference column. This is in contrast to the `extract()` function shown in Example 4–4 where the entire `<Reference>` element is extracted.

Example 4–5 Extracting the Scalar Value of an XML Fragment Using extractValue()

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference') "REFERENCE"
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[SpecialInstructions="Expidite"]') =
1;

REFERENCE
--------------------------------------------------------------
AMCEWEN-20021009123336271PDT
SKING-20021009123336321PDT
AWALSH-20021009123337303PDT
JCHEN-20021009123337123PDT
AWALSH-20021009123336642PDT
SKING-20021009123336622PDT
SKING-20021009123336822PDT
AWALSH-20021009123336101PDT
WSMITH-20021009123336412PDT
AWALSH-20021009123337954PDT
SKING-20021009123338294PDT
WSMITH-20021009123338154PDT
TFOX-20021009123337463PDT

13 rows selected.
```

> **Note:** Functions `extract().getStringVal()` and `extractValue()` differ in their treatment of entity encoding. Function `extractValue()` unescapes any encoded entities; `extract().getStringVal()` returns the data with entity encoding intact.

Querying XML Data With SQL

The following examples illustrate ways you can query XML data with SQL.

Example 4–6 inserts two rows into the PURCHASEORDER table and performs a query of data in those rows using `extractValue()`.

Example 4–6 Querying XMLType Using extractValue() and existsNode()

```
INSERT INTO PURCHASEORDER
       VALUES
       (
         xmltype
         (
            bfilename('XMLDIR','SMCCAIN-2002091213000000PDT.xml'),
            nls_charset_id('AL32UTF8')
         )
       );

1 row created.

--
INSERT INTO PURCHASEORDER
       VALUES
       (
         xmltype
         (
            bfilename('XMLDIR','VJONES-20020916140000000PDT.xml'),
            nls_charset_id('AL32UTF8')
         )
       );

1 row created.

--
column REFERENCE   format A32
column USERID      format A8
column STATUS      format A8
column STATUS_DATE format A12
set LINESIZE 132
--
SELECT extractValue(object_value,'/PurchaseOrder/Reference') REFERENCE,
       extractValue(object_value,'/PurchaseOrder/*//User') USERID,
       case
         when existsNode(object_value,'/PurchaseOrder/Reject/Date') = 1
           then 'Rejected'
           else 'Accepted'
       end "STATUS",
       extractValue(object_value,'//Date') STATUS_DATE
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'//Date') = 1
 ORDER By extractValue(object_value,'//Date');

REFERENCE                       USERID   STATUS   STATUS_DATE
------------------------------- -------- -------- ------------
VJONES-20020916140000000PDT     SVOLLMAN Accepted 2002-10-11
SMCCAIN-2002091213000000PDT     SKING    Rejected 2002-10-12

2 rows selected.
```

Example 4–7 demonstrates using a cursor in PL/SQL to query XML data. A local `XMLType` instance is used to store transient data.

Example 4–7 Querying Transient XMLType Data

```
  declare
  xNode          XMLType;
  vText          VARCHAR2(256);
  vReference     VARCHAR2(32);

  cursor getPurchaseOrder (REFERENCE in VARCHAR2) is
     SELECT object_value XML
     FROM   PURCHASEORDER
     WHERE
     EXISTSNODE(object_value,'/PurchaseOrder[Reference="'|| REFERENCE || '"]')
                  = 1;

begin
    vReference := 'EABEL-20021009123335791PDT';
    FOR c IN getPurchaseOrder(vReference)
    LOOP
      xNode := c.XML.extract('//Requestor');
      vText := xNode.extract('//text()').getStringVal();
      dbms_output.put_line(' The Requestor for Reference ' || vReference ||
         ' is '|| vText);
    END LOOP;

    vReference := 'PTUCKER-20021009123335430PDT';
    FOR c IN getPurchaseOrder(vReference)
    LOOP
      xNode := c.XML.extract('//LineItem[@ItemNumber="1"]/Description');
      vText := xNode.extract('//text()').getStringVal();
      dbms_output.put_line(' The Description of LineItem[1] for Reference '
        || vReference || ' is '|| vText);
    END LOOP;
end;
/

The Requestor for Reference EABEL-20021009123335791PDT is Ellen S. Abel
The Description of LineItem[1] for Reference PTUCKER-20021009123335430PDT is
Picnic at Hanging Rock

PL/SQL procedure successfully completed.
```

Example 4–8 shows how to extract data from an XML purchase order and insert it into a SQL relational table using the `extract()` function.

Example 4–8 Extracting Data From an XML Document and Inserting It Into a Table

```
create table PURCHASEORDER_TABLE
(

 REFERENCE                                VARCHAR2(28) PRIMARY KEY,
 REQUESTER                                VARCHAR2(48),
 ACTIONS                                  XMLTYPE,
 USERID                                   VARCHAR2(32),
 COSTCENTER                               VARCHAR2(3),
 SHIPTONAME                               VARCHAR2(48),
 ADDRESS                                  VARCHAR2(512),
 PHONE                                    VARCHAR2(32),
 REJECTEDBY                               VARCHAR2(32),
 DATEREJECTED                             DATE,
 COMMENTS                                 VARCHAR2(2048),
 SPECIALINSTRUCTIONS                      VARCHAR2(2048)
```

```
);

Table created.

create table PURCHASEORDER_LINEITEM
(

 REFERENCE,
 FOREIGN KEY ("REFERENCE")                          REFERENCES "PURCHASEORDER_TABLE" ("REFERENCE") ON DELETE
CASCADE,
 LINENO                                             NUMBER(10),
 PRIMARY KEY ("REFERENCE","LINENO"),
 UPC                                                VARCHAR2(14),
 DESCRIPTION                                        VARCHAR2(128),
 QUANTITY                                           NUMBER(10),
 UNITPRICE                                          NUMBER(12,2)
);

Table created.

insert into PURCHASEORDER_TABLE
(

  REFERENCE,
  REQUESTER,
  ACTIONS,
  USERID,
  COSTCENTER,
  SHIPTONAME,
  ADDRESS,
  PHONE,
  REJECTEDBY,
  DATEREJECTED,
  COMMENTS,
  SPECIALINSTRUCTIONS
)

select x.object_value.extract('/PurchaseOrder/Reference/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/Requestor/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/Actions'),
       x.object_value.extract('/PurchaseOrder/User/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/CostCenter/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/ShippingInstructions/name/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/ShippingInstructions/address/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/ShippingInstructions/telephone/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/Rejection/User/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/Rejection/Date/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/Rejection/Comments/text()').getStringVal(),
       x.object_value.extract('/PurchaseOrder/SpecialInstructions/text()').getStringVal()
  from PURCHASEORDER x
 where x.object_value.existsNode('/PurchaseOrder[Reference="EABEL-20021009123336251PDT"]') = 1;

1 row created.

insert into PURCHASEORDER_LINEITEM
(

  REFERENCE,
  LINENO,
  UPC,
  DESCRIPTION,
  QUANTITY,
  UNITPRICE
)

select x.object_value.extract('/PurchaseOrder/Reference/text()').getStringVal(),
```

```
       value(l).extract('/LineItem/@ItemNumber').getNumberVal(),
       value(l).extract('/LineItem/Part/@Id').getNumberVal(),
       value(l).extract('/LineItem/Description/text()').getStringVal(),
       value(l).extract('/LineItem/Part/@Quantity').getNumberVal(),
       value(l).extract('/LineItem/Part/@UnitPrice').getNumberVal()
  from PURCHASEORDER x,
 table (xmlsequence(value(x).extract('/PurchaseOrder/LineItems/LineItem'))) l
 where existsNode(object_value,'/PurchaseOrder[Reference="EABEL-20021009123336251PDT"]') = 1;

3 rows created.

set linesize 132
column USERID format A8
column SPECIALINSTRUCTIONS format A32
column DESCRIPTION format A34
--
select REFERENCE, USERID, SHIPTONAME, SPECIALINSTRUCTIONS
from PURCHASEORDER_TABLE;

REFERENCE                       USERID   SHIPTONAME                                      SPECIALINSTRUCTIONS
------------------------------- -------- ----------------------------------------------- ------------------
EABEL-20021009123336251PDT      EABEL    Ellen S. Abel                                   Counter to Counter

1 row selected.

select REFERENCE, LINENO, UPC, DESCRIPTION, QUANTITY
  from PURCHASEORDER_LINEITEM;

REFERENCE                          LINENO UPC            DESCRIPTION                      QUANTITY
------------------------------- ---------- -------------- -------------------------------- ----------
EABEL-20021009123336251PDT              1 37429125526    Samurai 2: Duel at Ichijoji Temple          3
EABEL-20021009123336251PDT              2 37429128220    The Red Shoes                               4
EABEL-20021009123336251PDT              3 715515009058   A Night to Remember                         1

3 rows selected.
```

Note: PNAME is NULL, because 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 4–9 shows how to extract data from an XML purchase order and insert it into a SQL relational table using the `extractValue()` function.

Example 4–9 Extracting Data from an XML Document and Inserting It Into a Table Using extractValue()

```
create or replace procedure InsertPurchaseOrder(PurchaseOrder xmltype)
as
        REFERENCE             VARCHAR2(28);
begin

       insert into PURCHASEORDER_TABLE
       (
              REFERENCE,
              REQUESTER,
              ACTIONS,
              USERID,
              COSTCENTER,
              SHIPTONAME,
              ADDRESS,
              PHONE,
              REJECTEDBY,
```

```
                DATEREJECTED,
                COMMENTS,
                SPECIALINSTRUCTIONS
        )
        values
        (
                extractValue(PurchaseOrder,'/PurchaseOrder/Reference'),
                extractValue(PurchaseOrder,'/PurchaseOrder/Requestor'),
                extract(PurchaseOrder,'/PurchaseOrder/Actions'),
                extractValue(PurchaseOrder,'/PurchaseOrder/User'),
                extractValue(PurchaseOrder,'/PurchaseOrder/CostCenter'),
                extractValue(PurchaseOrder,'/PurchaseOrder/ShippingInstructions/name'),
                extractValue(PurchaseOrder,'/PurchaseOrder/ShippingInstructions/address'),
                extractValue(PurchaseOrder,'/PurchaseOrder/ShippingInstructions/telephone'),
                extractValue(PurchaseOrder,'/PurchaseOrder/Rejection/User'),
                extractValue(PurchaseOrder,'/PurchaseOrder/Rejection/Date'),
                extractValue(PurchaseOrder,'/PurchaseOrder/Rejection/Comments'),
                extractValue(PurchaseOrder,'/PurchaseOrder/SpecialInstructions')
        )
        returning REFERENCE
        into      REFERENCE;

        insert into PURCHASEORDER_LINEITEM
        (
                REFERENCE,
                LINENO,
                UPC,
                DESCRIPTION,
                QUANTITY,
                UNITPRICE
        )
        select  REFERENCE,
                extractValue(value(l),'/LineItem/@ItemNumber'),
                extractValue(value(l),'/LineItem/Part/@Id'),
                extractValue(value(l),'/LineItem/Description'),
                extractValue(value(l),'/LineItem/Part/@Quantity'),
                extractValue(value(l),'/LineItem/Part/@UnitPrice')
           from table(xmlsequence(extract(PurchaseOrder,'/PurchaseOrder/LineItems/LineItem'))) l;
end;
/
Procedure created.

call insertPurchaseOrder(xmltype(bfilename('XMLDIR','purchaseOrder.xml'),nls_charset_id('AL32UTF8')));

Call completed.

set linesize 132
column USERID format A8
column SPECIALINSTRUCTIONS format A32
column DESCRIPTION format A34
--
select REFERENCE, USERID, SHIPTONAME, SPECIALINSTRUCTIONS
from PURCHASEORDER_TABLE;

REFERENCE                       USERID   SHIPTONAME                                          SPECIALINSTRUCTIONS
------------------------------- -------- --------------------------------------------------- -------------------
SBELL-20021009123336O1PDT       SBELL    Sarah J. Bell                                       Air Mail

1 row selected.

--
select REFERENCE, LINENO, UPC, DESCRIPTION, QUANTITY
  from PURCHASEORDER_LINEITEM;

REFERENCE                           LINENO UPC            DESCRIPTION                         QUANTITY
------------------------------- ---------- -------------- ---------------------------------- ----------
```

```
SBELL-20021009123336O1PDT                1 715515009058   A Night to Remember                        2
SBELL-20021009123336O1PDT                2 37429140222    The Unbearable Lightness Of Being          2
SBELL-20021009123336O1PDT                3 715515011020   Sisters                                    4

3 rows selected.
```

Example 4–10 demonstrates some operations you can perform using the `extract()` and `existsNode()` functions. This example extracts the purchase order name from the purchase order element `PurchaseOrder`, for customers with "`ll`" (double L) in their names and the word "`Shores`" in the shipping instructions.

Example 4–10 Searching XML Data with extract() and existsNode()

```
SELECT p.object_value.extract('/PurchaseOrder/Requestor/text()').getStringVal() NAME,
count(*)
  FROM PURCHASEORDER p
 WHERE p.object_value.existsNode
        (
           '/PurchaseOrder/ShippingInstructions[ora:contains(address/text(),"Shores")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb'
        ) = 1
   AND p.object_value.extract('/PurchaseOrder/Requestor/text()').getStringVal() like '%ll%'
 GROUP BY p.object_value.extract('/PurchaseOrder/Requestor/text()').getStringVal();

NAME                   COUNT(*)
-------------------- ----------
Allan D. McEwen               9
Ellen S. Abel                 4
Sarah J. Bell                13
William M. Smith              7

4 rows selected.
```

Example 4–11 shows the proceeding query rewritten using the `extractValue()` function.

Example 4–11 Searching XML Data with extractValue()

```
SELECT extractValue(object_value,'/PurchaseOrder/Requestor') NAME, count(*)
  FROM PURCHASEORDER p
 WHERE existsNode
       (
         object_value,
         '/PurchaseOrder/ShippingInstructions[ora:contains(address/text(),"Shores")>0]',
         'xmlns:ora="http://xmlns.oracle.com/xdb'
       ) = 1
   AND extractValue(object_value,'/PurchaseOrder/Requestor/text()') like '%ll%'
 GROUP BY extractValue(object_value,'/PurchaseOrder/Requestor');

NAME                   COUNT(*)
-------------------- ----------
Allan D. McEwen               9
Ellen S. Abel                 4
Sarah J. Bell                13
William M. Smith              7

4 rows selected.
```

Example 4–12 shows usage of the `extract()` function to extract nodes identified by an XPath expression. An `XMLType` instance containing the XML fragment is returned by the `extract()` call. The result may be a set of nodes, a singleton node, or a text value. You can determine whether the result is a fragment using the `isFragment()` function on the `XMLType`.

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 15-41. You can, however, query further on the fragment using the various XMLType functions.

Example 4–12 Extracting Fragments from XMLType Using extract()

```
select extractValue(object_value,'/PurchaseOrder/Reference') REFERENCE,
       count(*)
  from PURCHASEORDER,
 table (xmlsequence(extract(object_value,' //LineItem[Part@Id="37429148327"]'))) l
 where extract(object_value,'/PurchaseOrder/LineItems/LineItem[Part/@Id="37429148327"]').isFragment() = 1
 group by extractValue(object_value,'/PurchaseOrder/Reference')
 order by extractValue(object_value,'/PurchaseOrder/Reference');

REFERENCE                       COUNT(*)
------------------------------ ----------
AWALSH-20021009123337303PDT             1
AWALSH-20021009123337954PDT             1
DAUSTIN-20021009123337553PDT            1
DAUSTIN-20021009123337613PDT            1
LSMITH-2002100912333722PDT              1
LSMITH-20021009123337323PDT             1
PTUCKER-20021009123336291PDT            1
SBELL-2002100912333771PDT               1
SKING-2002100912333560PDT               1
SMCCAIN-20021009123336151PDT            1
SMCCAIN-20021009123336842PDT            1
SMCCAIN-2002100912333894PDT             1
TFOX-2002100912333681PDT                1
TFOX-20021009123337784PDT               3
WSMITH-20021009123335650PDT             1
WSMITH-20021009123336412PDT             1

16 rows selected.
```

Updating XML Instances and XML Data in Tables

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

For CLOB-based storage, an update effectively replaces the whole document. To update the whole XML document use the SQL UPDATE statement. The right hand side of the UPDATE statement SET clause must be an XMLType instance. This can be created using the SQL functions and XML constructors that return an XML instance, or by using the PL/SQL DOM APIs for XMLType or Java DOM API, that change and bind existing XML instances.

Example 4–13 updates an XMLType instance using the UPDATE statement.

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

Example 4–13 Updating XMLType Using the UPDATE Statement

```
select extractValue(object_value,'/PurchaseOrder/Reference') REFERENCE,
       extractValue(value(l),'/LineItem/@ItemNumber') LINENO,
       extractValue(value(l),'/LineItem/Description') DESCRIPTION
```

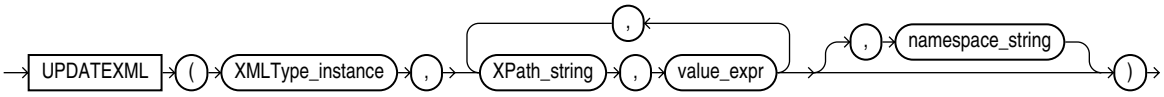
```
  from PURCHASEORDER,
 table (xmlsequence(extract(object_value,'//LineItem'))) l
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="DAUSTIN-20021009123335811PDT"]') = 1
   and ROWNUM < 6;

REFERENCE                      LINENO DESCRIPTION
------------------------------ ------- --------------------------------
DAUSTIN-20021009123335811PDT          1 Nights of Cabiria
DAUSTIN-20021009123335811PDT          2 For All Mankind
DAUSTIN-20021009123335811PDT          3 Dead Ringers
DAUSTIN-20021009123335811PDT          4 Hearts and Minds
DAUSTIN-20021009123335811PDT          5 Rushmore

5 rows selected.

--
UPDATE PURCHASEORDER
   SET object_value = xmltype
                      (
                         bfilename('XMLDIR','NEW-DAUSTIN-20021009123335811PDT.xml'),
                         nls_charset_id('AL32UTF8')
                      )
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="DAUSTIN-20021009123335811PDT"]') = 1;

1 row updated.

--
select extractValue(object_value,'/PurchaseOrder/Reference') REFERENCE,
       extractValue(value(l),'/LineItem/@ItemNumber') LINENO,
       extractValue(value(l),'/LineItem/Description') DESCRIPTION
  from PURCHASEORDER,
 table (xmlsequence(extract(object_value,'//LineItem'))) l
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="DAUSTIN-20021009123335811PDT"]') = 1;

REFERENCE                      LINENO DESCRIPTION
------------------------------ ------- --------------------------------
DAUSTIN-20021009123335811PDT          1 Dead Ringers
DAUSTIN-20021009123335811PDT          2 Getrud
DAUSTIN-20021009123335811PDT          3 Branded to Kill

3 rows selected.
```

updateXML() XMLType Function

updateXML() function takes in a source `XMLType` instance, and a set of XPath value pairs. Figure 4–4 illustrates the `updateXML()` syntax. It returns a new XML instance consisting of the original `XMLType` instance with appropriate XML nodes updated with the given values. The optional `namespace` parameter specifies the namespace mapping of prefix(es) in the XPath parameters.

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

Figure 4–4 updateXML() Syntax



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

```
UPDATEXML(xmlinstance, xpath1, value_expr1
            [, xpath2, value_expr2]...[,namespace_string]);
```

Example 4–14 demonstrates using the `updateXML()` function in the right hand side of an `UPDATE` statement to update the XML document in the table instead of creating a new one. Note that `updateXML()` updates the whole document, not just the part selected.

Example 4–14 Updating XMLType Using UPDATE and updateXML()

```
SELECT extract(object_value,'/PurchaseOrder/Actions/Action[1]') ACTION
FROM PURCHASEORDER
WHERE
existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

ACTION
--------------------------------------------------------------------------------------------------------------
<Action>
  <User>SVOLLMAN</User>
</Action>


1 row selected.

UPDATE PURCHASEORDER
SET object_value = updateXML(object_value,'/PurchaseOrder/Actions/Action[1]/User/text()','SKING')
WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

1 row updated.

SELECT extract(object_value,'/PurchaseOrder/Actions/Action[1]') ACTION
FROM PURCHASEORDER
WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

ACTION
--------------------------------------------------------------------------------------------------------------
<Action>
  <User>SKING</User>
</Action>


1 row selected.
```

Example 4–15 shows how you can update multiple nodes using the `updateXML()` function.

Example 4–15 Updating Multiple Text Nodes and Attribute Values Using updateXML()

```
SELECT extractValue(object_value,'/PurchaseOrder/Requestor') Name,
       extract(object_value,'/PurchaseOrder/LineItems') LINEITEMS
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

NAME             LINEITEMS
---------------- --------------------------------------------------------------------------
Sarah J. Bell    <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>A Night to Remember</Description>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="2">
                     <Description>The Unbearable Lightness Of Being</Description>
                     <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
```

```
                  </LineItem>
                  <LineItem ItemNumber="3">
                    <Description>Sisters</Description>
                    <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                  </LineItem>
                </LineItems>


1 row selected.

UPDATE PURCHASEORDER
   SET object_value = updateXML
                      (
                         object_value,
                         '/PurchaseOrder/Requestor/text()','Stephen G. King',
                         '/PurchaseOrder/LineItems/LineItem[1]/Part/@Id','786936150421',
                         '/PurchaseOrder/LineItems/LineItem[1]/Description/text()','The Rock',
                         '/PurchaseOrder/LineItems/LineItem[3]',
                         XMLType
                         (
                            '<LineItem ItemNumber="99">
                               <Description>Dead Ringers</Description>
                               <Part Id="715515009249" UnitPrice="39.95" Quantity="2"/>
                             </LineItem>'
                          )
                      )
WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123333601PDT"]') = 1;

1 row updated.

SELECT extractValue(object_value,'/PurchaseOrder/Requestor') Name,
       extract(object_value,'/PurchaseOrder/LineItems') LINEITEMS
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123333601PDT"]') = 1;

NAME             LINEITEMS
---------------- --------------------------------------------------------------------------
Stephen G. King  <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>The Rock</Description>
                     <Part Id="786936150421" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="2">
                     <Description>The Unbearable Lightness Of Being</Description>
                     <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="99">
                     <Description>Dead Ringers</Description>
                     <Part Id="715515009249" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                 </LineItems>


1 row selected.
```

Example 4–16 demonstrates how you can use the `updateXML()` function to update selected nodes within a collection.

Example 4–16 Updating Selected Nodes Within a Collection Using updateXML()

```
SELECT extractValue(object_value,'/PurchaseOrder/Requestor') Name,
       extract(object_value,'/PurchaseOrder/LineItems') LINEITEMS
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123333601PDT"]') = 1;

NAME             LINEITEMS
```

```
--------------- --------------------------------------------------------------------------
Sarah J. Bell    <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>A Night to Remember</Description>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="2">
                     <Description>The Unbearable Lightness Of Being</Description>
                     <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="3">
                     <Description>Sisters</Description>
                     <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                   </LineItem>
                 </LineItems>


1 row selected.

UPDATE PURCHASEORDER
   SET object_value = updateXML
                    (
                     object_value,
                     '/PurchaseOrder/Requestor/text()','Stephen G. King',
                     '/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]/@Quantity',25,
                     '/PurchaseOrder/LineItems/LineItem[Description/text()="The Unbearable Lightness Of Being"]',
                      XMLType
                      (
                              '<LineItem ItemNumber="99">
                                <Part Id="786936150421" Quantity="5" UnitPrice="29.95"/>
                                <Description>The Rock</Description>
                               </LineItem>'
                           )
                       )
WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

1 row updated.

SELECT extractValue(object_value,'/PurchaseOrder/Requestor') Name,
       extract(object_value,'/PurchaseOrder/LineItems') LINEITEMS
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

NAME             LINEITEMS
---------------- --------------------------------------------------------------------------
Stephen G. King  <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>A Night to Remember</Description>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity="25"/>
                   </LineItem>
                   <LineItem ItemNumber="99">
                     <Part Id="786936150421" Quantity="5" UnitPrice="29.95"/>
                     <Description>The Rock</Description>
                   </LineItem>
                   <LineItem ItemNumber="3">
                     <Description>Sisters</Description>
                     <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                   </LineItem>
                 </LineItems>

1 row selected.
```

updateXML() and NULL Values

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

If you update the `text` node of an element to `NULL`, Oracle Database removes the text value of the element, and the element itself remains, but is empty.

Example 4–17 updates the Description element, Quantity element, and the `text()` node for the Quantity element to `NULL` using the `updateXML()` function.

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

You cannot use `updateXML()` to remove, add, or delete a particular element or an attribute. To do so, you must update the *containing* element with a new value.

Example 4–17 NULL Updates With updateXML()

```
SELECT extractValue(object_value,'/PurchaseOrder/Requestor') Name,
       extract(object_value,'/PurchaseOrder/LineItems') LINEITEMS
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

NAME            LINEITEMS
--------------- --------------------------------------------------------------------------
Sarah J. Bell   <LineItems>
                  <LineItem ItemNumber="1">
                    <Description>A Night to Remember</Description>
                    <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
                  </LineItem>
                  <LineItem ItemNumber="2">
                    <Description>The Unbearable Lightness Of Being</Description>
                    <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
                  </LineItem>
                  <LineItem ItemNumber="3">
                    <Description>Sisters</Description>
                    <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                  </LineItem>
                </LineItems>


1 row selected.

UPDATE PURCHASEORDER
   SET object_value = updateXML
               (
               object_value,
               '/PurchaseOrder/LineItems/LineItem[Part/@Id="715515009058"]/Description',null,
               '/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]/@Quantity',null,
               '/PurchaseOrder/LineItems/LineItem[Description/text()="The Unbearable Lightness Of Being"]',null
               )
WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

1 row updated.

--
SELECT extractValue(object_value,'/PurchaseOrder/Requestor') Name,
       extract(object_value,'/PurchaseOrder/LineItems') LINEITEMS
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

NAME            LINEITEMS
--------------- --------------------------------------------------------------------------
```

```
Sarah J. Bell    <LineItems>
                   <LineItem ItemNumber="1">
                     <Description/>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity=""/>
                   </LineItem>
                   <LineItem/>
                   <LineItem ItemNumber="3">
                     <Description>Sisters</Description>
                     <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                   </LineItem>
                 </LineItems>

1 row selected.
```

The XPath expressions in the `updateXML()` Statement shown in Example 4–18 are processed by Oracle XML DB and rewritten into the equivalent object relational SQL statement given in Example 4–19.

Example 4–18 XPATH Rewrite with UpdateXML()

```
SELECT extractValue(object_value,'/PurchaseOrder/User')
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVAL
----------
SBELL

1 row selected.

--
UPDATE PURCHASEORDER
   SET object_value = updateXML(object_value,'/PurchaseOrder/User/text()','SVOLLMAN')
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

1 row updated.

--
SELECT extractValue(object_value,'/PurchaseOrder/User')
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVAL
----------
SVOLLMAN

1 row selected.
```

Example 4–19 Rewritten Object Relational Equivalent of XPATH Rewrite with UpdateXML()

```
SELECT extractValue(object_value,'/PurchaseOrder/User')
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVAL
----------
SBELL

1 row selected.

--
UPDATE PURCHASEORDER p
   SET p."XMLDATA"."USERID" = 'SVOLLMAN'
 WHERE p."XMLDATA"."REFERENCE" = 'SBELL-20021009123336O1PDT';
```

```
1 row updated.

--
SELECT extractValue(object_value,'/PurchaseOrder/User')
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVAL
----------
SVOLLMAN

1 row selected.
```

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.

Guidelines For Preserving DOM Fidelity When Using updateXML()

Here are some guidelines for preserving DOM fidelity when using `updateXML()`:

When DOM Fidelity is Preserved

When you update an element to `NULL`, you make that element appear empty in its parent, such as in <myElem/>.

When you update a text node inside an element to `NULL`, you remove that text node from the element.

When you update an attribute node to `NULL`, you make the value of the attribute become the empty string, for example, myAttr="".

When DOM Fidelity is Not Preserved

When you update a complexType element to `NULL`, you make the element appear empty in its parent, for example, `<myElem/>`.

When you update a SQL-inlined simpleType element to `NULL`, you make the element disappear from its parent.

When you update a text node to `NULL`, you are doing the same thing as setting the parent simpleType element to `NULL`. Furthermore, text nodes can appear only inside `simpleTypes` when DOM fidelity is not preserved, since there is no positional descriptor with which to store mixed content.

When you update an attribute node to `NULL`, you remove the attribute from the element.

Optimization of updateXML()

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

met, then the `updateXML()` is rewritten to update the object-relational columns directly with the values.

For example, the XPath expressions in the `updateXML()` statement shown in Example 4–20 are processed by Oracle XML DB and re-written into the equivalent object relational SQL statement shown in Example 4–21.

> **See Also:** Chapter 3, "Using Oracle XML DB" and Chapter 5, "XML Schema Storage and Query: The Basics" for information on the conditions for XPath rewrite.

Example 4–20 XPath expressions in updateXML() Statement

```
SELECT extractValue(object_value,'/PurchaseOrder/User')
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVAL
----------
SBELL

1 row selected.

--
UPDATE PURCHASEORDER
   SET object_value = updateXML(object_value,'/PurchaseOrder/User/text()','SVOLLMAN')
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

1 row updated.

--
SELECT extractValue(object_value,'/PurchaseOrder/User')
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVAL
----------
SVOLLMAN

1 row selected.
```

Example 4–21 Object Relational Equivalent of updateXML() Statement

```
SELECT extractValue(object_value,'/PurchaseOrder/User')
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

EXTRACTVAL
----------
SBELL

1 row selected.

--
UPDATE PURCHASEORDER p
   SET p."XMLDATA"."USERID" = 'SVOLLMAN'
 WHERE p."XMLDATA"."REFERENCE" = 'SBELL-20021009123336O1PDT';

1 row updated.

--
SELECT extractValue(object_value,'/PurchaseOrder/User')
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;
```

```
EXTRACTVAL
----------
SVOLLMAN

1 row selected.
```

Creating Views of XML Data with updateXML()

You can use `updateXML()` to create new views of XML data. Example 4–22 creates a view of the `PURCHASEORDER` table using the `updateXML()` function.

Example 4–22 Creating Views Using updateXML()

```
CREATE OR REPLACE VIEW purchaseorder_summary of XMLType
as
select updateXML
       (
         object_value,
         '/PurchaseOrder/Actions',null,
         '/PurchaseOrder/ShippingInstructions',null,
         '/PurchaseOrder/LineItems',null
       ) as XML
FROM PURCHASEORDER p;

View created.

select object_value
from
PURCHASEORDER_SUMMARY
where existsNode(object_value, '/PurchaseOrder[Reference="DAUSTIN-20021009123335811PDT"]') = 1;

OBJECT_VALUE
--------------------------------------------------------------------------------
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "http://localhost:8080//home/SCOTT/poSource/xsd/purchaseOrder.xsd">
  <Reference>DAUSTIN-20021009123335811PDT</Reference>
  <Actions/>
  <Reject/>
  <Requestor>David L. Austin</Requestor>
  <User>DAUSTIN</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions/>
  <SpecialInstructions>Courier</SpecialInstructions>
  <LineItems/>
</PurchaseOrder>

1 row selected.
```

Indexing XMLType Columns

Chapter 3 provided a basic introduction to creating indexes on XML documents that have been stored using the structured storage option. It demonstrated how to use the `extractValue()` function to create indexes on `XMLType` documents stored in tables or columns that are based on the structured storage option.

This section discusses other indexing techniques including:

- XPATH REWRITE for indexes on Singleton Elements or Attributes
- Creating B-Tree Indexes on the Contents of a Collection

- Creating Function-Based Indexes on XMLType Tables and Columns
- CTXXPath Indexes on XMLType Columns
- Oracle Text Indexes on XMLType Columns

XPATH REWRITE for indexes on Singleton Elements or Attributes

When indexes are created on structured `XMLType` tables or columns, XML DB attempts to re-write the XPath expressions provided to the `extractValue()` function into `CREATE INDEX` statements that operate directly on the underlying objects.

For instance, given an index created as shown in Example 4–23, XPath re-write will re-write the index resulting in the create index statement shown in Example 4–24 being executed. As can be seen, the rewritten index is created directly on the columns that manage the attributes of the underlying SQL objects. This technique works well when the Element or Attribute being indexed only occurs once in the XML Document.

Example 4–23 Using extractValue() to Create an Index on a singleton Element or Attribute

```
CREATE INDEX iPURCHASEORDER_REJECTEDBY
ON PURCHASEORDER
(extractValue(object_value,'/PurchaseOrder/Reject/User'));
```

Example 4–24 XPath Re-write of an Index on a singleton Element or Attribute

```
CREATE INDEX iPURCHASEORDER_REJECTEDBY
ON PURCHASEORDER p
(p."XMLDATA"."REJECTION"."REJECTED_BY");
```

Creating B-Tree Indexes on the Contents of a Collection

You might often need to create an index over nodes that occur more than once in the target document. For instance, assume you wanted to create an index on the Id attribute of the LineItem element. A logical first attempt would be to create an index using the syntax shown in Example 4–25.

Example 4–25 Using extractValue() to Create an Index on a repeating Element or Attributes

```
CREATE INDEX iLINEITEM_UPCCODE
ON PURCHASEORDER
(extractValue(object_value,'/PurchaseOrder/LineItems/LineItem/Part/@Id'));
(extractValue(object_value,'/PurchaseOrder/LineItems/LineItem/Part/@Id'))
                           *
ERROR at line 3:
ORA-19025: EXTRACTVALUE returns value of only one node
```

As can be seen, when the Element or Attribute being indexed occurs multiple times in the document, the create index fails because `extractValue()` is only allowed to return a single value for each row it processes. It is possible to create an Index replacing `extractValue()` with `extract().getStringVal()` as shown in Example 4–26.

Example 4–26 Using extract().getStringVal() to Create a Function-Based Index on an extract()

```
CREATE INDEX iLINEITEM_UPCCODE
ON PURCHASEORDER
( extract(object_value,'PurchaseOrder/LineItems/LineItem/Part/@Id').getStringVal());

Index created.
```

This allows the Create Index statement to succeed. However, the index that is created is not what is expected. The index is created by invoking the `extract()` and `getStringVal()` functions for each row in the table and then indexing the result of the function against the rowid of the row.

The problem with this technique, is that when the XPath expression supplied to the `extract()` function, the `extract()` function can only returns multiple nodes. The result of the `extract()` function is a single `XMLType` consisting of a fragment containing the matching nodes. The result of invoking `getStringVal()` on an `XMLType` that contains a fragment is a concatenation of the nodes in question as shown in Example 4–27.

As can be seen, what is indexed for this row is the concatenation of the 3 UPC codes, not each of the individual UPC codes. In general, care should be taken when creating an index using the `extract()` function. It is unlikely that this index will be useful.

As was shown in Chapter 3, for Schema-Based `XMLType`, the best way to resolve this issue is adopt a storage structure that uses nested tables to force each node that is indexed to be stored as a separate row. The index can then be created directly on the nested table using object relational SQL similar to the SQL that is generated by XPath re-write.

Example 4–27 Problem with using extract().getStringVal() to Create a Function-Based Index on an extract() Function

```
SELECT extract(object_value,'/PurchaseOrder/LineItems') XML,
       extract(object_value,'PurchaseOrder/LineItems/LineItem/Part/@Id').getStringVal()
INDEX_VALUE
  FROM PURCHASEORDER
 WHERE existsNode(object_value,'/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]') = 1;

XML                                                                  INDEX_VALUE
-------------------------------------------------------------------- --------------
<LineItems>                                                          71551500905837
  <LineItem ItemNumber="1">                                          42914022271551
    <Description>A Night to Remember</Description>                   5011020
    <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>The Unbearable Lightness Of Being</Description>
    <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>Sisters</Description>
    <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
  </LineItem>
</LineItems>


1 row selected.
```

Creating Function-Based Indexes on XMLType Tables and Columns

The index that is created in Example 4–26 is an example of a function-based index. A function-based index is created by evaluating the specified functions for each row in the table. In that particular case, the results of the functions were not useful and consequently the index itself was not useful. However, there are many cases were function-based indexes are useful.

One example of when a function-based index is useful is when the XML content is not being managed using structured storage. In this case, instead of the CREATE INDEX statement being re-written, the index will be created by invoking the function on the XML content and indexing the result.

Given the table created in Example 4–28, which uses CLOB storage rather than structured storage to persist the XML, the following CREATE INDEX statement will result in a function-based index being created on the value of the text node belonging to the Reference element. As the example shows, this index will enforce the unique constraint on the value of the text node associated with the Reference element.

Example 4–28 Creating a Function-Based Index on a CLOB-based XMLType()

```
create table PURCHASEORDER_CLOB of XMLTYPE
XMLType store as CLOB
ELEMENT "http://localhost:8080/home/SCOTT/poSource/xsd/purchaseOrder.xsd#PurchaseOrder";

Table created.

--
insert into PURCHASEORDER_CLOB
select object_value from PURCHASEORDER;

134 rows created.

--
create unique index iPURCHASEORDER_REFERENCE
on PURCHASEORDER_CLOB
(extractValue(object_value,'/PurchaseOrder/Reference'));

Index created.

--
insert into PURCHASEORDER_CLOB
       VALUES
       (
         xmltype
         (
            bfilename('XMLDIR','EABEL-20021009123335791PDT.xml'),
            nls_charset_id('AL32UTF8')
         )
       );
insert into PURCHASEORDER_CLOB
*

ERROR at line 1:
ORA-00001: unique constraint (SCOTT.IPURCHASEORDER_REFERENCE) violated
```

One thing to bear in mind when creating and using function-based indexes is that the optimizer will only consider using the index when the function included in the WHERE clause is identical to the function used to create the index.

Consider the queries in Example 4–29 which both find a PurchaseOrder-based value of the text node associated with the Reference element. Note that the first query, which

uses `existsNode()` to locate the document, does not use the index, while the second query, which uses `extractValue()`, does use the index.

Example 4–29 Queries that use Function-Based indexes

```
explain plan for
select object_value
from PURCHASEORDER_CLOB
where existsNode(object_value,'/PurchaseOrder[Reference="EABEL-20021009123335791PDT"') = 1;

Explained.

--
set ECHO OFF

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------
Plan hash value: 3761539978

-------------------------------------------------------------------------------------
| Id  | Operation         | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                    |     2 |  4004 |     3  (34)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| PURCHASEORDER_CLOB |     2 |  4004 |     3  (34)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(EXISTSNODE(SYS_MAKEXML('3A7F7DBBEE5543A486567A908C71D65A',3664,"PU
              RCHASEORDER_CLOB"."XMLDATA"),'/PurchaseOrder[Reference="EABEL-20021009123335791P
              DT"')=1)

15 rows selected.

--
explain plan for
select object_value
from PURCHASEORDER_CLOB
where extractValue(object_value,'/PurchaseOrder/Reference') = 'EABEL-20021009123335791PDT';

Explained.

--
set ECHO OFF

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------------------------
Plan hash value: 1408177405

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                           |     1 |  2002 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER_CLOB        |     1 |  2002 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | IPURCHASEORDER_REFERENCE  |     1 |       |            | 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(EXTRACTVALUE(SYS_MAKEXML('3A7F7DBBEE5543A486567A908C71D65A',3664,"XMLDATA"),'/Purc
              haseOrder/Reference')='EABEL-20021009123335791PDT')

Note
-----
```

```
   - warning: inconsistencies found in estimated optimizer costs
   - dynamic sampling used for this statement

20 rows selected.
```

Function-based indexes can be created on both structured and unstructured schema-based `XMLType` tables and columns as well as non-schema-based `XMLType` tables and columns. If XPath re-write cannot process the XPath expression supplied as part of the Create Index statement, the statement will result in a function-based index being created.

An example of this would be creating an index based on the existsNode() function. The existsNode() function simply returns 1 or 0 depending on whether or not a document contains a node that matches the supplied XPath expression. This means that it is not possible for XPath re-write to generate an equivalent object-relational CREATE INDEX statement. In general, since existsNode() returns 0, or 1, it makes sense to use BITMAP indexes when creating an index based on the existsNode() function.

In Example 4–30, an index is created that can be used to speed up a query that searches for instances of a rejected PurchaseOrder by looking for the presence of a text() node under the element /PurchaseOrder/Reject/User.

Since the index is a function-based index, it can be used with structured and unstructured schema-based `XMLType` tables and columns, as well as non-schema-based `XMLType` tables and columns.

Example 4–30 Creating a Function-Based index on Schema-Based XMLType

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference')
  from PURCHASEORDER
 where existsNode(object_Value,'/PurchaseOrder/Reject/User/text()') = 1;

EXTRACTVALUE(OBJECT_VALUE,'/PU
------------------------------
SMCCAIN-2002091213000000PDT

1 row selected.

--
CREATE BITMAP INDEX iPURCHASEORDER_REJECTED
ON PURCHASEORDER
(existsNode(object_Value,'/PurchaseOrder/Reject/User/text()'));

Index created.

--
call dbms_stats.gather_table_stats(USER,'PURCHASEORDER');

Call completed.

--
explain plan for
SELECT extractValue(object_value,'/PurchaseOrder/Reference')
  from PURCHASEORDER
 where existsNode(object_Value,'/PurchaseOrder/Reject/User/text()') = 1;

Explained.

--
set ECHO OFF

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------------
```

```
Plan hash value: 841749721

-------------------------------------------------------------------------------------
| Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |     1 |   419 |     4   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| PURCHASEORDER |     1 |   419 |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("PURCHASEORDER"."SYS_NC00018$" IS NOT NULL)

13 rows selected.
```

CTXXPath Indexes on XMLType Columns

The indexing techniques outlined earlier in this chapter require you to be aware in advance of the set of XPath expressions that will be used when searching XML content. Oracle XML DB also makes it possible to create a `CTXXPATH` index—a general purpose XPath-based index, based on Oracle Text technology, that can be used to improve the performance of any existsNode() based search. Such an index has the following advantages:

- You do not need prior knowledge of the XPath expressions that will be searched on.
- It can be used with both structured and unstructured schema-based `XMLType` tables and columns, as well as non-schema-based `XMLType` tables and columns.
- It can be used to create indexes that make it improve the performance of searches that involve XPath expressions that target nodes that occur multiple times within a document.

The `CTXXPATH` index is based on Oracle Text Technology and the functionality provided in the HASPATH and INPATH operators provided by the Oracle Text CONTAINS function. The HASPATH and INPATH operators allow high performance XPath-like searches to be performed over XML content. Unfortunately, they do not support true XPath compliant syntax.

The `CTXXPATH` index is designed to re-write the XPath expression supplied to existsNode() into HASPATH and INPATH operators that can use the underlying text index to quickly locate a superset of the documents that match the supplied XPath expression. Each document identified by the text index is then checked, using a DOM-based evaluation, to ensure that it is a true match for the supplied XPath expression. Due to the asynchronous nature of the underlying text technology, the `CTXXPATH` index will also perform a DOM based evaluation of all un-indexed documents to see if they also should be included in the result set.

CTXXPATH Indexing Features

`CTXXPATH` indexing has the following characteristics:

- 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
- `CTXXPATH` index will only work for queries where the XPath expressions that identify the required documents are supplied using an `existsNode()` function that appears in the `WHERE` clause of the SQL statement being executed.

- Only handles a limited set of XPath expressions. See "Choosing the Right Plan: Using CTXXPATH Index in existsNode() Processing" on page 4-35 for the list of XPath expressions not supported by the index.
- Only supports the STORAGE preference parameter. See "Creating CTXXPATH Storage Preferences With CTX_DDL. Statements" on page 4-33.
- Data Manipulation Language (DML) operations such as updating and deleting are asynchronous. You must use a special command to synchronize the DML operations, in a similar fashion to Oracle Text index. Despite the asynchronous nature of DML operations, CTXXPATH indexing still follows the 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

Create CTXXPATH indexes in the same way that you create Oracle Text indexes, using the syntax:

```
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 4–31 demonstrates how to create a CTXXPATH index for XPath searching.

> **See Also:** "existsNode() XMLType Function" on page 4-5 for more information on using `existsNode()`.

Example 4–31 Using CTXXPATH Index and existsNode() for XPath Searching

```
create index PURCHASEORDER_CLOB_XPATH
on PURCHASEORDER_CLOB (object_value)
indextype is CTXSYS.CTXXPATH;
```

Creating CTXXPATH Storage Preferences With CTX_DDL. Statements

The only preference allowed in CTXXPATH indexing is the STORAGE preference. Create the STORAGE preference in the same way that you would for an Oracle Text index as shown in Example 4–32.

> **Note:** You must be granted execute privileges on the CTXSYS.CTX_DLL package in order to create storage preferences.

Example 4–32 Creating and Using Storage Preferences for CTXXPATH Indexes

```
begin
  ctx_ddl.create_preference('CLOB_XPATH_STORE', 'BASIC_STORAGE');
  ctx_ddl.set_attribute('CLOB_XPATH_STORE', 'I_TABLE_CLAUSE',
                        'tablespace USERS storage (initial 1K)');
  ctx_ddl.set_attribute('CLOB_XPATH_STORE', 'K_TABLE_CLAUSE',
                        'tablespace USERS storage (initial 1K)');
  ctx_ddl.set_attribute('CLOB_XPATH_STORE', 'R_TABLE_CLAUSE',
                        'tablespace USERS storage (initial 1K)');
  ctx_ddl.set_attribute('CLOB_XPATH_STORE', 'N_TABLE_CLAUSE',
                        'tablespace USERS storage (initial 1K)');
  ctx_ddl.set_attribute('CLOB_XPATH_STORE', 'I_INDEX_CLAUSE',
```

```
                        'tablespace USERS storage (initial 1K)');
end;
/

PL/SQL procedure successfully completed.

create index PURCHASEORDER_CLOB_XPATH
on PURCHASEORDER_CLOB (object_value)
indextype is CTXSYS.CTXXPATH
PARAMETERS('storage CLOB_XPATH_STORE memory 120M');

Index created.

--
explain plan for
select extractValue(object_value,'/PurchaseOrder/Reference')
  from PURCHASEORDER_CLOB
 where existsNode(object_value,'//LineItem/Part[@Id="715515011624"]') = 1;

Explained.

--

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------
Plan hash value: 2191955729

-------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                          |     1 |  2031 |     4   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER_CLOB       |     1 |  2031 |     4   (0)| 00:00:01 |
|   2 |   DOMAIN INDEX              | PURCHASEORDER_CLOB_XPATH |       |       |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(EXISTSNODE(SYS_MAKEXML('3A7F7DBBEE5543A486567A908C71D65A',3664,"PURCHASEORDER_CLOB
              "."XMLDATA"),'//LineItem/Part[@Id="715515011624"]')=1)

Note
-----
   - dynamic sampling used for this statement

19 rows selected.
```

Performance Tuning a CTXXPATH Index: Synchronizing and Optimizing

Example 4–33 illustrates how to synchronize DML operations using the SYNC_INDEX procedure in the CTX_DDL package.

Example 4–33 Synchronizing the CTXXPATH Index

```
call ctx_ddl.sync_index('PURCHASEORDER_CLOB_XPATH');
```

Example 4–34 illustrates how to optimize the CTXXPATH index using the OPTIMIZE_INDEX procedure in the CTX_DDL package.

Example 4–34 Optimizing the CTXXPATH Index

```
exec ctx_ddl.optimize_index('PURCHASEORDER_CLOB_XPATH', 'FAST');

PL/SQL procedure successfully completed.
```

```
--
exec ctx_ddl.optimize_index('PURCHASEORDER_CLOB_XPATH', 'FULL');

PL/SQL procedure successfully completed.
```

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 for the following reasons:

- Oracle Database cost-based optimizer may decide it is too costly to use CTXXPATH index as a primary filter
- XPath expressions cannot all be handled by CTXXPATH index. The following XPath constructs cannot be handled by CTXXPATH index:
  - XPath functions
  - Numerical range operators
  - Numerical equality
  - Arithmetic operators
  - UNION operator "|"
  - Parent and sibling axes
  - An attribute following a *, //, .. , in other words,'/A/*/@attr', '/A//@attr', '/A//../@attr'
  - '.' or '*' at the end of the path expression
  - A predicate following '.' or '*'
  - String literal equalities are supported with the following restrictions:
    * The left hand side must be a path ('.' by itself is not allowed, for example .="dog")
    * The right hand side must be a literal
  - Anything not expressible by abbreviated syntax is also not supported

For the cost-based optimizer to better estimate the costs and selectivities for the existsNode() function, you must first gather statistics on your CTXXPATH indexing by using the ANALYZE command or DBMS_STATS package as follows:

```
ANALYZE INDEX myPathIndex COMPUTE STATISTICS;
```

or you can simply analyze the whole table:

```
ANALYZE TABLE XMLTab COMPUTE STATISTICS;
```

CTXXPATH Indexes On XML Schema-Based XMLType Tables

XPath queries on XML schema-based XMLType table are candidates for XPath query rewrite. An existsNode() expression in a query may be rewritten to a set of operators on the underlying object-relational columns of the schema-based table. In

such a case, the CTXXPATH index can no longer be used by the query, since it can only be used to satisfy existsNode() queries on the index expression, specified during index creation time.

In Example 4–35, a CTXXPATH index is created on table PURCHASEORDER. The existsNode() expression specified in the WHERE clause gets rewritten into an expression that checks if the underlying object-relational column is not NULL. This is in accordance with XPath query rewrite rules. If the hint /*+ NO_XML_QUERY_REWRITE */ causes XPath query rewrite to be turned off for the query, the existsNode() expression is left unchanged.

Example 4–35 Creating a CTXXPATH Index on a Schema-Based XMLType Table

```
create index PURCHASEORDER_XPATH
on PURCHASEORDER (object_value)
indextype is CTXSYS.CTXXPATH;

Index created.

--
explain plan for
select extractValue(object_value,'/PurchaseOrder/Reference')
  from PURCHASEORDER
 where existsNode(object_value,'/PurchaseOrder/LineItems/LineItem[Description="The Rock"]') = 1;

Explained.

--
set ECHO OFF

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------
Plan hash value: 122532357

-------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |    13 | 65520 |   823   (1)| 00:00:10 |
|*  1 |  HASH JOIN SEMI       |                    |    13 | 65520 |   823   (1)| 00:00:10 |
|   2 |   TABLE ACCESS FULL   | PURCHASEORDER      |   134 | 56146 |     4   (0)| 00:00:01 |
|*  3 |   INDEX FAST FULL SCAN| LINEITEM_TABLE_IOT |    13 | 60073 |   818   (0)| 00:00:10 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")
   3 - filter("DESCRIPTION"='The Rock')

Note
-----
   - dynamic sampling used for this statement

20 rows selected.

--
explain plan for
select /*+ NO_XML_QUERY_REWRITE */ extractValue(object_value,'/PurchaseOrder/Reference')
  from PURCHASEORDER
 where existsNode(object_value,'/PurchaseOrder/LineItems/LineItem[Description="The Rock"]') = 1;

Explained.

--
set ECHO OFF
```

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------------------
Plan hash value: 3192700042

-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                     |     1 |   419 |     4   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER       |     1 |   419 |     4   (0)| 00:00:01 |
|   2 |   DOMAIN INDEX              | PURCHASEORDER_XPATH |       |       |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(EXISTSNODE(SYS_MAKEXML('3A7F7DBBEE5543A486567A908C71D65A',3664,"PURCHASEORDER
              "."XMLEXTRA","PURCHASEORDER"."XMLDATA"),'/PurchaseOrder/LineItems/LineItem[Description="The
               Rock"]')=1)

16 rows selected.
```

Determining If an Index is Being Used: Tracing

Use tracing to determine whether or not an index is being used.

See Also:

- *Oracle Database SQL Reference*
- *Oracle Database Performance Tuning Guide*

CTXXPATH Indexing Depends on Storage Options and Document Size

The choice of whether to use `CTXXPATH` indexes depends on the storage options used, the size of the documents being indexed, and the query mix involved.

`CTXXPATH` indexes can be used for `existsnode()` queries on non-schema-based `XMLType` tables and columns when the data is stored as a CLOB. `CTXXPATH` indexes are also useful when CLOB portions of *schema*-based documents are queried. The term CLOB-based storage is used to apply to these cases. `CTXXPATH` indexes can also be used for `existsnode()` queries on schema-based `XMLType` columns, tables and views, as well as non-schema-based views. The term object-relational storage is used to apply to these cases.

- **CLOB-based storage.** If the storage option used is CLOB-based storage:
  - Check the query mix to see if a significant fraction involves the same set of XPath expressions. If so, then Oracle Corporation recommends that you create function-based indexes for those expressions.
  - Check the query mix to see if a significant fraction involves `existsnode()` queries. `CTXXPATH` indexes are particularly useful if there are a large number of small documents and for `existsnode()` queries with low selectivity, that is, with relatively fewer number of hits. Under such scenarios, build `CTXXPATH` indexes.

  As a general rule, the use of indexes is recommended for Online Transaction Processing (OLTP) environments with few updates.
- **Object-Relational storage.** If the storage option is object-relational:
  - Check the query mix to see if a significant fraction involves XPath queries that can be rewritten. Chapter 5, "XML Schema Storage and Query: The Basics"

lists the set of XPath queries that can potentially get rewritten. The set of XPath queries that are actually rewritten depends on the type of XPath query as well as the registered XML schema. B*tree, Bitmap and other relational and domain indexes can further be built to improve performance. XPath rewrite offers significant performance advantages. Use it in general. It is enabled by default.

- Check the query mix to see if a significant fraction involves the same set of XPath expressions. If so, then Oracle Corporation recommends that you create function-based indexes for these expressions. In the presence of XPath rewrite, the XPath expressions are sometimes better evaluated using function-based indexes when:

  The queries involve traversing through collections. For example, in `extractvalue(/PurchaseOrder/Lineitems/Lineitem/Addresses/Address)`, multiple collections are traversed under XPath rewrite.

  The queries involve returning a scalar element of a collection. For example, in `extractvalue(/PurchaseOrder/PONOList/PONO[1])`, a single scalar item needs to be returned, and function-based indexes are more efficient for this. In such a case, you can turn off XPath rewrite using query-level or session-level hints, and use the function-based index

- Of the non-rewritten queries, check the query mix to see if a significant fraction involves `existsnode()` queries. If so, then you should build `CTXXPATH` indexes. `CTXXPATH` indexes are particularly useful if there are a large number of small documents, and for `existsnode()` queries with low selectivity, that is, with relatively fewer number of hits.

Note: The use of indexes is in general recommended for OLTP environments that are seldom updated. Maintaining `CTXXPATH` and function-based indexes when there are frequent updates adds an additional overhead. Take this into account when deciding whether function-based indexes, `CTXXPATH` indexes, or both should be built and maintained. When both types of indexes are built, the Oracle Database cost-based optimizer makes a cost-based decision which index to use. Try to first determine statistics on the `CTXXPATH` indexing in order to assist the optimizer in choosing the `CTXXPATH` index when appropriate.

Oracle Text Indexes on XMLType Columns

You can create an Oracle Text index on an `XMLType` column. An Oracle Text index enables the `CONTAINS` operator for Full Text Search over XML.

To create an Oracle Text index, use the `CREATE INDEX` SQL statement with the `INDEXTYPE` specified as shown in Example 4–36.

Example 4–36 Creating an Oracle Text Index

```
create index iPurchaseOrderTextIndex
         on purchaseorder p (object_value)
   indextype is ctxsys.context;

Index created.
```

You can also perform Oracle Text operations such as `CONTAINS` and `SCORE` on `XMLType` columns. Example 4–37 shows and Oracle Text search using `CONTAINS`.

Example 4–37 Searching XML Data Using CONTAINS

```
SELECT DISTINCT extractValue(object_value,'/PurchaseOrder/ShippingInstructions/address') "Address"
  from purchaseorder
 where CONTAINS(object_value,
                '$(Fortieth) INPATH (PurchaseOrder/ShippingInstructions/address)') > 0;

Address
------------------------------------------------------------
1200 East Forty Seventh Avenue
New York
NY
10024
USA

1 row selected.
```

See Also: Chapter 9 "Full Text Search Over XML" for more information on using Oracle Text operations with XML DB.

5

XML Schema Storage and Query: The Basics

This chapter introduces XML Schema and explains how to register and use XML Schema with Oracle XML DB. It also describes how to delete and update XML Schema and create storage structures for schema-based XML. It discusses `simpleType` and `complexType` mapping from XML to SQL storage types as well as XPath rewrite fundamentals.

This chapter contains these topics:

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, `purchaseOrder.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, purchaseOrder.xsd

The following is an example of an XML Schema. It declares a `complexType` called `purchaseOrderType` and a global element `PurchaseOrder` of this type.

```
<xs:schema
  targetNamespace="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
  xmlns:po="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
  <xs:element name="PurchaseOrder" type="po:PurchaseOrderType"/>
    <xs:complexType name="PurchaseOrderType">
      <xs:sequence>
        <xs:element name="Reference" type="po:ReferenceType"/>
        <xs:element name="Actions" type="po:ActionsType"/>
        <xs:element name="Reject" type="po:RejectionType" minOccurs="0"/>
        <xs:element name="Requestor" type="po:RequestorType"/>
        <xs:element name="User" type="po:UserType"/>
        <xs:element name="CostCenter" type="po:CostCenterType"/>
        <xs:element name="ShippingInstructions"
                    type="po:ShippingInstructionsType"/>
        <xs:element name="SpecialInstructions"
                    type="po:SpecialInstructionsType"/>
        <xs:element name="LineItems" type="po:LineItemsType"/>
        <xs:element name="Notes" type="po:NotesType"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="LineItemsType">
      <xs:sequence>
        <xs:element name="LineItem" type="po:LineItemType"
                    maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="LineItemType">
      <xs:sequence>
        <xs:element name="Description" type="po:DescriptionType"/>
        <xs:element name="Part" type="po:PartType"/>
      </xs:sequence>
      <xs:attribute name="ItemNumber" type="xs:integer"/>
    </xs:complexType>
    <xs:complexType name="PartType">
      <xs:attribute name="Id">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:minLength value="10"/>
            <xs:maxLength value="14"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="Quantity" type="po:moneyType"/>
      <xs:attribute name="UnitPrice" type="po:quantityType"/>
    </xs:complexType>
    <xs:simpleType name="ReferenceType">
```

```
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="User" type="po:UserType"/>
          <xs:element name="Date" type="po:DateType" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType">
  <xs:all>
    <xs:element name="User" type="po:UserType" minOccurs="0"/>
    <xs:element name="Date" type="po:DateType" minOccurs="0"/>
    <xs:element name="Comments" type="po:CommentsType" minOccurs="0"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType">
  <xs:sequence>
    <xs:element name="name" type="po:NameType" minOccurs="0"/>
    <xs:element name="address" type="po:AddressType" minOccurs="0"/>
    <xs:element name="telephone" type="po:TelephoneType" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostCenterType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="4"/>
  </xs:restriction>
```

```
  </xs:simpleType>
  <xs:simpleType name="VendorType">
    <xs:restriction base="xs:string">
      <xs:minLength value="0"/>
      <xs:maxLength value="20"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="PurchaseOrderNumberType">
    <xs:restriction base="xs:integer"/>
  </xs:simpleType>
  <xs:simpleType name="SpecialInstructionsType">
    <xs:restriction base="xs:string">
      <xs:minLength value="0"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NameType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="20"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="AddressType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="TelephoneType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="24"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DateType">
    <xs:restriction base="xs:date"/>
  </xs:simpleType>
  <xs:simpleType name="CommentsType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DescriptionType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NotesType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="32767"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Example 5–2 XML Document, purchaseOrder.xml Conforming to XML Schema, purchaseOrder.xsd

The following is an example of an XML document that conforms to XML Schema `purchaseOrder.xsd`:

```
<po:PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:po="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
  xsi:schemaLocation=
    "http://xmlns.oracle.com/xdb/documentation/purchaseOrder
     http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd">
  <Reference>SBELL-20021009123336O1PDT</Reference>
  <Actions>
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Sarah J. Bell</Requestor>
  <User>SBELL</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Sarah J. Bell</name>
    <address>400 Oracle Parkway
             Redwood Shores
             CA
             94065
             USA
    </address>
    <telephone>650 506 7400</telephone>
  </ShippingInstructions>
  <SpecialInstructions>Air Mail</SpecialInstructions>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="2">
      <Description>The Unbearable Lightness Of Being</Description>
      <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>Sisters</Description>
      <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
    </LineItem>
  </LineItems>
  <Notes>Section 1.10.32 of &quot;de Finibus Bonorum et Malorum&quot;,
         written by Cicero in 45 BC

&quot;Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
 doloremque laudantium, totam rem aperiam, eaque ips
a quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt
 explicabo. Nemo enim ipsam voluptatem quia voluptas s
it aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui
 ratione voluptatem sequi nesciunt. Neque porro quisqua
m est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed
 quia non numquam eius modi tempora incidunt ut labore
 et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis
 nostrum exercitationem ullam corporis suscipit laborios
am, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure
 reprehenderit qui in ea voluptate velit esse quam nihil moles
```

```
tiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla
 pariatur?&amp;quot;

1914 translation by H. Rackham

&amp;quot;But I must explain to you how all this mistaken idea of denouncing
 pleasure and praising pain was born and I will give you a c
omplete account of the system, and expound the actual teachings of the great
 explorer of the truth, the master-builder of human happ
iness. No one rejects, dislikes, or avoids pleasure itself, because it is
 pleasure, but because those who do not know how to pursue
pleasure rationally encounter consequences that are extremely painful. Nor again
 is there anyone who loves or pursues or desires to
obtain pain of itself, because it is pain, but because occasionally
 circumstances occur in which toil and pain can procure him some
great pleasure. To take a trivial example, which of us ever undertakes laborious
 physical exercise, except to obtain some advantage
from it? But who has any right to find fault with a man who chooses to enjoy a
 pleasure that has no annoying consequences, or one wh
o avoids a pain that produces no resultant pleasure?&amp;quot;

Section 1.10.33 of &amp;quot;de Finibus Bonorum et Malorum&amp;quot;, written by Cicero
in 45 BC

&amp;quot;At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis
 praesentium voluptatum deleniti atque corrupti quos
dolores et quas molestias excepturi sint occaecati cupiditate non provident,
 similique sunt in culpa qui officia deserunt mollitia a
nimi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et
 expedita distinctio. Nam libero tempore, cum soluta nobis
 est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere
 possimus, omnis voluptas assumenda est, omnis dolor
 repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum
 necessitatibus saepe eveniet ut et voluptates repudiandae
 sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente
 delectus, ut aut reiciendis voluptatibus maiores alias
consequatur aut perferendis doloribus asperiores repellat.&amp;quot;

1914 translation by H. Rackham

&amp;quot;On the other hand, we denounce with righteous indignation and dislike men
 who are so beguiled and demoralized by the charms of
 pleasure of the moment, so blinded by desire, that they cannot foresee the pain
 and trouble that are bound to ensue; and equal blam
e belongs to those who fail in their duty through weakness of will, which is the
 same as saying through shrinking from toil and pain
. These cases are perfectly simple and easy to distinguish. In a free hour, when
 our power of choice is untrammelled and when nothin
g prevents our being able to do what we like best, every pleasure is to be
 welcomed and every pain avoided. But in certain circumsta
nces and owing to the claims of duty or the obligations of business it will
 frequently occur that pleasures have to be repudiated an
d annoyances accepted. The wise man therefore always holds in these matters to
 this principle of selection: he rejects pleasures to
secure other greater pleasures, or else he endures pains to avoid worse
 pains.&amp;quot;
  </Notes>
</po:PurchaseOrder>
```

Note:

The URL used here (`http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd`) is simply a name that uniquely identifies the registered XML Schema within the database; it need not be the physical URL at the which the XML Schema document is located. The target namespace of the XML Schema is another URL, different from the XML Schema location URL, which specifies an abstract namespace within which elements and types get declared.

An XML Schema can optionally specify the target namespace URL. If this attribute is omitted, the XML Schema has no target namespace. The target namespace is commonly the same as the URL of the XML Schema.

An XML instance document must specify the namespace of the root element (same as the target namespace of the XML Schema) and the location (URL) of the XML Schema that defines this root element. The location is specified with attribute `xsi:schemaLocation`. When the XML Schema has no target namespace, use attribute `xsi:noNamespaceSchemaLocation` to specify the schema URL.

Using Oracle XML DB 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 control how instance XML documents get mapped to the database. Because 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 with XML Schema, you must first register the XML Schema. You can then use the XML Schema URLs while creating `XMLType` tables, columns, and views. The XML Schema URL, in other words, the URL that identifies the XML Schema in the database, is associated with the `schemaurl` parameter of `registerSchema`.

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.

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

Why We Need XML Schema

As described in Chapter 4, "XMLType Operations", `XMLType` is a datatype that facilitates storing `XMLType` 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), then 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), then 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.

DTD Support in Oracle XML DB

In addition to supporting XML Schema that provide a structured mapping to object-relational storage, Oracle XML DB also supports DTD specifications in XML instance documents. Though DTDs are not used to derive the mapping, XML processors can still access and interpret the DTDs.

Inline DTD Definitions

When an XML instance document has an inline DTD definition, it is used during document parsing. Any DTD validations and entity declaration handling is done at this point. However, once parsed, the entity references are replaced with actual values and the original entity reference is lost.

External DTD Definitions

Oracle XML DB also supports external DTD definitions if they are stored in the repository. Applications needing to process an XML document containing an external DTD definition such as `/public/flights.dtd`, must first ensure that the DTD document is stored in Oracle XML DB at path `/public/flights.xsd`.

> **See Also:** Chapter 18, "Accessing Oracle XML DB Repository Data"

Managing XML Schemas Using DBMS_XMLSCHEMA

Before an XML Schema can be used by Oracle XML DB, it must be registered with Oracle Database. You register an XML Schema using the PL/SQL package `DBMS_XMLSCHEMA`.

> **See Also:** *Oracle XML API Reference*

Some of the main `DBMS_XMLSCHEMA` functions are:

- **registerSchema()**. This registers an XML Schema with Oracle Database, given:
- **deleteSchema()**. This deletes a previously registered XML Schema.
- **copyEvolve()**. This function is described in Chapter 7, "XML Schema Evolution".

Registering Your XML Schema

The main arguments to function `registerSchema()` are the following:

- `schemaURL` – the XML Schema URL. This is a unique identifier for the XML Schema within Oracle XML DB. Convention is that this identifier is in the form of a URL; however, this is not a requirement. The XML Schema URL is used with Oracle XML DB to identify instance documents, by making the schema location hint identical to the XML Schema URL. Oracle XML DB will never attempt to access the Web server identified by the specified URL.
- `schemaDoc` – the XML Schema source document. This is a `VARCHAR`, `CLOB`, `BLOB`, `BFILE`, `XMLType`, or `URIType` value.
- `CSID` – the character-set ID of the source-document encoding, when `schemaDoc` is a `BFILE` or `BLOB` value.

Example 5–3 Registering an XML Schema Using Package DBMS_XMLSCHEMA

The following code registers the XML Schema at URL `http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd`. This example shows how to register an XML Schema using the `BFILE` mechanism to read the source document from a file on the local file system of the database server.

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC => bfilename('XMLDIR','purchaseOrder.xsd'),
    CSID => nls_charset_id('AL32UTF8'));
END;
/
```

Storage and Access Infrastructure

As part of registering an XML Schema, Oracle XML DB also performs several tasks that facilitate storing, accessing, and manipulating XML instances that conform to the XML Schema. These steps include:

- ***Creating types:*** When an XML Schema is registered, Oracle Database creates the appropriate SQL object types that enable the structured storage of XML documents that conform to this XML Schema. You can use the Schema annotations to control how these object types are named and generated. See "SQL Object Types" on page 5-11 for details.
- ***Creating default tables:*** As part of XML Schema registration, Oracle XML DB generates default `XMLType` tables for all global elements. You can use schema annotations to control the names of the tables and to provide column-level and table-level storage clauses and constraints for use during table creation.

  See Also:

  - "Creating Default Tables During XML Schema Registration" on page 5-12
  - "Oracle XML Schema Annotations" on page 5-21

After registration has completed:

- XMLType tables and columns can be created that are constrained to the global elements defined by this XML Schema.
- XML documents conforming to the XML Schema, and referencing it using the XML Schema instance mechanism, can be processed automatically by Oracle XML DB.

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

Transactional Action 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, then the operation is auto committed.
- If registration fails, then the database is rolled back to the state before the registration began.

Because 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.

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`.

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 26, "Managing Oracle XML DB Using Oracle Enterprise Manager"
- Appendix A, "Installing and Configuring Oracle XML DB"

Debugging XML Schema Registration

You can monitor the object types and tables created during XML Schema registration by setting the following event before calling `DBMS_XMLSCHEMA.registerSchema()`:

```
ALTER SESSION SET events='31098 trace name context forever'
```

Setting this event causes the generation of a log of all the `CREATE TYPE` and `CREATE TABLE` statements. The log is written to the user session trace file, typically found in `<ORACLE_HOME>/admin/<ORACLE_SID>/udump`. This script can be a useful aid in diagnosing problems during XML Schema registration.

See Also: Chapter 3, "Using Oracle XML DB", "XML Schema and Oracle XML DB" on page 3-21

SQL Object Types

Assuming that the parameter `GENTYPES` is set to `TRUE` 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. By default, all SQL object types are created in the database schema of the user who registers the XML Schema. If the `defaultSchema` annotation is used, then Oracle XML DB attempts to create the object type using the specified database schema. The current user must have the necessary privileges to perform this.

Example 5–4 Creating SQL Object Types to Store XMLType Tables

For example, when `purchaseOrder.xsd` is registered with Oracle XML DB, the following SQL types are created.

```
SQL> DESCRIBE "PurchaseOrderType1668_T"

"PurchaseOrderType1668_T" is NOT FINAL
Name                 Null?  Type
-------------------- ------ -------------------------------
SYS_XDBPD$                  XDB.XDB$RAW_LIST_T
Reference                   VARCHAR2(30 CHAR)
```

```
Actions                    ActionsType1661_T
Reject                     RejectionType1660_T
Requestor                  VARCHAR2(128 CHAR)
User                       VARCHAR2(10 CHAR)
CostCenter                 VARCHAR2(4 CHAR)
ShippingInstructions       ShippingInstructionsTyp1659_T
SpecialInstructions        VARCHAR2(2048 CHAR)
LineItems                  LineItemsType1666_T
Notes                      VARCHAR2(4000 CHAR)

SQL> DESCRIBE "LineItemsType1666_T"

"LineItemsType1666_T" is NOT FINAL
Name                 Null? Type
-------------------- ----- ------------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
LineItem                   LineItem1667_COLL

SQL> DESCRIBE "LineItem1667_COLL"

"LineItem1667_COLL" VARRAY(2147483647) OF LineItemType1665_T
"LineItemType1665_T" is NOT FINAL
Name                Null? Type
------------------- ----- --------------------------------
SYS_XDBPD$                XDB.XDB$RAW_LIST_T
ItemNumber                NUMBER(38)
Description               VARCHAR2(256 CHAR)
Part                      PartType1664_T
```

Note: By default, the names of the object types and attributes in the preceding example are system-generated.

- Developers can use schema annotations to provide user-defined names (see "Oracle XML Schema Annotations" for details).
- If the XML Schema does not contain the `SQLName` attribute, then the name is derived from the XML name.

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`, then 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.

Example 5–5 Default Table for Global Element PurchaseOrder

```
SQL> DESCRIBE "PurchaseOrder1669_TAB"

Name                        Null? Type
--------------------------- ----- ----------------------
TABLE of
```

```
SYS.XMLTYPE(
  XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  Element "PurchaseOrder")
STORAGE Object-relational TYPE "PurchaseOrderType1668_T"
```

Generated Names are Case Sensitive

The names of SQL tables, object, and attributes generated by XML Schema registration are *case sensitive*. For instance, in Example 5–3, "Registering an XML Schema Using Package DBMS_XMLSCHEMA", a table called `PurchaseOrder1669_TAB` was created automatically during registration of the XML Schema. Since the table name was derived from the element name, `PurchaseOrder`, the name of the table is also mixed case. This means that you must refer to this table in SQL using a quoted identifier: `"PurchaseOrder1669_TAB"`. Failure to do so results in an object-not-found error, such as `ORA-00942: table or view does not exist`.

Objects That Depend on Registered XML Schemas

The following objects are dependent on 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.

How to Obtain a List of Registered XML Schemas

To obtain a list of the XML Schemas registered with Oracle XML DB using `DBMS_XMLSCHEMA.registerSchema` use the following code. You can also use `user_xml_schemas`, `all_xml_schemas`, `user_xml_tables`, and `all_xml_tables`.

Example 5–6 Data Dictionary Table for Registered Schemas

```
SQL> DESCRIBE DBA_XML_SCHEMAS

Name          Null? Type
------------ ----- ----------------------
OWNER               VARCHAR2(30)
SCHEMA_URL          VARCHAR2(700)
LOCAL               VARCHAR2(3)
SCHEMA              XMLTYPE(XMLSchema "http://xmlns.oracle.com/xdb/XDBSchema.xsd"
                            Element "schema")
INT_OBJNAME         VARCHAR2(4000)
QUAL_SCHEMA_URL     VARCHAR2(767)

SQL> SELECT owner, local, schema_url FROM dba_xml_schemas;

OWNER   LOC   SCHEMA_URL
-----   ---   ---------------------
XDB     NO    http://xmlns.oracle.com/xdb/XDBSchema.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBResource.xsd
XDB     NO    http://xmlns.oracle.com/xdb/acl.xsd
XDB     NO    http://xmlns.oracle.com/xdb/dav.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBStandard.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/xdblog.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/ftplog.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/httplog.xsd
```

```
XDB     NO    http://www.w3.org/2001/xml.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBFolderListing.xsd
XDB     NO    http://xmlns.oracle.com/xdb/stats.xsd
XDB     NO    http://xmlns.oracle.com/xdb/xdbconfig.xsd
SCOTT   YES   http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd

13 rows selected.

SQL> DESCRIBE DBA_XML_TABLES

Name          Null? Type
------------ ----- ----------------------
OWNER               VARCHAR2(30)
TABLE_NAME          VARCHAR2(30)
XMLSCHEMA           VARCHAR2(700)
SCHEMA_OWNER        VARCHAR2(30)
ELEMENT_NAME        VARCHAR2(2000)
STORAGE_TYPE        VARCHAR2(17)

SQL> SELECT table_name FROM dba_xml_tables WHERE
       XMLSCHEMA =
         'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd';

TABLE_NAME
------------------------------
PurchaseOrder1669_TAB

1 row selected.
```

Deleting 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, then it raises an error and the deletion operation fails. This is referred to as the **RESTRICT** mode of deleting XML Schemas.

FORCE Mode

When deleting XML Schemas, if you specify the `FORCE` mode option, then the XML Schema deletion proceeds even if it fails the dependency check. In this mode, XML Schema deletion marks all its dependents as invalid.

The `CASCADE` mode option drops all generated types and default tables as part of a previous call to register XML Schema.

> **See Also:** *Oracle XML API Reference* the chapter on `DBMS_XMLSCHEMA`

Example 5–7 Deleting the XML Schema Using DBMS_XMLSCHEMA

The following example deletes XML Schema `purchaseOrder.xsd`. Then, the schema is deleted using the `FORCE` and `CASCADE` modes with `DBMS_XMLSCHEMA.DELETESCHEMA`:

```
BEGIN
  DBMS_XMLSCHEMA.deleteSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    DELETE_OPTION => dbms_xmlschema.DELETE_CASCADE_FORCE);
END;
/
```

XML Schema-Related Methods of XMLType

Table 5–1 lists the `XMLType` XML Schema-related methods.

Table 5–1 XMLType API XML Schema-Related Methods

XMLType API Method	Description
`isSchemaBased()`	Returns `TRUE` if the `XMLType` instance is based on an XML Schema, `FALSE` otherwise.

`getSchemaURL()`
`getRootElement()`
`getNamespace()`
	Returns the XML Schema URL, name of root element, and the namespace for an XML Schema-based `XMLType` instance.

`schemaValidate()`
`isSchemaValid()`
`is SchemaValidated()`
`setSchemaValidated()`
	An `XMLType` instance can be validated against a registered XML Schema using the `validation` methods. See Chapter 8, "Transforming and Validating XMLType Data".

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 to the Oracle XML DB repository. The XML Schema URL determines the path name of the resource in Oracle XML DB repository (and is associated with the `schemaurl` parameter of `registerSchema`) according to the following rules:

Local XML Schema

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. Such XML Schemas are referred to as *local*. In general, they are usable only by you, the owner.

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 5–8 Registering A Local XML Schema

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC => bfilename('XMLDIR','purchaseOrder.xsd'),
    LOCAL => TRUE,
    GENTYPES => TRUE,
    GENTABLES => FALSE,
    CSID => nls_charset_id('AL32UTF8'));
END;
/
```

If this local XML Schema is registered by user `SCOTT`, it is given this path name:

```
/sys/schemas/SCOTT/xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd
```

Database users need appropriate permissions and Access Control Lists (ACL) to create a resource with this path name in order to register the XML Schema as a local XML Schema.

> **See Also:** Chapter 23, "Oracle XML DB Resource Security"

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 Database supports fully qualified XML Schema URLs, which can be specified as:

```
http://xmlns.oracle.com/xdb/schemas/SCOTT/xmlns.oracle.com/xdb/documentation/purchaseOrder.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 schemas, 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 writable 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 Access Control Lists (ACL). See Chapter 23, "Oracle XML DB Resource Security" for further information on privileges and for details on the 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 5–9 Registering A Global XML Schema

```
SQL> GRANT XDBADMIN TO SCOTT;

Grant succeeded.

CONNECT scott/tiger

Connected.

BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC => bfilename('XMLDIR','purchaseOrder.xsd'),
    LOCAL => FALSE,
    GENTYPES => TRUE,
    GENTABLES => FALSE,
    CSID => nls_charset_id('AL32UTF8'));
END;
/
```

If this local XML Schema is registered by user SCOTT, it is given this path name:

```
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd
```

Database users need appropriate permissions (ACLs) to create this resource in order to register the XML Schema as *global*.

DOM Fidelity

Document Object Model (DOM) fidelity is the concept of retaining the structure of a retrieved XML document, compared to the original XML document, for DOM traversals. DOM fidelity is needed to ensure the accuracy and integrity of XML documents stored in Oracle XML DB.

> **See Also:** "Overriding the SQLType Value in XML Schema When Declaring Attributes" on page 5-31 and "Overriding the SQLType Value in XML Schema When Declaring Elements" on page 5-31

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 similar the *returned* and original XML documents are, particularly for purposes of DOM traversals.

DOM Fidelity and SYS_XDBPD$

In order to provide DOM fidelity, Oracle XML DB has to maintain instance-level metadata. This metadata is tracked at a type level using the system-defined binary attribute SYS_XDBPD$. This attribute is referred to as the **Positional Descriptor**, or PD for short. The PD attribute is intended for Oracle Corporation internal use only. You should never directly access or manipulate this column.

This positional descriptor 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.

If DOM fidelity is not required, you can suppress SYS_XDBPD$ in the XML Schema definition by setting the attribute, maintainDOM=FALSE at the type level.

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.

In general however, it is not a good idea to suppress the PD attribute because the extra pieces of information, such as comments, processing instructions, and so on, could be lost if there is no PD column.

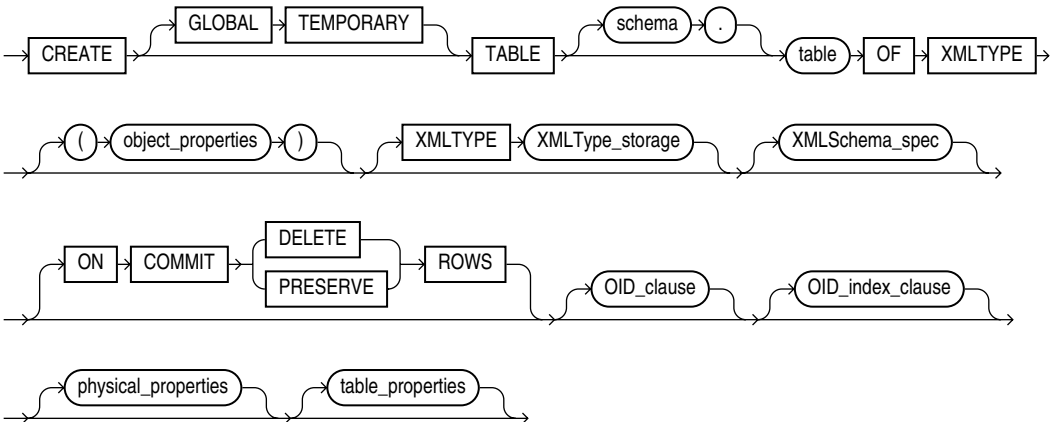
Creating XMLType Tables and Columns Based on XML Schema

Using Oracle XML DB, developers can create XMLType tables and columns that are constrained to a global element defined by a registered XML Schema. After an XMLType column has been constrained to a particular element and XML Schema it can only contain documents that are compliant with the schema definition of that element. An XMLType table column is constrained to a particular element and XML Schema by adding the appropriate XMLSCHEMA and ELEMENT clauses to the CREATE TABLE operation.

Figure 5–1 shows the syntax for creating an XMLType table:

```
CREATE [GLOBAL TEMPORARY] TABLE [schema.] table OF XMLType
  [(object_properties)] [XMLType XMLType_storage] [XMLSchema_spec]
  [ON COMMIT {DELETE | PRESERVE} ROWS] [OID_clause] [OID_index_clause]
  [physical_properties] [table_properties];
```

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. See also Chapter 4, "XMLType Operations".

Example 5–10 Creating XML Schema-Based XMLType Tables and Columns

This example shows `CREATE TABLE` statements. The first creates an `XMLType` table, `purchaseorder_as_table`. The second creates a relational table, `purchaseorder_as_column`, with an `XMLType` column, `xml_document`. In both, the `XMLType` value is constrained to the `PurchaseOrder` element defined by the schema registered under the URL `http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd`.

```
CREATE TABLE purchaseorder_as_table OF XMLType
  XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder";

CREATE TABLE purchaseorder_as_column (id NUMBER, xml_document XMLType)
  XMLType COLUMN xml_document
  ELEMENT
    "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd#PurchaseOrder";
```

Note there are two ways to define the element and schema to be used. In one way, the `XMLSchema` and `Element` are specified as separator clauses. In the other way, the `XMLSchema` and `Element` are specified using the `Element` clause, using an XPointer notation.

The data associated with an `XMLType` table or column that is constrained to an XML Schema can be stored in two different ways:

- Shred the contents of the document and store it as a set of objects. This is known as **structured storage**.
- Stored the contents of the document as text, using a single LOB column. This is known as **unstructured storage**.

Specifying Unstructured (LOB-Based) Storage of Schema-Based XMLType

The default storage model is structured storage. To override this behavior, and store the entire XML document as a single LOB column, use the `STORE AS CLOB` clause.

Example 5–11 Specifying CLOB Storage for Schema-Based XMLType Tables and Columns

This example shows how to create an XMLType table and a table with an XMLType column, where the contents of the XMLType are constrained to a global element defined by a registered XML Schema, and the contents of the XMLType are stored using a single LOB column.

```
CREATE TABLE purchaseorder_as_table OF XMLType
  XMLType STORE AS CLOB
  XMLSCHEMA "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder";

CREATE TABLE purchaseorder_as_column (id NUMBER, xml_document XMLType)
  XMLType COLUMN xml_document
  STORE AS CLOB
  XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder";
```

Note that you can add LOB storage parameters to the STORE AS CLOB clause.

Specifying Storage Models for Structured Storage of Schema-Based XMLType

When structured storage is selected, collections (elements which have maxOccurs > 1, allowing them to appear multiple times) are mapped into SQL VARRAY values. By default, the entire contents of such a VARRAY is serialized using a single LOB column. This storage model provides for optimal ingestion and retrieval of the entire document, but it has significant limitations when it is necessary to index, update, or retrieve individual members of the collection. A developer may override the way in which a VARRAY is stored, and force the members of the collection to be stored as a set of rows in a nested table. This is done by adding an explicit VARRAY STORE AS clause to the CREATE TABLE statement.

Developers can also add STORE AS clauses for any LOB columns that will be generated by the CREATE TABLE statement.

Note that the collection and the LOB column must be identified using object-relational notation. Therefore, it is important to understand the structure of the objects that are generated when a XML Schema is registered.

Example 5–12 Specifying Storage Options for Schema-Based XMLType Tables and Columns Using Structured Storage

This example shows how to create an XMLType table and a table with an XMLType column, where the contents of the XMLType are constrained to a global element defined by a registered XML Schema, and the contents of the XMLType are stored using as a set of SQL objects.

```
CREATE table purchaseorder_as_table
  OF XMLType (UNIQUE ("XMLDATA"."Reference"),
              FOREIGN KEY ("XMLDATA"."User") REFERENCES hr.employees (email))
ELEMENT
  "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd#PurchaseOrder"
  VARRAY "XMLDATA"."Actions"."Action"
    STORE AS TABLE action_table1 ((PRIMARY KEY (nested_table_id, array_index))
                                  ORGANIZATION INDEX OVERFLOW)
  VARRAY "XMLDATA"."LineItems"."LineItem"
    STORE AS TABLE lineitem_table1 ((PRIMARY KEY (nested_table_id, array_index))
                                    ORGANIZATION INDEX OVERFLOW)
  LOB ("XMLDATA"."Notes")
    STORE AS (TABLESPACE USERS ENABLE STORAGE IN ROW
```

```
              STORAGE(INITIAL 4K NEXT 32K));

CREATE TABLE purchaseorder_as_column (
  id NUMBER,
  xml_document XMLType,
  UNIQUE (xml_document."XMLDATA"."Reference"),
  FOREIGN KEY (xml_document."XMLDATA"."User") REFERENCES hr.employees (email))

XMLType COLUMN xml_document
XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
ELEMENT "PurchaseOrder"
  VARRAY xml_document."XMLDATA"."Actions"."Action"
    STORE AS TABLE action_table2 ((PRIMARY KEY (nested_table_id, array_index))
                                  ORGANIZATION INDEX OVERFLOW)
  VARRAY xml_document."XMLDATA"."LineItems"."LineItem"
    STORE AS TABLE lineitem_table2 ((PRIMARY KEY (nested_table_id, array_index))
                                    ORGANIZATION INDEX OVERFLOW)
  LOB (xml_document."XMLDATA"."Notes")
    STORE AS (TABLESPACE USERS ENABLE STORAGE IN ROW
              STORAGE(INITIAL 4K NEXT 32K));
```

The example also shows how to specify that the collection of `Action` elements and the collection of `LineItem` elements are stored as rows in nested tables, and how to specify LOB storage clauses for the LOB that will contain the content of the `Notes` element.

> **Note:** Oracle Corporation recommends the use of the thick JDBC driver especially to operate on `XMLType` values stored object-relationally. Note that the thin JDBC driver currently supports only `XMLType` values stored as `CLOB` values.

Specifying Relational Constraints on XMLType Tables and Columns

When structured storage is selected, typical relational constraints can be specified for elements and attributes that occur once in the XML document. Example 5–12 shows how to use object-relational notation to define a unique constraint and a foreign key constraint when creating the table.

Note that it is not possible to define constraints for `XMLType` tables and columns that make use of unstructured storage.

Oracle XML Schema Annotations

Oracle XML DB gives application developers the ability to influence the objects and tables that are generated by the XML Schema registration process. You use the schema annotation mechanism to do this.

Annotation involves adding extra attributes to the `complexType`, `element`, and `attribute` definitions that are declared by the XML Schema. The attributes used by Oracle XML DB belong to the namespace `http://xmlns.oracle.com/xdb`. In order to simplify the process of annotationg an XML Schema, it is recommended that a namespace prefix be declared in the root element of the XML Schema.

Common reasons for wanting to annotate an XML Schema include the following:

- When `GENTYPES` or `GENTABLES` is set `TRUE`, schema annotation makes it possible for developers to ensure that the names of the tables, objects, and attributes

created by `registerSchema()` are well-known names, compliant with any application-naming standards.

- When `GENTYPES` or `GENTABLES` is set `FALSE`, schema annotation makes it possible for developers to map between the XML Schema and existing objects and tables within the database.
- To prevent the generation of mixed-case names that require the use of quoted identifies when working directly with SQL.

The most commonly used annotations are the following:

- `defaultTable` – Used to control the name of the default table generated for each global element when the `GENTABLES` parameter is `FALSE`. Setting this to `""` will prevent a default table from being generated for the element in question.
- `SQLName` – Used to specify the name of the SQL attribute that corresponds to each element or attribute defined in the XML Schema
- `SQLType` – For `complexType` definitions, `SQLType` is used to specify the name of the SQL object type that are corresponds to the `complexType` definitions. For `simpleType` definitions, `SQLType` is used to override the default mapping between XML Schema data types and SQL data types. A very common use of `SQLType` is to define when unbounded strings should be stored as `CLOB` values, rather than `VARCHAR(4000) CHAR` values (the default).
- `SQLCollType` – Used to specify the name of the `VARRAY` type that will manage a collection of elements.
- `maintainDOM` – Used to determine whether or not DOM fidelity should be maintained for a given `complexType` definition
- `storeVarrayAsTable` – Specified in the root element of the XML Schema. Used to force all collections to be stored as nested tables. There will be one nested table created for each element that specifies `maxOccurs > 1`. The nested tables will be created with system-generated names.

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.

Example 5–13 shows a partial listing of the XML Schema in Example 5–1, modified to include some of the most important XDB annotations.

Example 5–13 Using Common Schema Annotations

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

```
      <xs:element name="Reject" type="po:RejectionType" minOccurs="0"/>
      <xs:element name="Requestor" type="po:RequestorType"/>
      <xs:element name="User" type="po:UserType" minOccurs="1"
                  xdb:SQLName="EMAIL"/>
      <xs:element name="CostCenter" type="po:CostCenterType"/>
      <xs:element name="ShippingInstructions"
                  type="po:ShippingInstructionsType"/>
      <xs:element name="SpecialInstructions" type="po:SpecialInstructionsType"/>
      <xs:element name="LineItems" type="po:LineItemsType"
                  xdb:SQLName="LINEITEM_COLLECTION"/>
      <xs:element name="Notes" type="po:NotesType" xdb:SQLType="CLOB"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
    <xs:sequence>
      <xs:element name="LineItem" type="po:LineItemType" maxOccurs="unbounded"
                  xdb:SQLCollType="LINEITEM_V" xdb:SQLName="LINEITEM_VARRAY"/>
   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
    <xs:sequence>
      <xs:element name="Description" type="po:DescriptionType"/>
      <xs:element name="Part" type="po:PartType"/>
    </xs:sequence>
    <xs:attribute name="ItemNumber" type="xs:integer"/>
  </xs:complexType>
  <xs:complexType name="PartType" xdb:SQLType="PART_T" xdb:maintainDOM="false">
    <xs:attribute name="Id">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:minLength value="10"/>
          <xs:maxLength value="14"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="Quantity" type="po:moneyType"/>
    <xs:attribute name="UnitPrice" type="po:quantityType"/>
  </xs:complexType>
</xs:schema>
```

The schema element includes the declaration of the `xdb` namespace. It also includes the annotation `xdb:storeVarrayAsTable="true"`. This will force all collections within the XML Schema to be managed using nested tables.

The definition of the global element `PurchaseOrder` includes a `defaultTable` annotation that specifies that the name of the default table associated with this element is `PURCHASEORDER`.

The global `complexType` `PurchaseOrderType` includes a `SQLType` annotation that specifies that the name of the generated SQL object type will be `PURCHASEORDER_T`. Within the definition of this type, the following annotations are used:

- The element `Reference` includes a `SQLName` annotation that ensures that the name of the SQL attribute corresponding to the `Reference` element will be named `REFERENCE`.
- The element `Actions` includes a `SQLName` annotation that ensures that the name of the SQL attribute corresponding to the `Actions` element will be `ACTION_COLLECTION`.

- The element USER includes a SQLName annotation that ensures that the name of the SQL attribute corresponding to the User element will be EMAIL.
- The element LineItems includes a SQLName annotation that ensures that the name of the SQL attribute corresponding to the LineItems element will be LINEITEM_COLLECTION.
- The element Notes includes a SQLType annotation that ensures that the datatype of the SQL attribute corresponding to the Notes element will be CLOB.

The global complexType LineItemsType includes a SQLType annotation that specifies that the names of generated SQL object type will be LINEITEMS_T. Within the definition of this type, the following annotations are used:

- The element LineItem includes a SQLName annotation that ensures that the datatype of the SQL attribute corresponding to the LineItems element will be LINEITEM_VARRAY, and a SQLCollName annotation that ensures that the name of the SQL object type that manages the collection will be LINEITEM_V.

The global complexType LineItemType includes a SQLType annotation that specifies that the names of generated SQL object type will be LINEITEM_T.

The global complexType PartType includes a SQLType annotation that specifies that the names of generated SQL object type will be PART_T. It also includes the annotation xdb:maintainDOM="false", specifying that there is no need for Oracle XML DB to maintain DOM fidelity for elements based on this type.

Example 5–14 Results of Registering an Annotated XML Schema

The following code shows some of the tables and objects created when the annotated XML Schema is registered.

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC => bfilename('XMLDIR','purchaseOrder.Annotated.xsd'),
    LOCAL => TRUE,
    GENTYPES => TRUE,
    GENTABLES => TRUE,
    CSID => nls_charset_id('AL32UTF8'));
END;
/

SQL> SELECT TABLE_NAME, XMLSCHEMA, ELEMENT_NAME FROM USER_XML_TABLES;

TABLE_NAME     XMLSCHEMA                              ELEMENT_NAME
-------------  ---------------------------------   -------------
PURCHASEORDER  http://xmlns.oracle.com/xdb/documen   PurchaseOrder
               tation/purchaseOrder.xsd

1 row selected.

SQL> DESCRIBE PURCHASEORDER

Name                           Null? Type
-----------------------------  ----- -----------------
TABLE of SYS.XMLTYPE(XMLSchema
 "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
 ELEMENT "PurchaseOrder") STORAGE Object-relational TYPE "PURCHASEORDER_T"

SQL> DESCRIBE PURCHASEORDER_T
```

```
PURCHASEORDER_T is NOT FINAL
Name                 Null? Type
-------------------- ----- -------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
REFERENCE                  VARCHAR2(30 CHAR)
ACTION_COLLECTION          ACTIONS_T
Reject                     REJECTION_T
Requestor                  VARCHAR2(128 CHAR)
EMAIL                      VARCHAR2(10 CHAR)
CostCenter                 VARCHAR2(4 CHAR)
ShippingInstructions       SHIPPING_INSTRUCTIONS_T
SpecialInstructions        VARCHAR2(2048 CHAR)
LINEITEM_COLLECTION        LINEITEMS_T
Notes                      CLOB

SQL> DESCRIBE LINEITEMS_T
LINEITEMS_T is NOT FINAL
Name                 Null? Type
-------------------- ----- -------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
LINEITEM_VARRAY            LINEITEM_V

SQL> DESCRIBE LINEITEM_V

LINEITEM_V VARRAY(2147483647) OF LINEITEM_T
LINEITEM_T is NOT FINAL
Name                 Null? Type
-------------------- ----- -------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
ItemNumber                 NUMBER(38)
Description                VARCHAR2(256 CHAR)
Part                       PART_T

SQL> DESCRIBE PART_T

PART_T is NOT FINAL
Name                 Null? Type
-------------------- ----- -------------------------
Id                         VARCHAR2(14 CHAR)
Quantity                   NUMBER(12,2)
UnitPrice                  NUMBER(8,4)

SQL> SELECT TABLE_NAME, PARENT_TABLE_COLUMN FROM USER_NESTED_TABLES
       WHERE PARENT_TABLE_NAME = 'PURCHASEORDER';

TABLE_NAME                     PARENT_TABLE_COLUMN
----------                     ----------------------
SYS_NTNOHV+tfSTRaDTA9FETvBJw==   "XMLDATA"."LINEITEM_COLLECTION"."LINEITEM_VARRAY"
SYS_NTV4bNVqQ1S4WdCIvBK5qjZA==   "XMLDATA"."ACTION_COLLECTION"."ACTION_VARRAY"

2 rows selected.
```

A table called PURCHASEORDER has been created.

Types called PURCHASEORDER_T, LINEITEMS_T, LINEITEM_V, LINEITEM_T, and PART_T have been created. The attributes defined by these types are named according to supplied the SQLName annotations.

The Notes attribute defined by PURCHASEORDER_T has a datatype of CLOB.

PART_T does not include a Positional Descriptor attribute.

Nested tables have been created to manage the collections of `LineItem` and `Action` elements.

Table 5–2 lists Oracle XML DB annotations that you can specify in element and attribute declarations.

Table 5–2 Annotations 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 `maxOccurs>1`.
`SQLSchema`	Any SQL username	User registering XML Schema	Name of database user owning the type specified by `SQLType`.
`SQLCollSchema`	Any SQL username	User registering XML Schema	Name of database user owning the type specified by `SQLCollType`.
`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 `maxOccurs` > 1). If false, a `REF` value is stored (or a collection of `REF` values, if `maxOccurs>1`). This attribute is 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 action as the input.
`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.

Table 5–3 Annotations 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.
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 action as the input.

Table 5–4 Annotations You Can Specify in XML Schema Declarations

Attribute	Values	Default	Description
mapUnbounded StringToLob	true | false	false	If true, unbounded strings are mapped to CLOB by default. Similarly, unbounded binary data gets mapped to a Binary Large Object (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.

Querying a Registered XML Schema to Obtain Annotations

The registered version of an XML Schema will contain a full set of XDB annotations. As was shown in Example 5–8, and Example 5–9, the location of the registered XML Schema depends on whether the schema is a local or global schema.

This document can be queried to find out the values of the annotations that were supplied by the user, or added by the schema registration process. For instance, the following query shows the set of global complexType definitions declared by the XMLSchema and the corresponding SQL objects types:

Example 5–15 Querying Metadata from a Registered XML Schema

```
SELECT extractValue(value(ct),
                    '/xs:complexType/@name',
                    'xmlns:xs="http://www.w3.org/2001/XMLSchema"
                     xmlns:xdb="http://xmlns.oracle.com/xdb"')
       XMLSCHEMA_TYPE_NAME,
       extractValue(value(ct),
                    '/xs:complexType/@xdb:SQLType',
                    'xmlns:xs="http://www.w3.org/2001/XMLSchema"
                     xmlns:xdb="http://xmlns.oracle.com/xdb"')
       SQL_TYPE_NAME
  FROM resource_view,
       table(
         xmlsequence(
           extract(
```

```
            res,
            '/r:Resource/r:Contents/xs:schema/xs:complexType',
            'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
             xmlns:po="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:xdb="http://xmlns.oracle.com/xdb"'))) ct
  WHERE
    equals_path(
      res,
     '/sys/schemas/SCOTT/xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd')
    =1;

XMLSCHEMA_TYPE_NAME              SQL_TYPE_NAME
-------------------------------- ------------------------------
PurchaseOrderType                PURCHASEORDER_T
LineItemsType                    LINEITEMS_T
LineItemType                     LINEITEM_T
PartType                         PART_T
ActionsType                      ACTIONS_T
RejectionType                    REJECTION_T
ShippingInstructionsType         SHIPPING_INSTRUCTIONS_T

7 rows selected.
```

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 5-30 and adds annotations to the XML Schema document to store the mapping information. Annotations are in the form of new attributes.

Example 5–16 Capturing SQL Mapping Using SQLType and SQLName Attributes

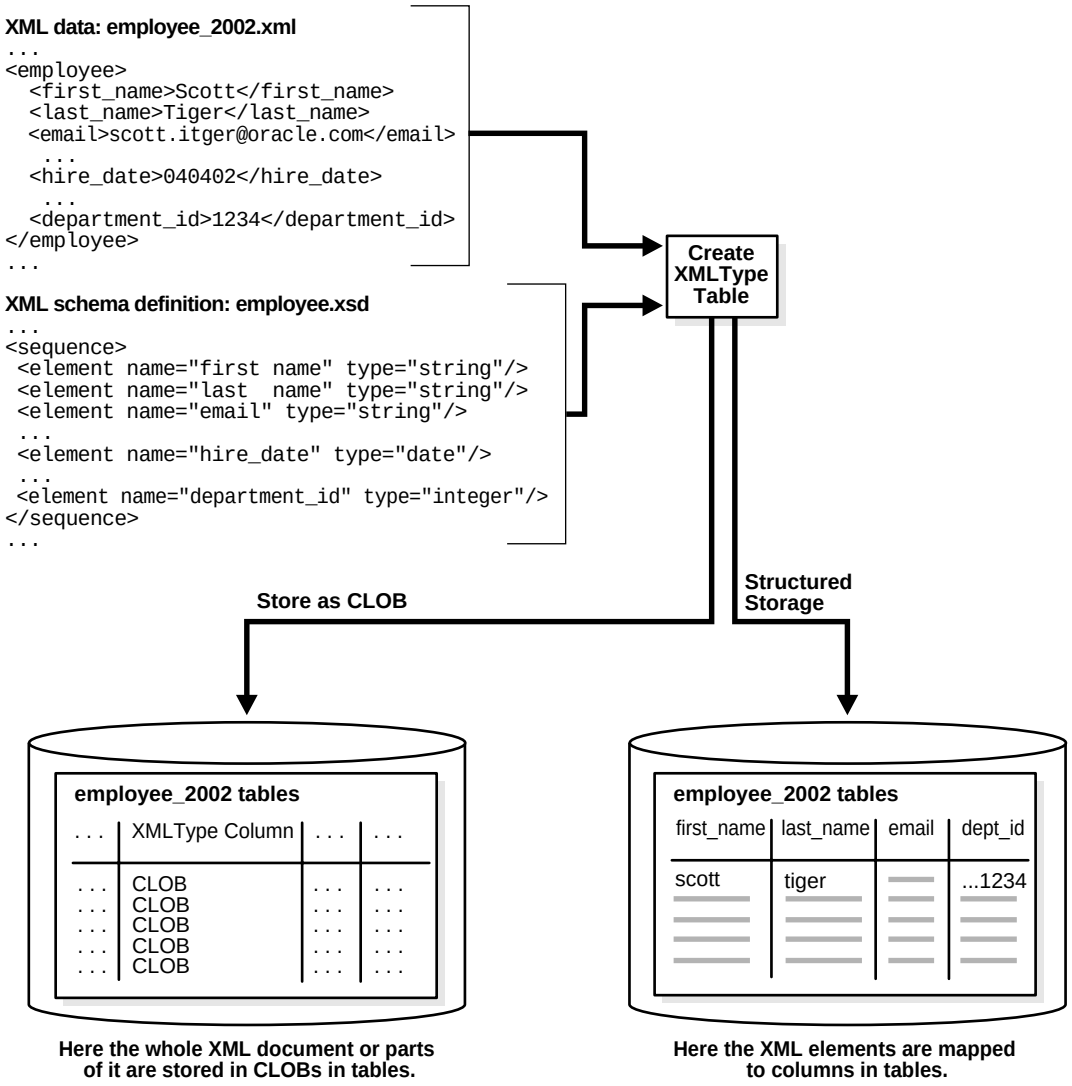
The following XML Schema definition shows how SQL mapping information is captured using `SQLType` and `SQLName` attributes:

```
DECLARE
  doc VARCHAR2(3000) :=
   '<schema
     targetNamespace="http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
     xmlns:po="http://xmlns.oracle.com/xdb/documentation/purchaseOrder.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>
      <element name="PurchaseOrder" type="po:PurchaseOrderType"/>
    </schema>';
BEGIN
   DBMS_XMLSCHEMA.registerSchema(
     'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.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.

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-relationally 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 the SQLType Value in XML Schema When Declaring Attributes

You can explicitly specify a `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`, then you can override it with any of the following: `CHAR`, `VARCHAR`, or `CLOB`.
- If the default type is `RAW`, then 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 5-30.
- Global `complexType`, specified within this XML Schema document or in an external XML Schema.
- Reference to a global element (`ref="..."`), which could itself be within this XML Schema document or in an external XML Schema.
- Local `complexType`.

Overriding the SQLType Value in XML Schema When Declaring Elements

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.

For example, to override the `SQLType` from `VARCHAR2` to `CLOB` declare the XDB namespace as follows:

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

and then use `xdb:SQLType="CLOB"`.

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`, then 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 be created.
  - If SQLInline ="true", then 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", then the SQL type is set to XDB.XDB$XMLTYPE_REF_LIST_T, a predefined type representing an array of REF values to XMLType.
- If the element is a global element, or if SQLInline="false", then the system creates a default table. The name of the default table is specified by you or derived by mangling the element name.

See Also: Chapter 6, "XML Schema Storage and Query: Advanced Topics" for more information about mapping simpleType values and complexType values to SQL.

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 or CLOB) datatype in SQL.

Figure 5–3 Mapping simpleType: XML Strings to SQL VARCHAR2 or CLOBs

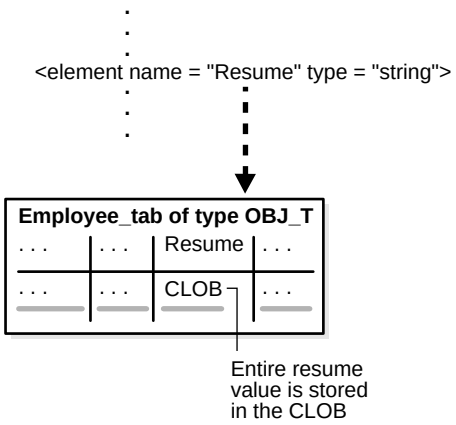


Table 5–5 Mapping XML String Datatypes to SQL

XML Primitive Type	Length or MaxLength Facet	Default Mapping	Compatible Datatype
string	n	VARCHAR2(n) if n < 4000, else VARCHAR2(4000)	CHAR, CLOB
string	--	VARCHAR2(4000) if mapUnboundedStringToLob="false", CLOB	CHAR, 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="false", 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)	FLOAT, DOUBLE, BINARY_FLOAT
double	NUMBER	NUMBER(m,n)	FLOAT, DOUBLE, BINARY_DOUBLE
decimal	NUMBER	NUMBER(m,n)	FLOAT, DOUBLE
integer	NUMBER	NUMBER(m,n)	NUMBER
nonNegativeInteger	NUMBER	NUMBER(m,n)	NUMBER
positiveInteger	NUMBER	NUMBER(m,n)	NUMBER
nonPositiveInteger	NUMBER	NUMBER(m,n)	NUMBER
negativeInteger	NUMBER	NUMBER(m,n)	NUMBER
long	NUMBER(20)	NUMBER(m,n)	NUMBER
unsignedLong	NUMBER(20)	NUMBER(m,n)	NUMBER
int	NUMBER(10)	NUMBER(m,n)	NUMBER
unsignedInt	NUMBER(10)	NUMBER(m,n)	NUMBER
short	NUMBER(5)	NUMBER(m,n)	NUMBER
unsignedShort	NUMBER(5)	NUMBER(m,n)	NUMBER
byte	NUMBER(3)	NUMBER(m,n)	NUMBER
unsignedByte	NUMBER(3)	NUMBER(m,n)	NUMBER

Table 5–8 Mapping XML Date Datatypes to SQL

XML Primitive Type	Default Mapping	Compatible Datatypes
datetime	TIMESTAMP	TIMESTAMP WITH TIME ZONE, DATE
time	TIMESTAMP	TIMESTAMP WITH TIME ZONE, DATE
date	DATE	TIMESTAMP WITH TIME ZONE
gDay	DATE	TIMESTAMP WITH TIME ZONE
gMonth	DATE	TIMESTAMP WITH TIME ZONE
gYear	DATE	TIMESTAMP WITH TIME ZONE
gYearMonth	DATE	TIMESTAMP WITH TIME ZONE
gMonthDay	DATE	TIMESTAMP WITH TIME ZONE
duration	VARCHAR2(4000)	none

Table 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, then it is mapped to a VARCHAR2 attribute of the specified length. However, if `maxLength` is not specified in the XML Schema, then it can only be mapped to a LOB. This is sub-optimal when most of the string values are small and only a small fraction of them are large enough to need a LOB.

See Also: Table 5–5, " Mapping XML String Datatypes to SQL"

Working with Time Zones

The following XML Schema types allow for an optional time-zone indicator as part of their literal values.

- `xsd:dateTime`
- `xsd:time`
- `xsd:date`
- `xsd:gYear`
- `xsd:gMonth`
- `xsd:gDay`
- `xsd:gYearMonth`
- `xsd:gMonthDay`

By default, the schema registration maps `xsd:dateTime` and `xsd:time` to `SQL TIMESTAMP` and all the other datatypes to SQL `DATE`. The `SQL TIMESTAMP` and `DATE` types do not permit the time-zone indicator.

However, if the application needs to work with time-zone indicators, then the schema should explicitly specify the SQL type to be `TIMESTAMP WITH TIME ZONE`, using the `xdb:SQLType` attribute. This ensures that values containing time-zone indicators can be stored and retrieved correctly.

Example:

```
<element name="dob" type="xsd:dateTime"
  xdb:SQLType="TIMESTAMP WITH TIME ZONE"/>

<attribute name="endofquarter" type="xsd:gMonthDay"
  xdb:SQLType="TIMESTAMP WITH TIME ZONE"/>
```

Note: Using trailing `Z` to indicate UTC time zone.

XML Schema allows the time-zone component to be specified as `Z` to indicate UTC time zone. When a value with a trailing `Z` is stored in a `TIMESTAMP WITH TIME ZONE` column, the time zone is actually stored as `+00:00`. Thus, the retrieved value contains the trailing `+00:00` and not the original `Z`.

Example: If the value in the input XML document is `1973-02-12T13:44:32Z`, the output will look like `1973-02-12T13:44:32.000000+00:00`.

Mapping complexTypes to SQL

Using XML Schema, a `complexType` is mapped to a SQL object type as follows:

- ***XML attributes*** declared within the `complexType` are mapped to ***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, then it is mapped to a `collection` attribute in SQL. The `collection` could be a `VARRAY` value (default) or nested table if the `maintainOrder` attribute is set to false. Further, the default storage

of the VARRAY value is in Ordered Collections in Tables (OCTs) instead of LOBs. You can choose LOB storage by setting the storeAsLob attribute to true.

Specifying Attributes in a complexType XML Schema Declaration

When you have an element based on a global complexType, the SQLType and SQLSchema attributes must be specified for the complexType declaration. In addition you can optionally include the same SQLType and SQLSchema attributes within the element declaration.

The reason is that if you do not specify the SQLType for the global complexType, Oracle XML DB creates a SQLType with an internally generated name. The elements that reference this global type cannot then have a different value for SQLType. In other words, the following code is fine:

```
<xsd:complexType name="PURCHASEORDERLINEITEM_TYPEType">
  <xsd:sequence>
    <xsd:element name="LineNo" type="xsd:double"
                 xdb:SQLName="LineNo" xdb:SQLType="NUMBER"/>
    <xsd:element name="Decription" type="xsd:string"
                 xdb:SQLName="Decription" xdb:SQLType="VARCHAR2"/>
    <xsd:element name="Part" type="PURCHASEORDERPART_TYPEType"
                 xdb:SQLName="Part" />
  </xsd:sequence>
</xsd:complexType>
  <xsd:complexType name="PURCHASEORDERPART_TYPEType" xdb:SQLSchema="XMLUSER"
                   xdb:SQLType="PURCHASEORDERPART_TYPE">
     <xsd:sequence>
       <xsd:element name="Id" type="xsd:string"
                    xdb:SQLName="Id"xdb:SQLType="VARCHAR2"/>
       <xsd:element name="Quantity" type="xsd:double"
                    xdb:SQLName="Quantity" xdb:SQLType="NUMBER"/>
       <xsd:element name="cost" type="xsd:double"
                    xdb:SQLName="cost"xdb:SQLType="NUMBER"/>
     </xsd:sequence>
   </xsd:complexType>
```

The following is also fine:

```
<xsd:complexType name="PURCHASEORDERLINEITEM_TYPEType">
  <xsd:sequence>
    <xsd:element name="LineNo" type="xsd:double"
                 xdb:SQLName="LineNo" xdb:SQLType="NUMBER"/>
    <xsd:element name="Decription" type="xsd:string"
                 xdb:SQLName="Decription" xdb:SQLType="VARCHAR2"/>
    <xsd:element name="Part" type="PURCHASEORDERPART_TYPEType"
                 xdb:SQLName="Part" xdb:SQLSchema="XMLUSER"
                 xdb:SQLType="PURCHASEORDERPART_TYPE" />
  </xsd:sequence>
</xsd:complexType>
```

You Must Specify a Namespace With Remote XMLType Functions

When using XMLType functions such as extract() and existsNode() remotely for XML Schema-based views or tables, you must specify the namespace completely.

NVARCHAR and NCHAR SQLType Values are Not Supported

Oracle XML DB does not support NVARCHAR or NCHAR as a SQLType when registering an XML Schema. In other words in the XML Schema .xsd file you cannot specify that

an element should be of type NVARCHAR or NCHAR. Also, if you provide your own type you should not use these datatypes.

XPath Rewrite with XML Schema-Based Structured Storage

This section describes XPath rewrite support in Oracle XML DB and how to use it for XML Schema based structured storage.

What Is XPath Rewrite?

When the XMLType is stored in structured storage (object-relationally) using an XML Schema and queries using XPath are used, they can potentially be rewritten directly to the underlying object-relational columns. This rewrite of queries can also potentially happen when queries using XPath are issued on certain non-schema-based XMLType views.

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 XPath 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.

Note: XPath queries that get rewritten are a subset of the set of supported XPath queries. As far as possible, queries should be written so that the XPath rewrite advantages are realized.

Example 5–17 XPath 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'. Because 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';
```

Note: XMLDATA is a pseudo-attribute of XMLType that enables direct access to the underlying object column. See Chapter 4, "XMLType Operations", under "Changing the Storage Options on an XMLType Column Using XMLData".

See Also: Chapter 4, "XMLType Operations"

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.

XPath rewrite happens for XML Schema-based tables and both schema-based and non-schema based views. In this chapter we consider examples related to schema-based tables.

> **See Also:** Chapter 3, "Using Oracle XML DB", "Understanding and Optimizing XPath Rewrite" on page 3-64, for additional examples of rewrite over schema-based and non-schema based views

When Does XPath Rewrite Occur?

XPath rewrite happens for the following SQL functions:

- `extract`
- `existsNode`
- `extractValue`
- `updateXML`
- `XMLSequence`

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 5–18 SELECT Statement and XPath 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 5–19 DML Statement and XPath 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 5–20 CREATE INDEX Statement and XPath Rewrites

This example creates an index on the `Company` column, because this is stored object relationally and the XPath 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?

The rewrite of XPath expressions happen if all of the following hold true:

- The XML function or method is rewritable.

The SQL functions `extract`, `existsNode`, `extractValue`, `updateXML` and `XMLSequence` get rewritten. Other than the `existsNode()` method, none of the methods of `XMLType` get rewritten. You can however use the SQL function equivalents instead.

- The XPath construct is rewritable

  XPath constructs such as simple expressions, wildcards, and descendent axes get rewritten. The XPath may select attributes, elements or text nodes. Predicates also get rewritten to SQL predicates. Expressions involving parent axes, sibling axis, and so on are not rewritten.

- The `XMLSchema` constructs for these paths are rewritable.

  `XMLSchema` constructs such as complex types, enumerated values, lists, inherited types, and substitution groups are rewritten. Constructs such as recursive type definitions are not rewritten.

- The storage structure chosen during the schema registration is rewritable.

  Storage using the object-relational mechanism is rewritten. Storage of complex types using `CLOB`s are not rewritten

Table 5–10 lists the kinds of XPath expressions that can be translated into underlying SQL queries in this release.

Table 5–10 Sample List of XPath Expressions for Translation to Underlying SQL constructs

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 `CREATE INDEX` or `updateXML()`.
Predicates: `[Company="Oracle"]`	Predicates in the XPath are rewritten into SQL predicates. Predicates are not rewritten for `updateXML()`
List index: `lineitem[1]`	Indexes are rewritten to access the nth item in a collection. These are not rewritten for `updateXML()`.
Wildcard traversals: `/PurchaseOrder/*/Part`	If the wildcard can be translated to a unique XPath (for example, `/PurchaseOrder/Item/Part`), then it gets rewritten, provided it is not the last entry in the path expression.
Descendent axis: `/PurchaseOrder//Part`	Similar to the wildcard expression. The descendent axis gets rewritten, if it can be mapped to a unique XPath expression and the subsequent element is not involved in a recursive type definition.

Table 5–10 (Cont.) Sample List of XPath Expressions for Translation to Underlying SQL constructs

XPath Expression for Translation	Description

Oracle provided extension functions and some XPath functions
`not`, `floor`, `ceiling`, `substring`, `string-length`, `translate`
`ora:contains`
	Any function from the Oracle XML DB namespace (`http://xmlns.oracle.com/xdb`) gets rewritten into the underlying SQL function. Some XPath functions also get rewritten.

String bind variables inside predicates `'/PurchaseOrder[@Id="'|| :1 || '"]'`	XPath expressions using SQL bind variables are also rewritten provided the bind variable occurs between the concat (`||`) operators and is inside the double quotes in XPath.
Un-nest operations using XMLSequence `TABLE(XMLSequence(extract(...)))`	XMLSequence combined with Extract, when used in a `TABLE` clause is rewritten to use the underlying nested table structures.

Common XPath Constructs Supported in XPath Rewrite

The following are some of the XPath constructs that get rewritten. This is not an exhaustive list and only illustrates some of the common forms of XPath expressions that get rewritten.

- Simple XPath traversals
- Predicates and index accesses

  Oracle provided extension functions on scalar values.

  SQL Bind variables.
- Descendant axes (XML Schema-based only): Rewrites over descendant axis (`//`) are supported if:
  - There is at least one XPath child or attribute access following the `//`
  - Only one descendant of the children can potentially match the XPath child or attribute name following the `//`. If the schema indicates that multiple descendants children can potentially match, and there is no unique path the `//` can be expanded to, then no rewrite is done.
  - None of the descendants have an element of type `xsi:anyType`
  - There is no substitution group that has the same element name at any descendant.
- Wildcards (XML Schema-based only): Rewrites over wildcard axis (`/*`) are supported if:
  - There is at least one XPath child or attribute access following the `/*`
  - Only one of the grandchildren can potentially match the XPath child or attribute name following the `/*`. If the schema indicates that multiple grandchildren can potentially match, and there is no unique path the `/*` can be expanded to, then no rewrite is done.
  - None of the children or grandchildren of the node before the `/*` have an element of type `xsi:anyType`
  - There is no substitution group that has the same element name for any child of the node before the `/*`.

Unsupported XPath Constructs in XPath Rewrite

The following XPath constructs do not get rewritten:

- XPath Functions other than the ones listed earlier. Also the listed functions are rewritten only if the input is a scalar element.
- XPath Variable references.
- All axis other than child and attribute axis.
- Recursive type definitions with descendent axis.
- UNION operations.

Common XMLSchema constructs supported in XPath Rewrite

In addition to the standard XML Schema constructs such as complex types, sequences, and so on, the following additional XML Schema constructs are also supported. This is not an exhaustive list and seeks to illustrate the common schema constructs that get rewritten.

- Collections of scalar values where the scalar values are used in predicates.
- Simple type extensions containing attributes.
- Enumerated simple types.
- Boolean simple type.
- Inheritance of complex types.
- Substitution groups.

Unsupported XML Schema Constructs in XPath Rewrite

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.
- Non-coercible datatype operations, such as a Boolean added with a number

Common storage constructs supported in XPath Rewrite

All rewritable XPath expressions over object-relational storage get rewritten. In addition to that, the following storage constructs are also supported for rewrite.

Simple numeric types mapped to SQL `RAW` datatype.

Various date and time types mapped to the SQL `TIMESTAMP_WITH_TZ` datatype.

Collections stored inline, out-of-line, as OCTs and nested tables.

XML functions over schema-based and non-schema based `XMLType` views and SQL/XML views also get rewritten. See the views chapter to get detailed information regarding the rewrite.

See Also: Chapter 16, "XMLType Views"

Unsupported Storage Constructs in XPath Rewrite

The following XML Schema storage constructs are not supported. This means that if the XPath expression includes nodes with the following storage construct then the entire expression will not get rewritten:

- CLOB storage: If the XML Schema maps part of the element definitions to a SQL CLOB value, then XPath expressions traversing such elements are not supported

Is there a difference in XPath logic with rewrite?

For the most part, there is no difference between rewritten XPath queries and functionally evaluated ones. However, since XPath rewrite uses XML Schema information to turn XPath predicates into SQL predicates, comparison of non-numeric entities are different.

In XPath 1.0, the comparison operators, >, <, >=, and <= use only numeric comparison. The two sides of the operator are turned into numeric values before comparison. If either of them fail to be a numeric value, the comparison returns FALSE.

For instance, if I have a schema element such as,

```
<element name="ShipDate" type="xs:date"   xdb:SQLType="DATE"/>
```

An XPath predicate such as `[ShipDate < '2003-02-01']` will always evaluate to false with functional evaluation, since the string value `'2003-02-01'` cannot be converted to a numeric quantity. With XPath rewrite, however, this gets translated to a SQL date comparison and will evaluate to true or false depending on the value of ShipDate.

Similarly if you have a collection value compared with another collection value, the XPath 1.0 semantics dictate that the values have to be converted to a string and then compared. With Query Rewrite, the comparison will use the SQL datatype comparison rules.

To suppress this behavior, you can turn off rewrite either using query hints or session level events.

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 5–21 Registering Example Schema

Consider the following `purchaseorder` XML Schema:

```
DECLARE
  doc VARCHAR2(2000) :=
   '<schema
     targetNamespace="http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
     xmlns:po="http://xmlns.oracle.com/xdb/documentation/purchaseOrder.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="20"/>
                  </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://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd', doc);
END;
/
```

The registration creates the internal types. We can now create a table to store the XML values and also create a nested table to store the Items.

```
SQL> CREATE TABLE MYPOs OF XMLType
   2 XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
   3 ELEMENT "PurchaseOrder"
   4 VARRAY xmldata."Item" store as table item_nested;

Table created
```

Now, we insert a purchase order into this table.

```
INSERT INTO MyPos
VALUES(
 XMLType(
 '<PurchaseOrder
   xmlns="http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd
   http://xmlns.oracle.com/xdb/documentation/purchaseOrder.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>'));
```

Because 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, because 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 XPath rewrite mechanism takes into account the presence or absence of the SYS_XDBPD$ attribute and rewrites queries appropriately.

Now this table has a hidden XMLData column of type "PurchaseOrder_T" that stores the actual data.

Rewriting XPath Expressions: Mapping Types and Path Expressions

XPath expression mapping of types and topics are described in the following sections.

Schema-Based: 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, because in this case there is no need to check for the existence of the element in the SYS_XBDPD$ attribute.

For example, the XPath "/PurchaseOrder/PONum" also maps to the SQL attribute XMLData."PONum",

However, in this case, XPath rewrite also has to check for the existence of the element itself, using the SYS_XDBPD$ in the XMLData column.

Schema-Based: Mapping of Predicates

Predicates are mapped to SQL predicate expressions. As discussed earlier, since the predicates are rewritten into SQL, the comparison rules of SQL are used instead of the XPath 1.0 semantics.

Example 5–22 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;
```

Schema-Based: 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 5–23 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 5–24 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;
```

Schema-Based: Document Ordering with Collection Traversals

Most of the rewrite preserves the original document ordering. However, because 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 5–25 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 MYPOs 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.

Schema-Based: 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 value, 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 value 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 value.

Schema-Based: 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 `NULL` values during rewrite. For example the XPath expression: `"/PurchaseOrder/ShipAddress"` cannot be satisfied by any instance document conforming to the `purchaseorder.xsd` XML Schema, because the XML Schema does not allow for `ShipAddress` elements under `PurchaseOrder`. Hence this expression would map to a SQL `NULL` literal.

Schema-Based: 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 5–26 Handling Namespaces

For example, the XPath expression `/PurchaseOrder/PONum` is treated as `/a:PurchaseOrder/a:PONum` with `xmlns:a="http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"` if the SQL function does not explicitly specify the namespace prefix and mapping. In other words:

```
SELECT * FROM MYPOs p
  WHERE  ExistsNode(value(p), '/PurchaseOrder/PONum') = 1;
```

is equivalent to the query:

```
SELECT *
  FROM MYPOs p
```

```
  WHERE ExistsNode(
          value(p),
          '/PurchaseOrder/PONum',
          'xmlns="http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd')
        = 1;
```

When performing XPath 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 `purchaseorder.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://xmlns.oracle.com/xdb/documentation/purchaseOrder.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.

Schema-Based: Date Format Conversions

The default date formats are different for XML Schema and SQL. Consequently, when rewriting XPath expressions involving comparisons with dates, you must use XML formats.

Example 5–27 Date Format Conversions

For example, the expression:

```
[@PurchaseDate="2002-02-01"]
```

cannot be simply rewritten as:

```
XMLData."PurchaseDate" = "2002-02-01"
```

because the default date format for SQL is not `YYYY-MM-DD`. Hence during XPath 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","SYYYY-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.

Existential Checks for Scalar Elements and Attributes

The `existsNode` function checks for the existence of a the node targeted by the XPath while `extract` returns the targeted node. In both cases we need to do special checks for scalar elements and for attributes used in `existsNode` expressions. This is

because the SQL column value alone cannot distinguish if a scalar element or attribute is missing or is empty. In both these cases, the SQL column value is NULL. Note that these special checks are not required for intermediate (non-scalar) elements since the SQL UDT value itself will indicate the absence or emptiness of the element.

For instance, an expression of the form,

```
existsNode(value(p),'/PurchaseOrder/PONum/text()') = 1;
```

is rewritten to become

```
(p.XMLDATA."PONum" IS NOT NULL)
```

since the user is only interested in the text value of the node. If however, the expression was,

```
existsNode(value(p),'/PurchaseOrder/PONum') = 1;
```

then we need to check the SYS_XDBPD$ attribute in the parent to check if the scalar element is empty or is missing.

```
(check-node-exists(p.XMLDATA."SYS_XDBPD$","PONum") IS NOT NULL)
```

The check-node-exists operation is implemented using internal SQL operators and returns null if the element or attribute is not present in the document. In the case of extract expressions, this check needs to be done for both attributes and elements. An expression of the form,

```
Extract(value(p),'/PurchaseOrder/PONum')
```

maps to an expression like,

```
CASE WHEN check-node-exists(p.XMLDATA.SYS_XDBPD$", "PONum") IS NOT NULL
   THEN XMLElement("PONum", p.XMLDATA."PONum")
   ELSE NULL END;
```

Note: Be aware of this overhead when writing your existsNode or extract expressions. You can avoid the overhead by using the text() node in the XPath, using extractValue to get only the node's value or by turning off the DOM fidelity for the parent node.

The DOM fidelity can be turned off by setting the value of the attribute maintainDOM in the element definition to be false. In this case all empty scalar elements or attributes are treated as missing.

Rewrite of SQL Functions

Section "Rewriting XPath Expressions: Mapping Types and Path Expressions" explains the various path mappings. This section talks in detail about the differences in rewrite for some of these functions. The objective of this is to explain the overhead involved in certain types of operations using existsNode or extract which can be avoided.

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 not 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$, 'PurchaseDate') IS NOT NULL THEN 1 ELSE 0 END`
`/PurchaseOrder/PONum`	`CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD$, 'PONum') IS NOT NULL 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$, 'PurchaseDate') IS NOT NULL THEN 1 ELSE 0 END`
`/PurchaseOrder/PONum/text()`	`CASE WHEN XMLData."PONum" IS NOT NULL THEN 1 ELSE 0`
`/PurchaseOrder/Item`	`CASE WHEN EXISTS ( SELECT NULL FROM TABLE (XMLData."Item") x WHERE value(x) IS NOT NULL) THEN 1 ELSE 0 END`
`/PurchaseOrder/Item/Part`	`CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE Check_Node_Exists(x.SYS_XDBPD$, 'Part') 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`

Example 5–28 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 DOM fidelity

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`
`/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, then `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)`
`/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 5–29 Rewriting extractValue

For example, a 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;
```

Because it gets rewritten to simple scalar columns, indexes if any, on the `PONum` attribute can 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 5–30 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, because 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 of XMLSequence Function

XMLSequence can be used in conjunction with extract and the TABLE clause to unnest collection values in the XML. When used with schema-based storage, they also get rewritten to go against the underlying collection storage. For example, to get the price and part numbers of all items in a relational form, we can write a query like,

```
SQL> SELECT extractValue(value(p),'/PurchaseOrder/PONum') as ponum,
 Extractvalue(value(i) , '/Item/Part') as part,
 Extractvalue(value(i), '/Item/Price') as price
FROM MyPOs p,
     TABLE(XMLSequence(extract(value(p),'/PurchaseOrder/Item'))) i;
     PONUM PART                 PRICE
---------- -------------------- ----------
      1001 9i Doc Set           2550
      1001 8i Doc Set           350
```

In this example, the extract function returns a fragment containing the list of Item elements and the XMLSequence function then converts the fragment into a collection of XMLType values one for each Item element. The TABLE clause converts the elements of the collection into rows of XMLType. The returned XML from the TABLE clause is used to extract out the Part and the Price.

XPath rewrite will rewrite the extract and the XMLSequence function so that it will become a simple select from the Item_nested nested table.

```
SQL> EXPLAIN PLAN
FOR SELECT extractValue(value(p),'/PurchaseOrder/PONum') AS ponum,
 extractValue(value(i) , '/Item/Part') AS part,
 extractValue(value(i), '/Item/Price') AS price
FROM MyPOs p,
     TABLE(XMLSequence(extract(value(p),'/PurchaseOrder/Item'))) i;

Explained

SQL> @utlxpls.sql

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
| Id  | Operation                     | Name         |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |
|   1 |  NESTED LOOPS                 |              |
|   2 |   TABLE ACCESS FULL           | ITEM_NESTED  |
|   3 |   TABLE ACCESS BY INDEX ROWID | MYPOS        |
|*  4 |    INDEX UNIQUE SCAN          | SYS_C002973  |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id)
---------------------------------------------------
   4 - access("NESTED_TABLE_ID"="P"."SYS_NC0001100012$")
```

The EXPLAIN PLAN output shows that the optimizer is able to use a simple nested loops join between the Item_nested nested table and MyPOs table. You can also query the Item values further and create appropriate indexes on the nested table to speed up such queries.

For example, if we want to search on the Price to get all the expensive items, we could create an index on the Price column on the nested table. The following EXPLAIN

PLAN uses the Price index to get the list of items and then joins back with the MYPOs table to get the PONum value.

```
SQL> CREATE INDEX price_index ON item_nested ("Price");

Index created.

SQL> EXPLAIN PLAN FOR
     SELECT extractValue(value(p),'/PurchaseOrder/PONum') AS ponum,
            extractValue(value(i) , '/Item/Part') AS part,
            extractValue(value(i), '/Item/Price') AS price
     FROM  MyPOs p,
          TABLE(XMLSequence(extract(value(p),'/PurchaseOrder/Item'))) i
     WHERE extractValue(value(i),'/Item/Price') > 2000;

Explained.

SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
| Id  | Operation                     | Name          |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |
|   1 |  NESTED LOOPS                 |               |
|   2 |   TABLE ACCESS BY INDEX ROWID | ITEM_NESTED   |
|*  3 |    INDEX RANGE SCAN           | PRICE_INDEX   |
|   4 |   TABLE ACCESS BY INDEX ROWID | MYPOS         |
|*  5 |    INDEX UNIQUE SCAN          | SYS_C002973   |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("ITEM_NESTED"."Price">2000)
   5 - access("NESTED_TABLE_ID"="P"."SYS_NC0001100012$")
```

Rewrite for extract()

The extract() function 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 DOM fidelity

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).

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$, 'PurchaseDate') IS NOT NULL THEN XMLElement("", XMLData."PurchaseDate") ELSE NULL END;
/PurchaseOrder/PONum	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD$, 'PONum') IS NOT NULL THEN XMLElement("PONum", XMLData."PONum") ELSE NULL END

Table 5–15 (Cont.) XPath Mapping for extract() with Document Ordering Preserved

XPath	Maps to
/PurchaseOrder[PONum=2100]	(SELECT XMLForest(XMLData as "PurchaseOrder") from dual WHERE XMLData."PONum" = 2100)
/PurchaseOrder[PONum = 2100]/@PurchaseDate	(SELECT CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD$, 'PurchaseDate') IS NOT NULL THEN XMLElement("", XMLData."PurchaseDate") ELSE NULL END FROM Dual WHERE XMLData."PONum" = 2100)
/PurchaseOrder/PONum/text()	XMLElement("", XMLData."PONum")
/PurchaseOrder/Item	(SELECT XMLAgg(XMLForest(value(p) as "Item")) FROM TABLE (XMLData."Item") p)
/PurchaseOrder/Item/Part	(SELECT XMLAgg(CASE WHEN CHECK_Node_Exists(p.SYS_XDBPD$, 'Part') IS NOT NULL THEN XMLForest(p."Part" As "Part") ELSE NULL END) FROM TABLE(XMLData."Item") p)
/PurchaseOrder/Item/Part/text()	(SELECT XMLAgg(XMLElement("", p."Part")) FROM TABLE(XMLData."Item") p)

Example 5–31 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   PurchaseOrder_table p;
```

would become:

```
SELECT (SELECT CASE WHEN check_node_exists(p.XMLData.SYS_XDBPD$, 'PONum')
                        IS NOT NULL
                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 PurchaseOrder_table p;
```

Extract mapping without DOM fidelity

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 "PurchaseDate")
/PurchaseOrder/PONum	XMLForest(XMLData."PONum" AS "PONum")
/PurchaseOrder[PONum = 2100]	(SELECT XMLForest(XMLData AS "PurchaseOrder") FROM Dual WHERE XMLData."PONum" = 2100)

Table 5–16 (Cont.) XPath Mapping for extract() Without Document Ordering Preserved

XPath	Equivalent to
/PurchaseOrder[PONum = 2100]/@PurchaseDate	(SELECT XMLForest(XMLData."PurchaseDate" AS "PurchaseDate "") FROM DUAL WHERE XMLData."PONum" = 2100)
/PurchaseOrder/PONum/text()	XMLForest(XMLData.PONum AS "")
/PurchaseOrder/Item	(SELECT XMLAgg(XMLForest(value(p) as "Item") FROM TABLE (XMLData."Item") p)
/PurchaseOrder/Item/Part	(SELECT XMLAgg(XMLForest(p."Part" AS "Part") FROM TABLE (XMLData."Item") p)
/PurchaseOrder/Item/Part/text()	(SELECT XMLAgg(XMLForest(p. "Part" AS "Part")) FROM TABLE (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 PurchaseOrder_table p SET value(p) = updatexml(value(p),...);
  ```

- The `XMLType` column must have been stored object relationally using Oracle XML DB 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 PurchaseOrder_table 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 PurchaseOrder_table p SET value(p) =
    updatexml(value(p),'/PurchaseOrder/@PurchaseDate','2002-01-02',
                       '/PurchaseOrder/PONum/text()', 2200);
```

becomes:

```
UPDATE PurchaseOrder_table p
  SET p.XMLData."PurchaseDate" = TO_DATE('2002-01-02','SYYYY-MM-DD'),
      p.XMLData."PONum" = 2100;
```

DATE Conversions

Date datatypes such as `DATE`, `gMONTH`, and `gDATE` have different format in XML Schemas and SQL. In such cases, if the `updateXML()` has a string value for these columns, then 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.

Diagnosing XPath Rewrite

To determine if your XPath expressions are getting rewritten, you can use one of the following techniques:

Using Explain Plans

This section shows how you can use the explain plan to examine the query plans after rewrite. See Chapter 3, "Using Oracle XML DB", "Understanding and Optimizing XPath Rewrite" on page 3-64 for examples on how to use `EXPLAIN PLAN` to optimize XPath rewrite.

With the explained plan, if the plan does not pick applicable indexes and shows the presence of the SQL function (such as `existsNode` or `extract`), then you know that the rewrite has not occurred. You can then use the events described later to understand why the rewrite did not happen.

For example, using the `MYPOs` table, we can see the use of explain plans. We create an index on the `Company` element of `PurchaseOrder` to show how the plans differ.

```
SQL> CREATE INDEX company_index ON MyPOs e
       (extractValue(object_value,'/PurchaseOrder/Company'));

Index created.

SQL> EXPLAIN PLAN FOR
       SELECT  extractValue(value(p),'/PurchaseOrder/PONum')
         FROM MyPOs p
         WHERE existsNode(value(p),'/PurchaseOrder[Company="Oracle"]')=1;

Explained.

SQL> @utlxpls.sql

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows | Bytes | Cost |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |      |       |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| MYPOS         |      |       |      |
|*  2 |   INDEX RANGE SCAN          | COMPANY_INDEX |      |       |      |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("MYPOS"."SYS_NC00010$"='Oracle')
```

In this explained plan, you can see that the predicate uses internal columns and picks up the index on the `Company` element. This shows clearly that the query has been rewritten to the underlying relational columns.

In the following query, we are trying to perform an arithmetic operation on the `Company` element which is a string type. This is not rewritten and hence the explain

plan shows that the predicate contains the original existsNode expression. Also, since the predicate is not rewritten, a full table scan instead of an index range scan is used.

```
SQL> EXPLAIN PLAN FOR
       SELECT  extractValue(value(p),'/PurchaseOrder/PONum')
         FROM MyPOs p
         WHERE existsNode(value(p),
                 '/PurchaseOrder[Company+PONum="Oracle"]') = 1;

Explained.

SQL> @utlxpls.sql

PLAN_TABLE_OUTPUT
------------------------------------------------------------
| Id  | Operation          | Name
------------------------------------------------------------
|   0 | SELECT STATEMENT   |
|*  1 |  FILTER            |
|   2 |   TABLE ACCESS FULL| MYPOS
|*  3 |   TABLE ACCESS FULL| ITEM_NESTED
------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(EXISTSNODE(SYS_MAKEXML('C6DB2B4A1A3B0
               6CDE034080020E5CF39',2300,"MYPOS"."XMLEXTRA",
               "MYPOS"."XMLDATA"),
      '/PurchaseOrder[Company+PONum="Oracle"]')=1)
   3 - filter("NESTED_TABLE_ID"=:B1)
```

Using Events

Events can be set in the initialization file or can be set for each session using the ALTER SESSION statement. The XML events can be used to turn off functional evaluation, turn off the query rewrite mechanism and to print diagnostic traces.

Turning off Functional Evaluation (Event 19021) By turning on this event, you can raise an error whenever any of the XML functions are not rewritten and get evaluated. The error ORA-19022 - XML XPath functions are disabled will be raised when such functions execute. This event can also be used to selectively turn off functional evaluation of functions. Table 5–17 lists the various levels and the corresponding behavior.

Table 5–17 Event Levels and Behaviors

Event	Behavior
Level 0x1	Turn off functional evaluation of all XML functions.
Level 0x2	Turn off functional evaluation of extract.
Level 0x4	Turn off functional evaluation of existsNode.
Level 0x8	Turn off functional evaluation of transform.
Level 0x10	Turn off functional evaluation of extractValue.
Level 0x20	Turn off the functional evaluation of UpdateXML.
Level 0x200	Turn off functional evaluation of XMLSequence

For example,

```
ALTER SESSION SET EVENTS '19021 trace name context forever, level 1';
```

would turn off the functional evaluation of all the XML operators listed earlier. Hence when you perform the query shown earlier that does not get rewritten, you will get an error during the execution of the query.

```
SQL> SELECT value(p) FROM MyPOs p
     WHERE Existsnode(value(p),
                 '/PurchaseOrder[Company+PONum="Oracle"]')=1 ;

ERROR:
ORA-19022: XML XPath functions are disabled
```

Tracing reasons for non-rewrite

Event `19027` with level `8192` (0x2000) can be used to dump traces that indicate the reason that a particular XML function is not rewritten. For example, to check why the query described earlier, did not rewrite, we can set the event and run an explain plan:

```
SQL> alter session set events '19027 trace name context forever, level 8192';

Session altered.

SQL> EXPLAIN PLAN FOR
     SELECT value(p) from MyPOs p
     WHERE Existsnode(value(p),'/PurchaseOrder[Company+100="Oracle"]')=1;

Explained.
```

This writes the following the Oracle trace file explaining that the rewrite for the XPath did not occur since there were non-numeric inputs to an arithmetic function.

```
NO REWRITE
        XPath ==> /PurchaseOrder[Company+PONum = "Oracle" ]
        Reason ==> non numeric inputs to arith{2}{4}
```

6

XML Schema Storage and Query: Advanced Topics

This chapter describes more advanced techniques for storing structured XML schema-based `XMLType` objects. It explains `simpleType` and `complexType` mapping from XML to SQL storage types and how querying on `XMLType` tables and columns based on this mapping are optimized using query rewrite techniques. It discusses the mechanism for generating XML schema from existing object types.

This chapter contains these topics:

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.

  See Also: "Creating XMLType Tables and Columns Based on XML Schema" on page 5-18

Example 6–1 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>
```

Adding Unique Constraints to An Attribute's Elements

How can you, after creating an `XMLType` table based on an XML schema, add a unique constraint to an attribute's elements? You may, for example, want to create a unique key based on an attribute of an element that repeats itself (therefore creating a collection type).

To create constraints on elements that can occur more than once within the instance document, you must store the VARRAY as a table. This is also known as **Ordered Collections in Tables** (OCT). You can then create constraints on the OCT. Example 6–2 shows how the attribute `No` of `<PhoneNumber>` can appear more than once, and how a unique constraint can be added to ensure that the same number cannot be repeated within the same instance document.

Note: This constraint applies to each collection, and not across all instances. This is achieved by creating a concatenated index with the collection id column. To apply the constraint across all collections of all instance documents, simply omit the collection id column.

Example 6–2 Adding Unique Constraints to an Attribute's Element

```
BEGIN DBMS_XMLSCHEMA.registerschema('emp.xsd',
   '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
               xmlns:xdb="http://xmlns.oracle.com/xdb">
      <xs:element name="Employee" xdb:SQLType="EMP_TYPE">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="EmployeeId" type="xs:positiveInteger"/>
            <xs:element name="PhoneNumber" maxOccurs="10">
              <xs:complexType>
                <xs:attribute name="No" type="xs:integer"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>',
   TRUE,
   TRUE,
   FALSE,
   FALSE);
END;
/

PL/SQL procedure successfully completed.

CREATE table emp_tab OF XMLType
  XMLSCHEMA "emp.xsd" ELEMENT "Employee"
  VARRAY xmldata."PhoneNumber" STORE AS table phone_tab;

Table created.

ALTER TABLE phone_tab ADD unique(nested_table_id, "No");

Table altered.

CREATE TABLE po_xtab OF XMLType; -- The default is CLOB based storage.
INSERT INTO emp_tab
```

```
  VALUES(XMLType('<Employee>
                    <EmployeeId>1234</EmployeeId>
                    <PhoneNumber No="1234"/>
                    <PhoneNumber No="2345"/>
                  </Employee>').createschemabasedxml('emp.xsd'));

1 row created.

INSERT INTO emp_tab
  VALUES(xmltype('<Employee>
                    <EmployeeId>3456</EmployeeId>
                    <PhoneNumber No="4444"/>
                    <PhoneNumber No="4444"/>
                  </Employee>').createschemabasedxml('emp.xsd'));
```

This returns the expected result:

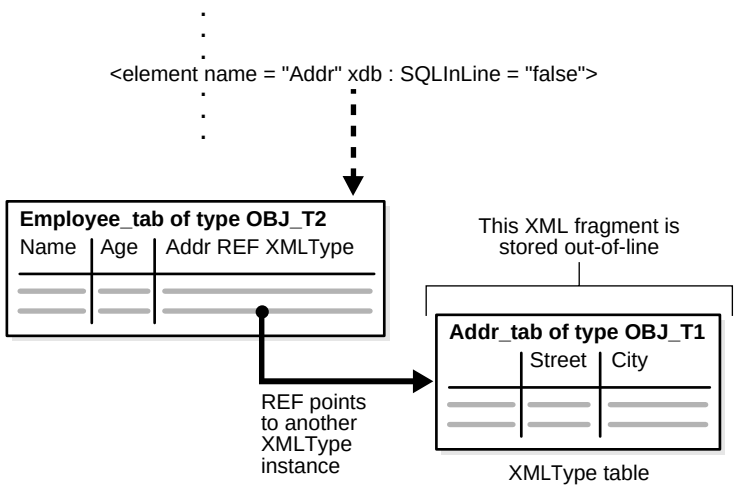
```
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C002136) violated
```

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 6–1 illustrates the mapping of a `complexType` to SQL for out-of-line storage.

Figure 6–1 Mapping complexType to SQL for Out-of-Line Storage



Example 6–3 Oracle XML DB XML Schema: complexType Mapping - Setting SQLInLine Attribute to False for Out-of-Line Storage

In this example, attribute `xdb:SQLInLine` of element `Addr` 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. Table 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="EmpType" xdb:SQLType="EMP_T">
         <sequence>
           <element name="Name" type="string"/>
           <element name="Age" type="decimal"/>
           <element name="Addr"
                    xdb:SQLInline="false"
                    xdb:defaultTable="ADDR_TAB">
             <complexType xdb:SQLType="ADDR_T">
               <sequence>
                 <element name="Street" type="string"/>
                 <element name="City" type="string"/>
               </sequence>
             </complexType>
           </element>
         </sequence>
       </complexType>
       <element name="Employee" type="emp:EmpType"
                xdb:defaultTable="EMP_TAB"/>
     </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema('emp.xsd', doc);
END;
/
```

On registering this XML schema, Oracle XML DB generates the following types and XMLType tables:

```
CREATE TYPE ADDR_T AS OBJECT (SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                              Street VARCHAR2(4000),
                              City VARCHAR2(4000));
CREATE TYPE EMP_T AS OBJECT (SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                             Name VARCHAR2(4000),
                             Age NUMBER,
                             Addr REF XMLType) NOT FINAL;
```

Two XMLType tables are also created: EMP_TAB and ADDR_TAB. Table EMP_TAB holds all the employees and contains an object reference to point to the address values that are stored only in table ADDR_TAB.

The advantage of this model is that it lets you query the out-of-line table (ADDR_TAB in this case) directly, to look up the address information. For example, if you want to get the distinct city information for all the employees, you can query the table ADDR_TAB directly.

```
INSERT INTO EMP_TAB
  VALUES
    (XMLType('<x:Employee
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:x="http://www.oracle.com/emp.xsd"
                xsi:schemaLocation="http://www.oracle.com/emp.xsd emp.xsd">
                <Name>Jason Miller</Name>
                <Age>22</Age>
```

```
                <Addr>
                  <Street>Julian Street</Street>
                  <City>San Francisco</City>
                </Addr>
              </x:Employee>'));
INSERT INTO EMP_TAB
  VALUES (XMLType('<x:Employee
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xmlns:x="http://www.oracle.com/emp.xsd"
                     xsi:schemaLocation="http://www.oracle.com/emp.xsd emp.xsd">
                     <Name>Jack Simon</Name>
                     <Age>23</Age>
                     <Addr>
                       <Street>Mable Street</Street>
                       <City>Redwood City</City>
                     </Addr>
                   </x:Employee>'));

REM The ADDR_TAB stores the addresses and can be queried directly
SELECT DISTINCT Extractvalue(object_value,'/Addr/City') AS city FROM ADDR_TAB;

CITY
------------------
Redwood City
San Francisco
```

The disadvantage of this storage is that to get the whole `Employee` element you need to look up an additional table for the address.

Query Rewrite For Out-Of-Line Tables

XPath expressions that involve elements stored out of line get rewritten. In this case, the query involves a join with the out-of-line table. For example, the following `EXPLAIN PLAN` shows how a query involving `Employee` and `Addr` elements is handled.

```
EXPLAIN PLAN FOR
  SELECT Extractvalue(object_value,
                      '/x:Employee/Name',
                      'xmlns:x="http://www.oracle.com/emp.xsd"')
    FROM emp_tab x
    WHERE Existsnode(value(x),
                     '/x:Employee/Addr[City="San Francisco"]',
                     'xmlns:x="http://www.oracle.com/emp.xsd"')=1;

SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
-----------------------------------------------------
| Id | Operation                   | Name        |
-----------------------------------------------------
|  0 | SELECT STATEMENT            |
|* 1 | FILTER                      |
|  2 | TABLE ACCESS FULL           | EMP_TAB     |
|* 3 | TABLE ACCESS BY INDEX ROWID | ADDR_TAB    |
|* 4 | INDEX UNIQUE SCAN           | SYS_C003111 |
-----------------------------------------------------
Predicate Information (identified by operation id):
-----------------------------------------------------
   1 - filter(EXISTS(SELECT 0
```

```
                      FROM "SCOTT"."ADDR_TAB" "SYS_ALIAS_1"
                      WHERE "SYS_ALIAS_1"."SYS_NC_OID$"=:B1
                        AND "SYS_ALIAS_1"."SYS_NC00009$"='San Francisco'))
   3 - filter("SYS_ALIAS_1"."SYS_NC00009$"='San Francisco')
   4 - access("SYS_ALIAS_1"."SYS_NC_OID$"=:B1)
```

In this example, the XPath expression was rewritten to an `Exists` subquery that queries table `ADDR_TAB` and joins it with table `EMP_TAB` using the object identifier column in table `ADDR_TAB`. The optimizer uses a full table scan to scan all the rows in the employee table and uses the unique index on the `SYS_NC_OID$` column in the address table to look up the address.

If there are a lot of entries in the `ADDR_TAB`, then you can make this query more efficient by creating an index on the `City` column.

```
CREATE INDEX addr_city_idx
  ON ADDR_TAB (extractvalue(object_value,'/Addr/City'));
```

The `EXPLAIN PLAN` for the previous statement now uses the `addr_city_idx` index.

```
SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
-----------------------------------------------------
| Id | Operation                  | Name          |
-----------------------------------------------------
|  0 | SELECT STATEMENT           |
|* 1 | FILTER                     |
|  2 | TABLE ACCESS FULL          | EMP_TAB       |
|* 3 | TABLE ACCESS BY INDEX ROWID| ADDR_TAB      |
|* 4 | INDEX RANGE SCAN           | ADDR_CITY_IDX |
-----------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(EXISTS (SELECT 0
                      FROM "SCOTT"."ADDR_TAB" "SYS_ALIAS_1"
                      WHERE "SYS_ALIAS_1"."SYS_NC_OID$"=:B1
                        AND "SYS_ALIAS_1"."SYS_NC00009$"='San Francisco'))
   3 - access("SYS_ALIAS_1"."SYS_NC_OID$"=:B1)
   4 - filter("SYS_ALIAS_1"."SYS_NC00009$"='San Francisco')
```

Storing Collections in Out-Of-Line Tables

You can also map list items to be stored out of line. In this case, instead of a single `REF` column, the parent element will contain a `VARRAY` of `REF` values that point to the members of the collection. For example, consider the case where we have a list of addresses for each employee and map that to an out of line storage.

```
DECLARE
doc VARCHAR2(3000) :=
  '<schema xmlns="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.oracle.com/emp.xsd"
           xmlns:emp="http://www.oracle.com/emp.xsd"
           xmlns:xdb="http://xmlns.oracle.com/xdb">
     <complexType name="EmpType" xdb:SQLType="EMP_T2">
       <sequence>
         <element name="Name" type="string"/>
         <element name="Age" type="decimal"/>
         <element name="Addr" xdb:SQLInline="false"
                  maxOccurs="unbounded" xdb:defaultTable="ADDR_TAB2">
```

```
          <complexType xdb:SQLType="ADDR_T2">
            <sequence>
              <element name="Street" type="string"/>
              <element name="City" type="string"/>
            </sequence>
          </complexType>
        </element>
       </sequence>
     </complexType>
     <element name="Employee" type="emp:EmpType"
              xdb:defaultTable="EMP_TAB2"/>
   </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema('emprefs.xsd', doc);
END;
/
```

On registering this XML schema, Oracle XML DB now generates the following types and XMLType tables:

```
CREATE TYPE ADDR_T2 AS OBJECT (SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                               Street VARCHAR2(4000),
                               City VARCHAR2(4000));
CREATE TYPE EMP_T2  AS OBJECT (SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                               Name VARCHAR2(4000),
                               Age NUMBER,
                               Addr XDB.XDB$XMLTYPE_REF_LIST_T) NOT FINAL;
```

The employee type (EMP_T2) now contains a VARRAY of REF values to address instead of a single REF attribute as in the previous XMLSchema. By default this VARRAY of REF values is stored in-line in the employee (EMP_TAB2) table. This storage is ideal for the cases where the more selective predicates in the query are on the employee table. This is because storing the VARRAY in line effectively forces any query involving the two tables to always be driven off of the employee table as there is no way to efficiently join back from the address table. The following example shows the plan for a query that selects the names of all San Francisco-based employees, and the streets in which they live, in an unnested form.

```
EXPLAIN PLAN FOR
  SELECT Extractvalue(value(e), '/x:Employee/Name',
                      'xmlns:x="http://www.oracle.com/emp.xsd"') AS name,
         Extractvalue(value(a), '/Addr/Street') AS street
    FROM
      EMP_TAB2 e,
      TABLE(XMLSequence(Extract(value(e),
                                '/x:Employee/Addr',
                                'xmlns:x="http://www.oracle.com/emp.xsd"'))) a
    WHERE Extractvalue(value(a),'/Addr/City') = 'San Francisco';

Explained.

SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
| Id | Operation                      | Name        |
------------------------------------------------------------------------------
|  0 | SELECT STATEMENT               |             |
|  1 | NESTED LOOPS                   |             |
|  2 | NESTED LOOPS                   |             |
```

```
|  3 | TABLE ACCESS FULL                | EMP_TAB2    |
|  4 | COLLECTION ITERATOR PICKLER FETCH|             |
|* 5 | TABLE ACCESS BY INDEX ROWID      | ADDR_TAB2   |
|* 6 | INDEX UNIQUE SCAN                | SYS_C003016 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - filter("SYS_ALIAS_2"."SYS_NC00009$"='San Francisco')
   6 - access(VALUE(KOKBF$)="SYS_ALIAS_2"."SYS_NC_OID$")
```

If there are several `Addr` elements for each employee, then building an index on the `City` element in table `ADDR_TAB2` will help speed up the previous query.

Intermediate table for storing the list of references

In cases where the number of employees is large, a full table scan of the `EMP_TAB2` table is too expensive. The correct plan is to query the address table on the `City` element and then join back with the employee table.

This can be achieved by storing the `VARRAY` of `REF` values as a separate table, and creating an index on the `REF` values in that table. This would allow Oracle Database to query the address table, get an object reference (`REF`) to the relevant row, join it with the intermediate table storing the list of `REF` values and then join that table back with the employee table.

The intermediate table can be created by setting the attribute `xdb:storeVarrayAsTable` to `TRUE` in the XMLSchema definition. This forces the schema registration to store all `VARRAY` values as separate tables.

```
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"
             xdb:storeVarrayAsTable="true">
       <complexType name="EmpType" xdb:SQLType="EMP_T3">
         <sequence>
           <element name="Name" type="string"/>
           <element name="Age" type="decimal"/>
           <element name="Addr" xdb:SQLInline="false"
                    maxOccurs="unbounded" xdb:defaultTable="ADDR_TAB3">
             <complexType xdb:SQLType="ADDR_T3">
               <sequence>
                 <element name="Street" type="string"/>
                 <element name="City" type="string"/>
               </sequence>
             </complexType>
           </element>
         </sequence>
       </complexType>
       <element name="Employee" type="emp:EmpType"
                xdb:defaultTable="EMP_TAB3"/>
     </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema('empreftab.xsd', doc);
END;
/
```

In addition to creating the types ADDR_T3 and EMP_T3 and the tables EMP_TAB3 and ADDR_TAB3, the schema registration also creates the intermediate table that stores the list of REF values.

```
SELECT table_name
  FROM user_nested_tables
  WHERE parent_table_name='EMP_TAB3';

TABLE_NAME
------------------------------
SYS_NTyjtiinHKYuTgNAgAIOXPOQ==

REM Rename nested table to more meaningful name
RENAME "SYS_NTyjtiinHKYuTgNAgAIOXPOQ==" TO EMP_TAB3_REFLIST;

DESCRIBE EMP_TAB3_REFLIST

Name          Null? Type
------------- ----  --------------------------
COLUMN_VALUE        REF OF XMLTYPE
```

We can create an index on the REF value in this table. Indexes on REF values can be only be created if the REF is scoped or has a referential constraint. Creating a scope on a REF column implies that the REF only stores pointers to objects in a particular table. In this example, the REF values in the EMP_TAB3_REFLIST will only point to objects in the ADDR_TAB3 table, so we can create a scope constraint and an index on the REF column, as follows.

```
ALTER TABLE emp_tab3_reflist ADD SCOPE FOR (column_value) IS addr_tab3;
CREATE INDEX reflist_idx ON emp_tab3_reflist (column_value);

REM Also create an index on the city element
CREATE INDEX city_idx ON ADDR_TAB3 p (extractvalue(value(p),'/Addr/City'));
```

Now, the EXPLAIN PLAN for the earlier query shows the use of the city_idx index, followed by a join with tables EMP_TAB3_REFLIST and EMP_TAB3.

```
EXPLAIN PLAN FOR
  SELECT Extractvalue(value(e), '/x:Employee/Name',
                      'xmlns:x="http://www.oracle.com/emp.xsd"') AS name,
         Extractvalue(value(a), '/Addr/Street') AS street
    FROM EMP_TAB3 e,
         TABLE(XMLSequence(Extract(value(e), '/x:Employee/Addr',
                                   'xmlns:x="http://www.oracle.com/emp.xsd"'))) a
    WHERE Extractvalue(value(a),'/Addr/City')='San Francisco';

SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
| Id | Operation                    | Name        |
------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |             |
|  1 | NESTED LOOPS                 |             |
|  2 | NESTED LOOPS                 |             |
|  3 | TABLE ACCESS BY INDEX ROWID | ADDR_TAB3   |
|* 4 | INDEX RANGE SCAN             | CITY_IDX    |
|* 5 | INDEX RANGE SCAN             | REFLIST_IDX |
|  6 | TABLE ACCESS BY INDEX ROWID | EMP_TAB3    |
|* 7 | INDEX UNIQUE SCAN            | SYS_C003018 |
------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("SYS_ALIAS_2"."SYS_NC00009$"='San Francisco')
   5 - access("EMP_TAB3_REFLIST"."COLUMN_VALUE"="SYS_ALIAS_2"."SYS_NC_OID$")
   7 - access("NESTED_TABLE_ID"="E"."SYS_NC0001100012$")
```

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.

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 the 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>/<schemaURL-minus-protocol>
```

Example 6–4 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
```

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 6–2. 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, and recomposition overheads.

Example 6–5 Oracle XML DB XML Schema: complexType Mapping XML Fragments to LOBs

In the following example, the XML schema specifies that the XML fragment element `Addr` uses the attribute `SQLType="CLOB"`:

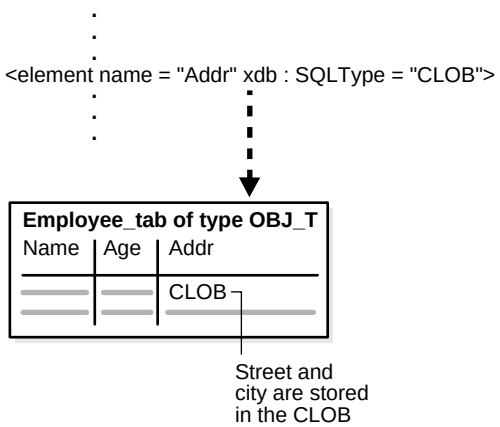
```
DECLARE
  doc VARCHAR2(3000) :=
```

```
    '<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 6–2 Mapping complexType XML Fragments to Character Large Objects (CLOBs)



Oracle XML DB complexType Extensions and Restrictions

In XML schema, `complexType` values are declared based on `complexContent` and `simpleContent`.

- `simpleContent` is declared as an extension of `simpleType`.
- `complexContent` is declared as one of the following:
  - Base type

- `complexType` extension
- `complexType` restriction

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 6–6 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 *non-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 6–7 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;
```

Because 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 type.

Example 6–8 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 6–9 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;
```

This 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));
```

Inserting New Instances into XMLType Columns

New instances can be inserted into an `XMLType` column as follows:

```
INSERT INTO MyPOs VALUES
     (XMLType('<PurchaseOrder>...</PurchaseOrder>'));
```

Examining Type Information in Oracle XML DB

Oracle XML DB supports schema-based XML, wherein elements and attributes in the XML data have XML Schema type information associated with them. However, XPath 1.0 is not aware of type information. Oracle XML DB extends XPath 1.0 with the following functions to support examining of type information:

- `instanceof()` in the namespace `http://xmlns.oracle.com/xdb`
- `instanceof-only()` in the namespace `http://xmlns.oracle.com/xdb`

An element is an instance of a specified XML Schema type if its type is the same as the specified type, or is a subtype of the specified type. A subtype of type T in the context of XML Schema refers to a type that extends or restricts T, or extends or restricts another subtype of T.

ora:instanceof() and ora:instanceof-only()

XPath queries can use `instanceof-only()` to restrict the result set to nodes of a certain type, and `instanceof()` to restrict it to nodes of a certain type and its subtypes for schema-based XML data. For non-schema-based data, elements and attributes do not have type information. Therefore, the functions return FALSE for non-schema-based XML data.

The semantics of the functions are as follows:

- **`ora:instanceof-only()`**: Function `ora:instanceof-only()` has the following signature:

  ```
  boolean instanceof-only(nodeset nodeset-expr,
                          string typename [, string schema-url])
  ```

  On schema-based data, the XPath function `instanceof-only` evaluates the `xpath-expr` corresponding to `nodeset-expr` and determines the XML Schema type for each of the resultant node(s). Note that the xpath expression `nodeset-expr` is typically a relative XPath expression. If the type of any of the nodes exactly matches the name `typename`, optionally qualified with a

namespace prefix, then the XPath function returns TRUE. Otherwise, the function returns FALSE. The XPath function returns FALSE for non-schema-based data.

Example 6–10 Using ora:instanceof-only

The following query selects Name attributes of AE children of the element Person that are of type PersonType (subtypes of PersonType are not matched).

```
SELECT extract(value(p),
               '/p9:Person[ora:instanceof-only(AE,"p9:PersonType")]/AE/Name',
               'xmlns:p9="person9.xsd" xmlns:ora="http://xmlns.oracle.com/xdb"')
  FROM PO_Table p;
```

- **ora:instanceof():** Function ora:instanceof() has the following signature:

  ```
  boolean instanceof(nodeset nodeset-expr,
                     string typename [, string schema-url])
  ```

  On schema-based data, the XPath function instanceof evaluates the xpath-expr corresponding to nodeset-expr and determines the XML Schema type for each of the resultant node(s). Note that the xpath expression nodeset-expr is typically a relative xpath expression. If the type of any of the nodes exactly matches the name typename, optionally qualified with a namespace prefix, then the XPath function returns TRUE. Otherwise, the function returns FALSE. The XPath function returns FALSE for non-schema-based data.

  For each node that matches the xpath expression nodeset-expr, the qualified name of the type of the node is determined. For ora:instanceofonly(), if the name and namespace of the type exactly matches the specified typename, the function returns TRUE. For ora:instanceof(), if the name and namespace of the type of the node or one of its supertypes exactly matches the specified typename, the function return TRUE. Otherwise, processing continues with the next node in the node set.

  The schema-url parameter can additionally be specified to indicate the schema location URL for the type to be matched. If the schema-url parameter is not specified, then the schema location URl is not checked. If the parameter is specified, then the schema in which the type of the node is declared must match the schema-url parameter.

Example 6–11 Using ora:instanceof

The following query selects Name attributes of AE children of the element Person that are of type PersonType or one of its subtypes.

```
SELECT extract(value(p),
               '/p9:Person[ora:instanceof(AE,"p9:PersonType")]/AE/Name',
               'xmlns:p9="person9.xsd" xmlns:ora="http://xmlns.oracle.com/xdb"')
  FROM PO_Table p;
```

Using ora:instanceof in a heterogeneous schema storage: One of the use cases for the schema location parameter is the heterogeneous XML Schema scenario. I f your scenario involves a schema-based table, consider omitting the schema location parameter. Heterogeneous XML Schema-based data can be present in a single table.

Consider a non-schema-based table of XMLType. Each row in the table is an XML document. Suppose that the contents of each XML document is XML data for which schema information has been specified. If the data in the table is converted

to schema-based data through a subsequent operation, then the rows in the table could pertain to different schemas. In such a case, you can specify not only the name and the namespace of the type to be matched, but also the schema location URL.

Example 6–12 Using ora:instanceof in a Heterogeneous Schema Storage

In the non-schema-based table `non_sch_p_tab`, the following query matches elements of type `PersonType` that pertain to schema `person9.xsd`.

```
SELECT extract(
         createschemabased(
           value(p)),
           '/p9:Person/AE[ora:instanceof(.,"p9:PersonType", "person9.xsd")]',
           'xmlns:p9="person9.xsd" xmlns:ora="http://xmlns.oracle.com/xdb"')
  FROM "non_sch_p_tab" p;
```

Working With Circular and Cyclical Dependencies

The W3C XML Schema Recommendation allows `complexTypes` and global elements to contain recursive references. For example, a `complexType` definition may contain an element based on the `complexType` itself being defined, or a global element can contain a reference to itself. In both cases the reference can be direct or indirect. This kind of structure allows for instance documents where the element in question can appear an infinite number of times in a recursive hierarchy.

Example 6–13 An XML Schema With Circular Dependency

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="person" type="personType" xdb:defaultTable="PERSON_TABLE"/>
  <xs:complexType name="personType" xdb:SQLType="PERSON_T">
    <xs:sequence>
      <xs:element name="decendant" type="personType" minOccurs="0"
                  maxOccurs="unbounded" xdb:SQLName="DESCENDANT"
                  xdb:defautlTable="DESCENDANT_TABLE"/>
    </xs:sequence>
    <xs:attribute name="personName" use="required" xdb:SQLName="PERSON_NAME">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:maxLength value="20"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:schema>
```

The XML schema shown in Example 6–13 includes a circular dependency. The `complexType personType` consists of a `personName` attribute and a collection of descendant elements. The descendant element is defined as being of `personType`.

For Circular Dependency Set GenTables Parameter to TRUE

Oracle XML DB supports XML schemas that define this kind of structure. To break the cycle implicit in this kind of structure, recursive elements are stored as rows in a

separate XMLType table. The table used to manage these elements is an XMLType table, created during the XML schema registration process.

Consequently it is important to ensure that the genTables parameter is always set to TRUE when registering an XML schema that defines this kind of structure. The name of the table used to store the recursive elements can be specified by adding an xdb:defaultTable annotation to the XML schema.

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 6–14 XML Schema: Cycling Between complexTypes

XML schemas permit cycling between definitions of complexTypes. Figure 6–3 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 two:

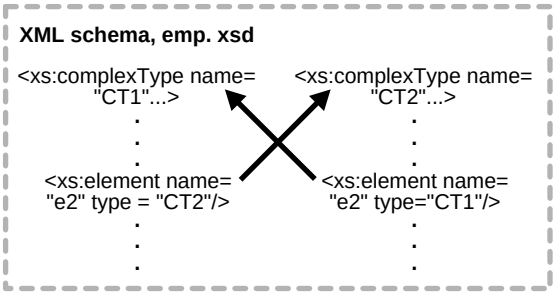
```
DECLARE
  doc VARCHAR2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:xdb="http://xmlns.oracle.com/xdb">
       <xs:complexType name="CT1" xdb:SQLType="CT1">
         <xs:sequence>
           <xs:element name="e1" type="xs:string"/>
           <xs:element 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 (reference) 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 6–3 Cross Referencing Between Different complexTypes in the Same XML Schema



Example 6–15 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.

```
CREATE TYPE BODY_COLL AS VARRAY(32767) OF VARCHAR2(4000);
CREATE TYPE SECTION_T AS OBJECT (SYS_XDBPD$  XDB.XDB$RAW_LIST_T,
                                 "title"     VARCHAR2(4000),
                                 "body"      BODY_COLL,
                                 "section"   XDB.XDB$REF_LIST_T) NOT FINAL;
```

Note: The `section` attribute is declared as a VARRAY of REF references to `XMLType` instances. Because there can be more than one occurrence of embedded sections, the attribute is a VARRAY. And it is a VARRAY of REF references to `XMLType` values in order to avoid forming a cycle of SQL objects.

How a complexType Can Reference Itself

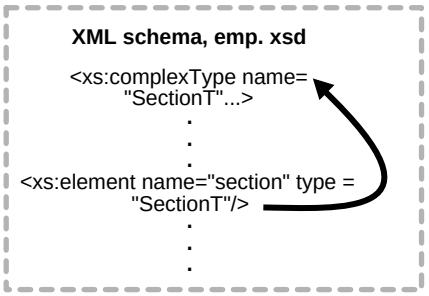
Assume that your XML schema, identified by "`http://www.oracle.com/PO.xsd`", has been registered. An `XMLType` table, `myPOs`, can then be created to store instances

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 6–4 illustrates schematically how a `complexType` can reference or cycle itself.

Figure 6–4 complexType Self Referencing Within an XML Schema



See Also: "Cyclical References Between XML Schemas" on page 6-27

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, "XMLType Operations", under "Changing the Storage Options on an XMLType Column Using XMLData".

Oracle XML DB: 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 not `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 6–1 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 6–1 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 XML Data."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 6–16 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 6–2 shows the mapping of existsNode() in the absence of the SYS_XDBPD$ attribute.

Table 6–2 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
/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

Oracle XML DB: 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 6–3 shows the mapping of various XPath expressions in the case of extractValue(). If an XPath expression targets an element, then 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 6–3 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"

Table 6–3 (Cont.) XPath Mapping for extractValue()

XPath Expression	Maps to
/PurchaseOrder/PONum	XMLData."PONum"
/PurchaseOrder[PONum=2100]	(SELECT TO_XML(x.XMLData) FROM Dual WHERE x."PONum"=2100)
/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 6–17 Rewriting extractValue()

For example, a 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;
```

Because it gets rewritten to simple scalar columns, indexes on the PONum attribute, if any, can 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 6–18 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, because unlike a function-based index, the same index can now satisfy queries that target the column, such as:

```
EXISTSNODE(value(x),'/PurchaseOrder[PONum=1001]')=1;
```

Oracle XML DB: Rewrite for extract()

Function `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 6–4 shows the mapping of various XPath values in the case of `extract()`, when document order is preserved (that is, when `SYS_XDBPD$` exists and `maintainDOM="true"` in the schema document).

Table 6–4 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$, 'PurchaseDate')=1 THEN XMLElement("PONum" , XMLData."PurchaseDate") ELSE NULL END`
`/PurchaseOrder/PONum`	`CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD$, '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$,'PurchaseDate")=1 THEN XMLElement("PONum", XMLData."PurchaseDate") ELSE NULL END FROM DUAL WHERE x."PONum"=2100)`
`/PurchaseOrder/PONum/text()`	`XMLElement("", XMLData.PONum)`
`/PurchaseOrder/Item`	`(SELECT XMLAgg(XMLForest(value(p) as "Item")) FROM TABLE ( x.XMLData."Item" ) p WHERE value(p) IS NOT NULL )`
`/PurchaseOrder/Item/Part`	`(SELECT XMLAgg( CASE WHEN Check_Node_Exists(p.SYS_XDBPD$,'Part")=1 THEN XMLForest(p."Part" as "Part") ELSE NULL END) FROM TABLE ( x.XMLData."Item" ) p)`
`/PurchaseOrder/Item/Part/text()`	`(SELECT XMLAgg(XMLElement("PONum", p."Part") ) FROM TABLE ( x.XMLData."Item" ) x )`

Example 6–19 XPath Mapping for extract() with Document Ordering Preserved

Using the mapping in Table 6–4, a query that extracts the `PONum` element where the purchase order 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 6–5 shows the mapping of `existsNode()` in the absence of the `SYS_XDBPD$` attribute.

Table 6–5 XPath Mapping for extract() Without Document Ordering Preserved

XPath	Equivalent to
`/PurchaseOrder`	`XMLForest(XMLData AS "PurchaseOrder")`
`/PurchaseOrder/@PurchaseDate`	`XMLForest(XMLData."PurchaseDate" AS "PurchaseDate")`
`/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 "PurchaseDate "") 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 in an object-relational manner 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.

Function `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 in an object-relational manner using Oracle XML DB 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, the preceding UPDATE operation becomes:

```
UPDATE po_tab p
  SET p.XMLData."PurchaseDate" = TO_DATE('2002-01-02', 'SYYYY-MM-DD'),
      p.XMLData."PONum" = 2100;
```

DATE Conversions

Date datatypes such as DATE, gMONTH, and gDATE have different formats in XML schema and SQL. In such cases, if the updateXML() has a string value for these columns, then 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, not the SQL DATE format.

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 6–20 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',
  '<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:my="xm40"
           targetNamespace="xm40">
     <include schemaLocation="xm40a.xsd"/>
     <!-- Define a global complextype here -->
     <complexType name="Company">
       <sequence>
         <element name="Name" type="string"/>
         <element name="Address" type="string"/>
       </sequence>
     </complexType>
     <!-- Define a global element depending on included schema -->
     <element name="Emp" type="my:Employee"/>
   </schema>',
  TRUE,
  TRUE,
  FALSE,
  TRUE);
END;
```

```
/
```

It can however be created with the FORCE option:

```
BEGIN DBMS_XMLSCHEMA.registerSchema(
  'xm40.xsd',
  '<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:my="xm40"
           targetNamespace="xm40">
     <include schemaLocation="xm40a.xsd"/>
     <!-- Define a global complextype here -->
     <complexType name="Company">
       <sequence>
         <element name="Name" type="string"/>
         <element name="Address" type="string"/>
       </sequence>
     </complexType>
     <!-- Define a global element depending on included schema -->
     <element name="Emp" type="my:Employee"/>
   </schema>',
  TRUE,
  TRUE,
  FALSE,
  TRUE,
  TRUE);
END;
/
```

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 the 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:

```
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 that 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.

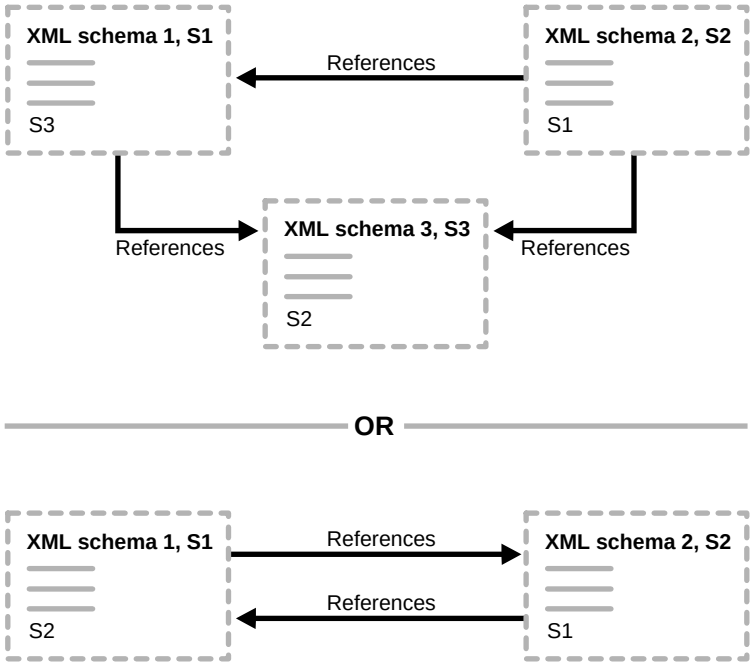
2. Step 2: Register s2.xsd in FORCE mode:

   ```
   DBMS_XMLSCHEMA.registerSchema("s2.xsd", "<schema ..", ..., FORCE => TRUE)
   ```

   The second operation automatically compiles s1.xsd and makes both XML schemas valid.

See Figure 6–5. The preceding example is illustrated in the lower half of the figure.

Figure 6–5 Cyclical References Between XML Schemas



Guidelines for Using XML Schema and Oracle XML DB

This section describes guidelines for using XML schema and Oracle XML DB:

Using Bind Variables in XPath Expressions

When you use bind variables, Oracle Database rewrites the queries for the cases where the bind variable is used in place of a string literal value. You can also use the CURSOR_SHARING set to force Oracle Database to always use bind variables for all string expressions.

XML Query Rewrites with Bind Variables in XPath

When bind variables are used as string literals in XPath, the expression can be rewritten to use the bind variables. The bind variable must used in place of the string

literal using the concatenation operator (||), and it must be surrounded by single (') or double (") quotes inside the XPath string. The following example illustrates the use of the bind variable with query rewrite.

Example 6–21 Using Bind Variables in XPath

```
BEGIN
  DBMS_XMLSCHEMA.registerschema(
    'bindtest.xsd',
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:xdb="http://xmlns.oracle.com/xdb">
       <xs:element name="Employee" xdb:SQLType="EMP_BIND_TYPE">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="EmployeeId"  type="xs:positiveInteger"/>
             <xs:element name="PhoneNumber" type="xs:string"/>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
     </xs:schema>',
    TRUE,
    TRUE,
    FALSE,
    FALSE);
END;
/
REM Create table corresponding to the Employee element
CREATE TABLE emp_bind_tab OF XMLType
  ELEMENT "bindtest.xsd#Employee";
REM Create an index to illustrate the use of bind variables
CREATE INDEX employeeId_idx ON EMP_BIND_TAB
  (ExtractValue(object_value, '/Employee/EmployeeId'));
EXPLAIN PLAN FOR
  SELECT Extractvalue(object_value, '/Employee/PhoneNumber')
    FROM emp_bind_tab p
    WHERE ExistsNode(object_value, '/Employee[EmployeeId="'||:1||'"] ') = 1;

SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
| Id | Operation | Name |
------------------------------------------------------------------------------
| 0 | SELECT STATEMENT |
| 1 | TABLE ACCESS BY INDEX ROWID| EMP_BIND_TAB |
|* 2 | INDEX RANGE SCAN | EMPLOYEEID_IDX |
------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("P"."SYS_NC00008$"=TO_NUMBER(:1))
```

The bind variable `:1` is used as a string literal value enclosed by double quotes ("). This allows the XPath expression `'/Employee[EmployeeId=" ' || :1 || '"]'` to be rewritten, and the optimizer can use the `EmployeeId_idx` index to satisfy the predicate.

Setting CURSOR_SHARING to FORCE

With query rewrites, Oracle Database changes the input XPath expression to use the underlying columns. This means that for a given XPath there is a particular set of

columns or tables that is referenced underneath. This is a compile-time operation, because the shared cursor must know *exactly* which tables and columns it references. This cannot change with each row or instantiation of the cursor.

Hence if the XPath expression itself is a bind variable, Oracle Database cannot do any rewrites, because each instantiation of the cursor can have totally different XPaths. This is similar to binding the name of the column or table in a SQL query. For example, `SELECT * FROM table(:1)`.

Note: You can specify bind variables on the right side of the query. For example, this query uses the usual bind variable sharing:

```
SELECT * FROM purchaseorder p WHERE
extractvalue(
  value(p),
 '/PurchaseOrder/LineItems/LineItem/ItemNumber')
:= :1;
```

When `CURSOR_SHARING` is set to `FORCE`, by default each string constant including XPath becomes a bind variable. When Oracle Database then encounters `extractvalue()`, `existsnode()`, and so on, it looks at the XPath bind variables to check if they are really constants. If so then it uses them and rewrites the query. Hence there is a large difference depending on where the bind variable is used.

Creating Constraints on Repetitive Elements in Schema-Based XML Instance Documents

After creating an `XMLType` table based on an XML schema, you may need to add a unique constraint to one of the elements. That element can occur more than once. To create constraints on elements that occur more than once in the XML instance document, you must store the `VARRAY` as a table. This is considered an **Ordered Collection in the Table**, or OCT. In an OCT the elements of the `VARRAY` are stored in separate tables. You can then create constraints on the OCT.

The following example shows the attribute `No` of `<PhoneNumber>` that can appear more than once, and a unique constraint added to ensure that the same number cannot be repeated in the same XML instance document.

Example 6–22 Creating Constraints on Elements in Schema-Based Tables that Occur More than Once Using OCT

In this example, the constraint applies to each collection and not across all XML instances. This is achieved by creating a concatenated index with the collection `id` column. To apply the constraint across all collections of all instance documents, simply omit the collection `id` column.

```
BEGIN DBMS_XMLSCHEMA.registerschema(
  'emp.xsd',
  '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:xdb="http://xmlns.oracle.com/xdb">
     <xs:element name="Employee" xdb:SQLType="EMP_TYPE">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="EmployeeId" type="xs:positiveInteger"/>
           <xs:element name="PhoneNumber" maxOccurs="10">
             <xs:complexType>
               <xs:attribute name="No" type="xs:integer"/>
```

```
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>',
  TRUE,
  TRUE,
  FALSE,
  FALSE);
END;
/
```

This returns the following:

```
PL/SQL procedure successfully completed.

CREATE TABLE emp_tab OF XMLType
  XMLSCHEMA "emp.xsd" ELEMENT "Employee"
  VARRAY xmldata."PhoneNumber" STORE AS table phone_tab;
```

This returns:

```
Table created.

ALTER TABLE phone_tab AD unique(nested_table_id, "No");
```

This returns:

```
Table altered.

INSERT INTO emp_tab
  VALUES (XMLType('<Employee>
                    <EmployeeId>1234</EmployeeId>
                    <PhoneNumber No="1234"/>
                    <PhoneNumber No="2345"/>
                  </Employee>').createschemabasedxml('emp.xsd'));
```

This returns:

```
1 row created.

INSERT INTO emp_tab
  VALUES(XMLType('<Employee>
                   <EmployeeId>3456</EmployeeId>
                   <PhoneNumber No="4444"/>
                   <PhoneNumber No="4444"/>
                 </Employee>').createschemabasedxml('emp.xsd'));
```

This returns:

```
INSERT INTO emp_tab values(XMLType(
 *
 ERROR at line 1:
 ORA-00001: unique constraint (SCOTT.SYS_C002136) violated
```

Guidelines for Loading and Retrieving Large Documents with Collections

Two parameters were added to `xdbconfig` in Oracle Database 10*g* in order to control the amount of memory used by the loading operation. These tunable parameters

provide mechanisms to optimize the loading process provided the following conditions are met:

- The document is loaded either through protocols (FTP, HTTP, or DAV) or through the `createResource` API.
- The document is a schema based document containing large collections (that is, it contains elements with `maxoccurs` set to a large number).
- The collections of the document are stored as OCTs. This is achieved by either of the following ways:
  - Setting `xdb:storeVarrayAsTable="true"` in the schema definition which turns this storage option on for all collections of the schema.
  - By setting the table properties appropriately in the element definition.
- These optimizations are most useful when there are no triggers on the base table. For situations where triggers appear, the performance may be suboptimal.

The basic idea behind this optimization is that it allows the collections to be swapped into or out of the memory in bounded sizes. As an illustration of this idea consider the following example conforming to a purchase order schema:

```
<PurchaseOrder>
  <LineItem itemID="1">
    ...
  </LineItem>
    .
    .
  <LineItem itemID="10240">
    ...
  </LineItem>
</PurchaseOrder>
```

The purchase order document here contains a collection of 10240 `LineItem` elements. Instead of creating the entire document in memory and then pushing it out to disk (a process that leads to excessive memory usage and in some instances a load failure due to inadequate system memory), we create the documents in finite chunks of memory called **loadable units**. In the example case, if we assume that each line item needs 1K memory and we want to use loadable units of size 512K, then each loadable unit will contain 512K/1K = 512 line items and there will be approximately 20 such units. Moreover, if we wish that the entire memory representation of the document never exceeds 2M in size, we ensure that at any time no more than 2M/512K = 4 loadable units are maintained in the memory. We use an LRU mechanism to swap out the loadable units.

By controlling the size of the loadable unit and the bound on the size of the document you can tune the memory usage and performance of the load or retrieval. Typically a larger loadable unit size translates into lesser number of disk accesses but takes up more memory. This is controlled by the parameter `xdbcore-loadableunit-size` whose default value is 16K. The user can indicate the amount of memory to be given to the document by setting the `xdbcore-xobmem-bound` parameter which defaults to 1M. The values to these parameters are specified in Kilobytes. So, the default value of `xdbcore-xobmem-bound` is 1024 and that of `xdbcore-loadableunit-size` is 16. These are soft limits that provide some guidance to the system as to how to use the memory optimally.

In the preceding example, when we do the FTP load of the document, the pattern in which the loadable units (LU) are created and flushed to the disk is as follows:

```
No LUs
```

```
Create LU1[LineItems(LI):1-512]
LU1[LI:1-512], Create LU2[LI:513-1024]
.
.
LU1[LI:1-512],...,Create LU4[LI:1517:2028]    <-   Total memory size = 2M
Swap Out LU1[LI:1-512], LU2[LI:513-1024],...,LU4[LI:1517-2028], Create
LU5[LI:2029-2540]
Swap Out LU2[LI:513-1024], LU3, LU4, LU5, Create LU6[LI:2541-2052]
.
.
.
Swap Out LU16, LU17, LU18, LU10, Create LU20[LI:9729-10240]
Flush LU17,LU18,LU19,LU20
```

Guidelines for Setting xdbcore Parameters

Typically if you have 1 Gigabyte of addressable PGA, give about 1/10th of PGA to the document. So, `xobcore-xobmem-bound` should be set to 1/10 of addressable PGA which equals 100M. During full document retrievals and loads, the `xdbcore-loadableunit-size` should be as close to the `xobcore-xobmem-bound` size as possible, within some error. However, in practice, we set it to half the value of `xdbcore-xobmem-bound`; in this case this is 50 M. Starting with these values, try to load the document. In case you run out of memory, lower the `xdbcore-xobmem-bound` and set the `xdbcore-loadableunot-size` to half of its value, and continue until the documents load. In case the load succeeds, try to see if you can increase the `xdbcore-loadableunit-size` to squeeze out better performance. If `xdbcore-loadableunit-size` equals `xdbcore-xobmem-bound`, then try to increase both parameters for further performance improvements.

Updating Your XML Schema Using Schema Evolution

You can update your XML schema after you have registered it with Oracle XML DB using the XML schema evolution process.

See: Chapter 7, "XML Schema Evolution"

7

XML Schema Evolution

This chapter describes how you can update your XML schema after you have registered it with Oracle XML DB. XML schema evolution is the process of updating your registered XML schema.

This chapter contains these topics:

Introducing XML Schema Evolution

Oracle XML DB supports the W3C XML Schema recommendation. XML instances that conform to an XML schema can be stored and retrieved using SQL and protocols such as FTP, HTTP, and WebDAV. In addition to specifying the structure of XML documents, XML schemas determine the mapping between XML and object-relational storage.

See: Chapter 5, "XML Schema Storage and Query: The Basics"

In prior releases an XML schema, once registered with Oracle XML DB at a particular URL, could not be modified or evolved because there may be `XMLType` tables that depend on the XML schema. There was no standard procedure for schema evolution. This release supports XML schema evolution by providing a PL/SQL procedure `CopyEvolve()` a part of the `DBMS_XMLSCHEMA` package. `CopyEvolve()` involves copying existing instance documents to temporary tables, dropping and re-registering the XML schema, and copying the instance documents to the new `XMLType` tables.

With `copyevolve()` you can evolve your registered XML schema in such a way that existing XML instance documents continue to be valid. If you do not care about the existing documents, you can simply drop the `XMLType` tables dependent on the XML schema, delete the old XML schema, and register the new XML schema at the same URL.

`CopyEvolve()` has certain limitations. These are described in the section, "Limitations of CopyEvolve()".

Limitations of CopyEvolve()

The following are the limitations of `CopyEvolve()`:

- Indexes, triggers, constraints, RLS policies and other metadata related to the `XMLType` tables that are dependent on the schemas that are evolved, will not be preserved. These must be re-created after evolution.
- If top-level element names are being changed, there are more steps to be followed after `CopyEvolve()` completes executing. See the section on "Top-Level Element Name Changes" on page 7-3 for more details.
- Data copy-based evolution cannot be used if there is a table with an object-type column that has an `XMLType` attribute that is dependent on any of the schemas to be evolved. For example, consider a table TAB1 that is created in the following way:

```
CREATE TYPE t1 AS OBJECT (n NUMBER, x XMLType);
CREATE TABLE tab1 (e NUMBER, o t1) XMLType COLUMN o.x XMLSchema "s1.xsd"
                  ELEMENT "Employee";
```

  The example assumes that an XML schema with a top-level element `Employee` has been registered under URL `s1.xsd`. It is not possible to evolve this XML schema since table `TAB1` with column `O` with `XMLType` attribute `X` is dependent on this XML schema.

Example XML Schema

The following is an example of an XML schema along with typical changes you may want to make. Changes to be made are shown in bold. For changes to attributes, the old value is shown in italics, followed by the new value:

Example 7–1 Example XML Schema to be Evolved

This example shows the changes that need to be made in bold.

```
<schema targetNamespace="http://www.oracle.com/po.xsd"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:po="http://www.oracle.com/po.xsd"
  elementFormDefault="qualified">
  <annotation>
   <documentation xml:lang="en">
     Purchase Order schema for US PO's.
   </documentation>
  </annotation>
<complexType name="Address">
 <sequence>
  <element name="name" type="string"/>
  <element name="street" type="string"/>
  <element name="city" type="string"/>
 </sequence>
</complexType>
<!-- A type representing US States -->
 <simpleType name="USState">
  <restriction base="string">
   <enumeration value="NY"/>
   <enumeration value="TX"/>
   <enumeration value="CA"/>
   <enumeration value="FL"/>
  </restriction>
 </simpleType>
<complexType name="USAddress">
 <complexContent>
 <extension base="po:Address">
```

```
   <sequence>
     <element name="STATE" name="State" type="po:USState"/>
     <element name="zip" type="positiveInteger"/>
   </sequence>
 </extension>
 </complexContent>
</complexType>
<element name="PurchaseOrder">
   <complexType>
     <sequence>
        <element name="PO-Number" type="string"/>
        <element name="LineItems" maxOccurs="unbounded">
            <complexType>
              <sequence>
                <element name="part-num" type="string" maxLength="20"/>
                <element name="unit-price" type="float"/>
                <element name="quantity" type="integer"/>
             </sequence>
            </complexType>
           </element>
        <element name="shipTo" type="po:Address"/>
     </sequence>
   </complexType>
```

The next section describes steps for accomplishing copy-based schema evolution.

Guidelines for Using DBMS_XMLSCHEMA.CopyEvolve()

Here are some guidelines for using `DBMS_XMLSCHEMA.CopyEvolve()`:

1. First identify the XML schemas that are dependent on the XML schema to be evolved. You can acquire the URLs of the dependent XML schemas using the following query:

   ```
   SELECT dxs.schema_url
       FROM dba_dependencies dd, dba_xml_schemas dxs
       WHERE dd.referenced_name=(SELECT int_objname
             FROM dba_xml_schemas WHERE schema_url=<EVOL_SCH_URL>
             AND owner=<EVOL_SCH_OWNER>)
             AND dxs.owner = <EVOL_SCH_OWNER>
             AND dxs.int_objname=dd.name;
   ```

   In many cases, no changes may be necessary in the dependent XML schemas. But if the dependent XML schemas need to be changed, you must also prepare new versions of those XML schemas.

2. If the existing instance documents do not conform to the new XML schema, you must provide an XSL style sheet that, when applied to an instance document, will transform it to conform to the new schema. This needs to be done for each XML schema identified in Step 1. The transformation must handle documents that conform to all top-level elements in the new XML schema.

3. Call `CopyEvolve()`, specifying the XML schema URLs, new schemas, and transformations.

Top-Level Element Name Changes

The `CopyEvolve()` procedure assumes that top-level elements have not been dropped and that their names have not been changed in the new XML schemas. If there are such changes in your new XML schemas, you can call `CopyEvolve()` with

the `generateTables` parameter set to FALSE and the `preserveOldDocs` parameter set to TRUE. In this way new tables are generated and the temporary tables holding the old documents are not dropped at the end of the procedure. You can then store the old documents in whatever form is appropriate and drop the temporary tables. See "DBMS_XMLSCHEMA.CopyEvolve() Syntax" on page 7-5 for more details on the using these parameters.

Ensure that the XML Schema and Dependents are Not Used by Concurrent Sessions

Ensure that the XML schema and its dependents are not used by any concurrent session during the XML schema evolution process. If other concurrent sessions have shared locks on this schema at the beginning of the evolution process, `DBMS_XMLSCHEMA.CopyEvolve()` waits for these sessions to release the locks so that it can acquire an exclusive lock. However this lock is released immediately to allow the rest of the process to continue.

What Happens When CopyEvolve() Raises an Error? Rollback

`CopyEvolve()` either completely succeeds or raises an error in which case it attempts to rollback as much of the operation as possible. Evolving a schema involves many database DDL statements. When an error occurs, compensating DDL statements are executed to undo the effect of all steps executed to that point. If the old tables/schemas have been dropped they are re-created but any table/column/storage properties and auxiliary structures associated with the tables/columns like indexes, triggers, constraints, and RLS policies are lost.

Failed Rollback From Insufficient Privileges

In certain cases you cannot rollback the operation. For example, if table creation fails due to reasons not related to the new schema, such as, from insufficient privileges, there is no way to rollback. The temporary tables are not deleted even if `preserveOldDocs` is false, so that the data can be recovered. If the `mapTabName` parameter is null, the mapping table name is `XDB$MAPTAB` followed by a sequence number. The exact table name can be found using a query such as:

```
SELECT table_name FROM user_tables
   WHERE table_name LIKE 'XDB$MAPTAB%';
```

Using CopyEvolve(): Privileges Needed

Schema evolution may involve dropping/creating types. Hence you need type-related privileges such as `DROP TYPE`, `CREATE TYPE`, and `ALTER TYPE`.

You need privileges to delete and register the XML schemas involved in the evolution. You need all privileges on `XMLType` tables that conform to the schemas being evolved. For `XMLType` columns the `ALTER TABLE` privilege is needed on corresponding tables. If there are schema-based `XMLType` tables or columns in other users' database schemas, you need privileges such as `CREATE ANY TABLE`, `CREATE ANY INDEX`, `SELECT ANY TABLE`, `UPDATE ANY TABLE`, `INSERT ANY TABLE`, `DELETE ANY TABLE`, `DROP ANY TABLE`, `ALTER ANY TABLE`, and `DROP ANY INDEX`.

To avoid having to grant all these privileges to the schema owner, Oracle Corporation recommends that the evolution be performed by a DBA if there are XML schema-based `XMLType` table or columns in other users' database schemas.

DBMS_XMLSCHEMA.CopyEvolve() Syntax

Here is the `DBMS_XMLSCHEMA.CopyEvolve()` syntax:

```
procedure CopyEvolve(schemaURLs       IN XDB$STRING_LIST_T,
                     newSchemas        IN XMLSequenceType,
                     transforms        IN XMLSequenceType := NULL,
                     preserveOldDocs   IN BOOLEAN := FALSE,
                     mapTabName        IN VARCHAR2 := NULL,
                     generateTables    IN BOOLEAN := TRUE,
                     force             IN BOOLEAN := FALSE,
                     schemaOwners      IN XDB$STRING_LIST_T := NULL);
```

Table 7–1 DBMS_XMLSCHEMA.CopyEvolve(): Parameters

Parameter	Description
schemaURLs	Varray of URLs of XML schemas to be evolved. This should include the dependent schemas as well. Unless the force parameter is TRUE, the URLs should be in the dependency order, that is, if URL A comes before URL B in the Varray, then schema A should not be dependent on schema B but schema B may be dependent on schema A.
newSchemas	Varray of new XML schema documents. Specify this in exactly the same order as the corresponding URLs. If no change is necessary in an XML schema, provide the unchanged schema.
transforms	Varray of XSL documents that will be applied to XML schema based documents to make them conform to the new schemas. Specify these in exactly the same order as the corresponding URLs. If no transformations are required, this parameter need not be specified.
preserveOldDocs	If this is TRUE the temporary tables holding old data are not dropped at the end of schema evolution. See also "How DBMS_XMLSCHEMA.CopyEvolve() Works".
mapTabName	Specifies the name of table that maps old XMLType table or column names to names of corresponding temporary tables.
generateTables	By default this parameter is TRUE; if this is FALSE, XMLType tables or columns will not be generated after registering new schemas. If this is FALSE, preserveOldDocs must be TRUE and mapTabName must be non-null.
force	If this is TRUE errors during the registration of new schemas are ignored. If there are circular dependencies among the schemas, set this flag to TRUE to ensure that each schema is stored even though there may be errors in registration.
schemaOwners	Varray of names of schema owners. Specify these in exactly the same order as the corresponding URLs.

Table 7–2 DBMS_XMLSCHEMA.CopyEvolve(): Errors and Exceptions

Error Number and Message	Cause	Action
30942 XML Schema Evolution error for schema '<schema_url>' table "<owner_name>.<table_name>" column '<column_name>'	The given XMLType table or column that conforms to the given schema had errors during evolution. In the case of a table the column name will be empty. See also the more specific error that follows this.	Based on the schema, table, and column information in this error and the more specific error that follows, take corrective action.

Table 7–2 (Cont.) DBMS_XMLSCHEMA.CopyEvolve(): Errors and Exceptions

Error Number and Message	Cause	Action
30943 XML Schema '<schema_url>' is dependent on XML schema '<schema_url>'	Not all dependent XML schemas were specified or the schemas were not specified in dependency order, that is, if schema S1 is dependent on schema S, S must appear before S1.	Include the previously unspecified schema in the list of schemas or correct the order in which the schemas are specified. Then retry the operation.
30944 Error during rollback for XML schema '<schema_url>' table "<owner_name>.<table_name>" column '<column_name>'	The given XMLType table or column that conforms to the given schema had errors during a rollback of XML schema evolution. For a table the column name will be empty. See also the more specific error that follows this.	Based on the schema, table, and column information in this error and the more specific error that follows, take corrective action.
30945 Could not create mapping table '<table_name>'	A mapping table could not be created during XML schema evolution. See also the more specific error that follows this.	Ensure that a table with the given name does not exist and retry the operation.
30946 XML Schema Evolution warning: temporary tables not cleaned up	An error occurred after the schema was evolved while cleaning up temporary tables. The schema evolution was successful.	If you need to remove the temporary tables, use the mapping table to get the temporary table names and drop them.

Example 7–2 Using DBMS_XMLSCHEMA.CopyEvolve() to Update an XML Schema

In this example, the `address.xsd` schema needs to be evolved. The new XML schema adds a new element `State` as a child of the top-level `Address` element. It also renames the element `STREET` to `Street`. Since it renames an existing element, the old instance documents may not conform to the new schema and so an XSL transformation is required to transform them to conform to the new schema.

```
declare
  newaddr    XMLType;
  transform  XMLType;
begin
  newaddr := xmltype(
   '<schema targetNamespace="http://www.example.com/IPO"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:ipo="http://www.example.com/IPO"
          xmlns:xdb="http://xmlns.oracle.com/xdb"
          elementFormDefault="qualified">

  <element name="Address" xdb:defaultTable="ADDR_TAB">
    <complexType>
      <sequence>
        <element name="Name"   type="string"/>
        <element name="Street" type="string"/>
        <element name="City"   type="string" />
        <element name="State"   type="string" />
      </sequence>
    </complexType>
  </element>
  </schema>');
```

```
transform := xmltype(
  '<xsl:stylesheet  version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.example.com/IPO"
 xmlns:ipo="http://www.example.com/IPO">
   <xsl:template match="@*|node()">
      <xsl:copy>
        <xsl:apply-templates select="@*|node()"/>
      </xsl:copy>
   </xsl:template>
   <xsl:template match="/ipo:Address/ipo:STREET">
    <Street>
      <xsl:for-each select="@*|node()">
         <xsl:copy-of select="."/>
      </xsl:for-each>
    </Street>
  </xsl:template>
 </xsl:stylesheet>');

dbms_xmlschema.CopyEvolve(xdb$string_list_t('address.xsd'),
      XMLSequenceType(newaddr), XMLSequenceType(transform));
end;
```

How DBMS_XMLSCHEMA.CopyEvolve() Works

The `DBMS_XMLSCHEMA.CopyEvolve()` procedure is used to evolve registered XML schemas such that existing XML instances continue to remain valid.

> **Note:** Since this procedure deletes all documents conforming to the XML schemas during the process of schema evolution, backup all these documents and schemas before executing this procedure.

First `CopyEvolve()` copies the data in schema based `XMLType` tables and columns to temporary tables. It then drops the tables and columns and deletes the old XML schemas. After registering the new XML schemas, it creates `XMLType` tables and columns and populates them with data (unless the `genTables` parameter is FALSE) but it does not create any auxiliary structures such as indexes, constraints, triggers, and row-level security (RLS) policies. `CopyEvolve()` creates the tables and columns in the following way:

- It creates default tables while registering the new schemas.
- It creates nondefault tables by a statement of the following form:

  ```
  CREATE TABLE <TABLE_NAME> OF XMLType OID '<OID>'
     XMLSCHEMA <SCHEMA_URL> ELEMENT <ELEMENT_NAME>
  ```

  where <OID> is the original OID of the table, before it was dropped.
- `XMLType` columns are added using a statement of the following form:

  ```
  ALTER TABLE <Table_Name> ADD (<Column_Name> XMLType) XMLType column
    <Column_Name> xmlschema <Schema_Url> ELEMENT <Element_Name>
  ```

When a new schema is registered, types or beans are generated if the registration of the corresponding old schema had generated types or beans. If an XML schema was global before the evolution it will be global after the evolution. Similarly if an XML

schema was local before the evolution it will be local (owned by the same user) after the evolution.

You have the option to preserve the temporary tables that contain the old documents by passing in TRUE for the `preserveOldDocs` parameter. In this case, the procedure does not drop the temporary tables at the end. All temporary tables are created in the current user's database schema. For `XMLType` tables the temp table will have the following columns:

Table 7–3 XML Schema Evolution: XMLType Table Temporary Table Columns

Name	Type	Comment
Data	CLOB	XML doc from old table in CLOB format.
OID	RAW(16)	OID of corresponding row in old table.
ACLOID	RAW(16)	This column is present only if old table is hierarchy enabled. ACLOID of corresponding row in old table.
OWNERID	RAW(16)	This column is present only if old table is hierarchy enabled. OWNERID of corresponding row in old table.

For `XMLType` columns the temp table will have the following columns:

Table 7–4 XML Schema Evolution: XMLType Column Temporary Table Columns

Name	Type	Comment
Data	CLOB	XML document from old column in CLOB format.
RID	ROWID	ROWID of corresponding row in the table that this column was a part of.

The `CopyEvolve()` procedure stores information about the mapping from the old table or column name to the corresponding temporary table name in a separate table specified by the `mapTabName` parameter. If `preserveOldDocs` is TRUE, the `mapTabName` parameter must be non-null and must not be the name of any existing table in the current user's schema. Each row in the mapping table has information about one of the old tables/columns. Table 7–5 shows the mapping table columns.

Table 7–5 CopyEvolve() Mapping Table

Column Name	Column Type	Comment
SCHEMA_URL	VARCHAR2(700)	URL of schema to which this table/column conforms.
SCHEMA_OWNER	VARCHAR(30)	Owner of the schema.
ELEMENT_NAME	VARCHAR2(256)	Element to which this table/column conforms.
TABLE_NAME	VARCHAR2(65)	Qualified Name of table (<owner_name>.<table_name>).
TABLE_OID	RAW(16)	OID of table.
COLUMN_NAME	VARCHAR2(4000)	Name of column (this will be null for XMLType tables).
TEMP_TABNAME	VARCHAR2(30)	Name of temporary table which holds the data for this table/column.

You can also avoid generating any tables or columns after registering the new XML schema, by using FALSE as the `genTables` parameter. If `genTables` is FALSE, the `preserveOldDocs` parameter must be TRUE and the `mapTabName` parameter must be non-null. This ensures that the data in the old tables is not lost. This is useful if you do not want the tables to be created by the procedure, as described in section "DBMS_XMLSCHEMA.CopyEvolve() Syntax".

By default it is assumed that all XML schemas are owned by the current user. If this is not true, you must specify the owner of each XML schema in the `schemaOwners` parameter.

8

Transforming and Validating XMLType Data

This chapter describes the SQL functions and `XMLType` APIs for transforming `XMLType` data using XSLT style sheets. It also explains the various functions and APIs available for validating the `XMLType` instance against an XML schema.

This chapter contains these topics:

Transforming XMLType Instances

XML documents have structure but no format. To add format to the XML documents you can use Extensible Stylesheet Language (XSL). XSL provides a way of displaying XML semantics. It can map XML elements into other formatting or mark-up languages such as HTML.

In Oracle XML DB, `XMLType` instances or XML data stored in `XMLType` tables, columns, or views in Oracle Database, can be (formatted) transformed into HTML, XML, and other mark-up languages, using XSL style sheets and the `XMLType` function, transform(). This process conforms to the W3C XSL Transformations 1.0 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.

> **Note:** The PL/SQL package `DBMS_XSLPROCESSOR` provides a convenient and efficient way of applying a single style sheet to multiple documents. The performance of this package will be better than `transform()` because the style sheet will be parsed only once.

See Also:

- Chapter 3, "Using Oracle XML DB", the section, "XSL Transformation" on page 3-80
- "PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)" on page 10-21
- Appendix D, "XSLT Primer"
- *Oracle XML Developer's Kit Programmer's Guide*, the chapter on XSQL Pages Publishing Framework

XMLTransform() and XMLType.transform()

Figure 8–1 shows the `XMLTransform()` syntax. The `XMLTransform()` function takes as arguments an `XMLType` instance and an XSLT style sheet (which is itself an `XMLType` instance). It applies the style sheet to the instance and returns an `XMLType` instance.

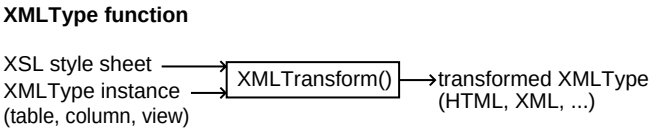
Note: You can also use the syntax, `XMLTYPE.transform()`. This is the same as `XMLTransform()`.

Figure 8–2 shows how `XMLTransform()` transforms the XML document by using the XSLT style sheet passed in. It returns the processed output as XML, HTML, and so on, as specified by the XSLT style sheet. You typically are required to use `XMLTransform()` when retrieving or generating XML documents stored as `XMLType` in the database.

See Also: Figure 1–1, "Oracle XML DB Architecture: XMLType Storage and Repository" in Chapter 1, "Introducing Oracle XML DB"

Figure 8–1 XMLTransform() Syntax

Figure 8–2 Using XMLTransform()



XMLTransform() Examples

Use the following code to set up the XML schema and tables needed to run the examples in this chapter. (The call to `deleteSchema` is to ensure that there is no existing schema before creating one. If no such schema exists, then `deleteSchema` produces an error.)

```
CONNECT scott/tiger
```

```
begin
-- delete the schema, if it already exists; otherwise, this produces an error
dbms_xmlschema.deleteSchema('http://www.example.com/schemas/ipo.xsd',4);
end;
/
begin
-- register the schema
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="[0-9]{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>
        <name>Helen Zoe</name>
        <street>121 Broadway</street>
        <city>Cardiff</city>
        <state>Wales</state>
        <country>UK</country>
        <zip>CF2 1QJ</zip>
    </shipTo>
    <billTo>
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>CA</state>
        <country>US</country>
        <zip>95819</zip>
    </billTo>
    <items>
        <item partNum="833-AA">
            <productName>Lapis necklace</productName>
            <quantity>1</quantity>
            <USPrice>99.95</USPrice>
            <ipo:comment>Want this for the holidays!</ipo:comment>
            <shipDate>1999-12-05</shipDate>
        </item>
    </items>
</ipo:purchaseOrder>'));
```

The following examples illustrate how to use `XMLTransform()` to transform XML data stored as `XMLType` to HTML, XML, or other languages.

Example 8–1 Transforming an XMLType Instance Using XMLTransform() and DBUriType to Get the XSL Style Sheet

`DBUriType` is described in Chapter 17, "Creating and Accessing Data Through URLs".

```
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('/XDB/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 8–2 Using XMLTransform() and a Subquery to Retrieve the Style Sheet

This example illustrates the use of a stored style sheet to transform XMLType instances. Unlike the previous example, this example uses a scalar subquery to retrieve the stored style sheet:

```
SELECT XMLTransform(x.xmlcol,
    (select stylesheet from stylesheet_tab where id = 1)).getStringVal()
     AS result
   FROM po_tab x;
```

Example 8–3 Using Transient Style Sheets and XMLTransform()

This example describes how you can transform XMLType instances using a transient style sheet:

```
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, the database checks to ensure that XMLType instances are well-formed. In addition, for schema-based XMLType instances, the database performs few basic validation checks. Because 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 a 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 own schema of the XMLType instance. If any of the arguments are specified to be NULL, then the result is NULL. If validation fails, then 0 is returned and no errors are reported explaining why the validation has failed.

Syntax

```
XMLIsValid ( XMLType_inst [, schemaurl [, 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, then the document 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, then the conformance is checked against the own schema of the `XMLType` instance. If the validation fails, then 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 8–4 Using isSchemaValid()

```
SELECT x.xmlcol.isSchemaValid('http://www.example.com/schemas/ipo.xsd',
               'purchaseOrder')
    FROM po_tab x;
```

Example 8–5 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 8–6 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.object_value;
   xmltype.schemavalidate(newxml);
END;
/
```

Example 8–7 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(object_value) = 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.

9

Full Text Search Over XML

This chapter describes Full Text search over XML using Oracle.

First we motivate the topic by introducing Full Text search and XML. Then we give an overview and comparison of the `CONTAINS` SQL function and the `ora:contains` XPath function, the two functions used by Oracle to do Full Text search over XML. Then we examine each of these functions in detail. The detailed descriptions have similar headings, so you can compare the two approaches easily.

To get the most out of this chapter you should be familiar with XML, XML DB and Oracle Text. This chapter includes a review of some Oracle Text features.

> **See Also:** *Oracle Text Reference* and *Oracle Text Application Developer's Guide* for more information on Oracle Text

This chapter contains these topics:

Full Text Search and XML

Oracle supports Full Text search on documents that are managed by the Oracle Database.

If your documents are XML, then you can use the XML structure of the document to restrict the Full Text search. For example, you may want to find all purchase orders that contain the word "electric" using Full Text search. If the purchase orders are XML, then you can restrict the search by finding all purchase orders that contain the word "electric" in a comment, or by finding all purchase orders that contain the word "electric" in a comment under items.

If your XML documents are of type `XMLType`, then you can project the results of your query using the XML structure of the document. For example, after finding all purchase orders that contain the word "electric" in a comment, you may want to return just the comments, or just the comments that contain the word "electric".

Comparison of Full Text Search and Other Search Types

Full Text search differs from structured search or substring search in the following ways:

- A Full Text search searches for words rather than substrings. A substring search for comments that contain the **string** "law" will return a comment that contains "my **law**n is going wild". A Full Text search for the **word** "law" will not.
- A Full Text search will support some language-based and word-based searches which substring searches cannot. You can use a language-based search, for example, to find all the comments that contain a word with the same linguistic stem as "mouse", and Oracle Text will find "mouse" and "mice". You can use a word-based search, for example, to find all the comments that contain the word "lawn" within 5 words of "wild".
- A Full Text search generally involves some notion of relevance. When you do a Full Text search for all the comments that contain the word "lawn", for example, some results are more relevant than others. Relevance is often related to the number of times the search word (or similar words) occur in the document.

XML search

XML search is different from unstructured document search. In unstructured document search you generally search across a set of documents to return the documents that satisfy your text predicate. In XML search you often want to use the structure of the XML document to restrict the search. And you often want to return just the part of the document that satisfies the search.

Search using Full Text and XML Structure

There are two ways to do a search that includes Full Text search and XML structure:

- Include the structure inside the Full Text predicate, using the `CONTAINS` SQL function:

  ```
  ... WHERE CONTAINS( DOC, 'electric INPATH (/purchaseOrder/items/item/comment)'
  )>0 ...
  ```

  The `CONTAINS` SQL function is an extension to SQL and can be used in any query. `CONTAINS` requires a `CONTEXT` Full Text index.

- Include the Full Text predicate inside the structure, using the `ora:contains` XPath function:

  ```
  ... '/purchaseOrder/items/item/comment[ora:contains(text(), "electric")>0]' ...
  ```

  The `ora:contains` XPath function is an extension to XPath and can be used in any call to `existsNode`, `extract` or `extractValue`.

About the Examples in this Chapter

This section describes details about the examples included in this chapter.

Roles and Privileges

To run the examples you will need the `CTXAPP` role, as well as `CONNECT` and `RESOURCE`. You must also have `EXECUTE` privilege on the ctxsys package `CTX_DDL`.

Examples Schema and Data

Examples in this chapter are based on "The Purchase Order Schema", w3c XML Schema Part 0: Primer.

> **See Also:**
> `http://www.w3.org/TR/xmlschema-0/#POSchema`

The data in the examples is "Purchase Order po001.xml" on page 9-31. Some of the performance examples are based on a bigger table (`PURCHASE_ORDERS_xmltype_big`), which is included in the downloadable version only.

> **See Also:** `http://www.w3.org/TR/xmlschema-0/#po.xml`

Some examples use `VARCHAR2`, others use `XMLType`. All the examples that use `VARCHAR2` will also work on `XMLType`.

> **See Also:** Oracle Technology Network (`http://otn.oracle.com`) for the example data, the example schema, and a script to run all the examples

Overview of CONTAINS and ora:contains

This section contains these topics:

- Overview of the CONTAINS SQL Function
- Overview of the ora:contains XPath Function
- Comparison of CONTAINS and ora:contains

Overview of the CONTAINS SQL Function

`CONTAINS` returns a positive number for rows where `[schema.]column` matches `text_query`, and zero otherwise. `CONTAINS` is a user-defined function, a standard extension method in SQL. `CONTAINS` requires an index of type `CONTEXT`. If there is no `CONTEXT` index on the column being searched, then `CONTAINS` throws an error.

Syntax

```
CONTAINS signatureCONTAINS(
    [schema.]column,
    text_query    VARCHAR2
    [,label       NUMBER]
    )
RETURN NUMBER
```

Example 9–1 Simple CONTAINS Query

A typical query looks like this:

```
SELECT ID
    FROM PURCHASE_ORDERS
    WHERE CONTAINS( DOC, 'lawn' )>0 ;
```

This query uses table `PURCHASE_ORDERS` and index `po_index`. It returns the ID for each row in table `PURCHASE_ORDERS` where the `DOC` column contains the word "lawn".

Example 9–2 CONTAINS with a Structured Predicate

CONTAINS can be used in any SQL query. Here is an example using table PURCHASE_ORDERS and index po_index:

```
SELECT ID
    FROM PURCHASE_ORDERS
    WHERE CONTAINS( DOC, 'lawn' )>0  AND id<25 ;
```

Example 9–3 CONTAINS Using XML Structure to Restrict the Query

Suppose DOC is a column that contains a set of XML documents. You can do Full Text search over DOC, using its XML structure to restrict the query. This query uses table PURCHASE_ORDERS and index po_index-path-section:

```
SELECT ID
    FROM PURCHASE_ORDERS
    WHERE CONTAINS( DOC, 'lawn WITHIN comment' )>0 ;
```

Example 9–4 CONTAINS with Structure Inside Full Text Predicate

More complex structure restrictions can be applied with the INPATH operator and an XPath expression. This query uses table PURCHASE_ORDERS and index po_index-path-section:

```
SELECT ID
FROM PURCHASE_ORDERS
WHERE CONTAINS( DOC, 'electric INPATH (/purchaseOrder/items/item/comment)' )>0 ;
```

Overview of the ora:contains XPath Function

Function ora:contains can be used in an XPath expression in a call to existsNode, extract, or extractValue to further restrict the structural search with a Full Text predicate. Function ora:contains returns a positive integer when the input_text matches text_query, and zero otherwise.

In this version, input_text must evaluate to a single text node or an attribute. The syntax and semantics of text_query in ora:contains are the same as text_query in CONTAINS, except that in ora:contains the text_query cannot include any structure operators (WITHIN, INPATH, or HASPATH). Function ora:contains extends XPath through a standard mechanism: it is a user-defined function in the Oracle XML DB namespace.

Syntax

```
ora:contains(
         input_text      node*,
         text_query      string
         [,policy_name   string]
         [,policy_owner  string]
         )
RETURN NUMBER
```

Example 9–5 shows a call to ora:contains in the XPath parameter to existsNode. Note that the third parameter (the XML DB namespace) is required. This example uses table PURCHASE_ORDERS_xmltype.

Example 9–5 ora:contains with an Arbitrarily Complex Text Query

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype
```

```
WHERE existsNode( DOC,
         '/purchaseOrder/comment[ora:contains(text(),
         "($lawns AND wild) OR flamingo")>0]',
         'xmlns:ora="http://xmlns.oracle.com/xdb"'
       ) = 1 ;
```

See Also: "ora:contains XPath Function" on page 9-19 for more on the `ora:contains` XPath function

Comparison of CONTAINS and ora:contains

The `CONTAINS` SQL function:

- Needs a `CONTEXT` index to run

  If there is no index, then you get an error.
- Does an indexed search and is generally very fast
- Returns a score (through the score operator)
- Can restrict a search using both Full Text and XML structure
- Restricts a search based on documents (rows in a table) rather than nodes
- Cannot be used for XML structure-based projection (pulling out parts of an XML document)

The ora:contains XPath function:

- Does not need an index to run, so it is very flexible
- Separates application logic from storing and indexing considerations
- Might do an unindexed search, so it might be resource-intensive
- Does not return a score
- Can restrict a search using Full Text in an XPath expression
- Can be used for XML structure-based projection (pulling out parts of an XML document)

Use `CONTAINS` when you want a fast, index-based Full Text search over XML documents, possibly with simple XML structure constraints. Use `ora:contains` when you need the flexibility of Full Text search in XPath (possibly without an index), or when you need to do projection, and you do not need a score.

CONTAINS SQL Function

This section contains these topics:

Full Text Search

The second argument to `CONTAINS`, `text_query`, is a string that specifies the Full Text search. `text_query` has its own language, based on the SQL/MM Full-Text

standard. The operators in the text_query language are documented in [Oracle Text Reference].

See Also:

- ISO/IEC 13249-2:2000, Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text, International Organization For Standardization, 2000
- *Oracle Text Reference* for more information on the operators in the `text_query` language

The examples in the rest of this section show some of the power of Full Text search. They use just a few of the available operators: Booleans (AND, OR, NOT) and stemming. The example queries search over a `VARCHAR2` column (`PURCHASE_ORDERS.doc`) with a text index (indextype `CTXSYS.CONTEXT`).

Boolean Operators: AND, OR, NOT

The `text_query` language supports arbitrary combinations of AND, OR and NOT. Precedence can be controlled using parentheses. The Boolean operators can be written as:

- AND, OR, NOT
- and, or, not
- &, |, ~

Example 9–6 CONTAINS Query with Simple Boolean

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'lawn AND wild' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index`.

Example 9–7 CONTAINS Query with Complex Boolean

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, '( (lawn OR garden) AND (wild OR flooded) )
        NOT(flamingo)' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index`.

See Also: *Oracle Text Reference* for a full list of the operators you can use in `CONTAINS` and `ora:contains`

Stemming: $

The `text_query` language supports stemmed search. Example 9–8 returns all documents that contain some word with the same linguistic stem as "lawns", so it will find "lawn" or "lawns". The stem operator is written as a dollar sign ($). There is no operator `STEM` or `stem`.

Example 9–8 CONTAINS Query with Stemming

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, '$(lawns)' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index.

Combining Boolean and Stemming Operators

operators in the text_query language can be arbitrarily combined, as shown in Example 9–9.

Example 9–9 CONTAINS Query with Complex Query Expression

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, '($lawns AND wild) OR flamingo' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index.

> **See Also:** *Oracle Text Reference* for a full list of text_query operators

Score

The CONTAINS function has an ancillary operator SCORE that can be used anywhere in the query. It is a measure of relevance, and it is especially useful when doing Full Text searches across large document sets. SCORE is typically returned as part of the query result, used in the ORDER BY clause, or both.

Syntax

```
SCORE( label       NUMBER )
RETURN NUMBER
```

In Example 9–10, SCORE(10) returns the score for each row in the result set. SCORE is the relevance of a row in the result set with respect to a particular CONTAINS call. A call to SCORE is linked to a call to CONTAINS by a LABEL (in this case the number 10).

Example 9–10 Simple CONTAINS Query with SCORE

```
SELECT SCORE(10), ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'lawn', 10 )>0
  AND SCORE(10)>2
  ORDER BY SCORE(10) DESC ;
```

This example uses table PURCHASE_ORDERS and index po_index.

SCORE always returns 0 if, for the corresponding CONTAINS, text_query does not match the input_text, according to the matching rules dictated by the text index. If the CONTAINS text_query does match the input_text, then SCORE will return a number greater than 0 and less than or equal to 100. This number indicates the relevance of the text_query to the input_text. A higher number means a better match.

If the CONTAINS text_query consists of only the HASPATH operator and a Text Path, the score will be either 0 or 100, because HASPATH tests for an exact match.

> **See Also:** *Oracle Text Reference* for details on how the score is calculated

Structure: Restricting the Scope of the Search

CONTAINS does a Full Text search across the whole document by default. In our examples, a search for "lawn" with no structure restriction will find all purchase orders with the word "lawn" anywhere in the purchase order.

Oracle offers three ways to restrict CONTAINS queries using XML structure:

- WITHIN
- INPATH
- HASPATH

> **Note:** For the purposes of this discussion, consider **section** to be the same as an **XML node**.

WITHIN

The WITHIN operator restricts a query to some section within an XML document. A search for purchase orders that contain the word "lawn" somewhere inside a comment section might use WITHIN. Section names are case-sensitive.

Example 9–11 WITHIN

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'lawn WITHIN comment' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index-path-section.

Nested WITHIN You can restrict the query further by nesting WITHIN. Example 9–12 finds all documents that contain the word "lawn" within a section "comment", where that occurrence of "lawn" is also within a section "item".

Example 9–12 Nested WITHIN

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, '(lawn WITHIN comment) WITHIN item' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index-path-section.

Example 9–12 returns no rows. Our sample purchase order does contain the word "lawn" within a comment. But the only comment within an item is "Confirm this is electric". So the nested WITHIN query will return no rows.

WITHIN Attributes You can also search within attributes. Example 9–13 finds all purchase orders that contain the word "10" in the orderDate attribute of a purchaseOrder element.

Example 9–13 WITHIN an Attribute

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, '10 WITHIN purchaseOrder@orderDate' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index-path-section.

Note that by default the minus sign ("-") is treated as a word-separator: "1999-10-20" is treated as the three words "1999", "10" and "20". So this query returns 1 row.

Text in an attribute is not a part of the main searchable document. If you search for "10" without qualifying the `text_query` with `WITHIN purchaseOrder@orderDate`, then you will get no rows.

You cannot search attributes in a nested `WITHIN`.

WITHIN and AND Suppose you want to find purchase orders that contain two words within a comment section: "lawn" and "electric". There can be more than one comment section in a `purchaseOrder`. So there are two ways to write this query, with two distinct results.

If you want to find purchase orders that contain both words, where each word occurs in **some comment section**, you would write a query like Example 9–14.

Example 9–14 WITHIN and AND: Two Words in Some Comment Section

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, '(lawn WITHIN comment) AND (electric WITHIN comment) '
 )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

If you run this query against the `purchaseOrder` data, then it returns 1 row. Note that the parentheses are not needed in this example, but they make the query more readable.

If you want to find purchase orders that contain both words, where both words occur **in the same comment**, you would write a query like Example 9–15.

Example 9–15 WITHIN and AND: Two Words in the Same Comment

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, '(lawn AND electric) WITHIN comment' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

Example 9–15 will return no rows. Example 9–16, which omits the parentheses around lawn AND electric, on the other hand, will return 1 row.

Example 9–16 WITHIN and AND: No Parentheses

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'lawn AND electric WITHIN comment' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

`WITHIN` has a higher operator precedence than `AND`, so Example 9–16 is parsed as Example 9–17.

Example 9–17 WITHIN and AND: Parentheses Illustrating Operator Precedence

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, ' lawn AND (electric WITHIN comment) ' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

Definition of Section The foregoing examples have used the `WITHIN` operator to search within a section. A section can be a:

- PATH or ZONE section

  This is a concatenation, in document order, of all text nodes that are descendants of a node, with whitespace separating the text nodes. To convert from a node to a ZONE section, you must serialize the node and replace all tags with whitespace. PATH sections have the same scope and behavior as ZONE sections, except that PATH sections support queries with `INPATH` and `HASPATH` operators.

- FIELD section

  This is the same as a ZONE section, except that repeating nodes in a document are concatenated into a single section, with whitespace as a separator.

- Attribute section

- Special section (sentence or paragraph)

> **See Also:** *Oracle Text Reference* for more information on special sections

INPATH

The `WITHIN` operator provides an easy and intuitive way to express simple structure restrictions in the `text_query`. For queries that use abundant XML structure, you can use the `INPATH` operator plus a Text Path instead of nested `WITHIN` operators.

The `INPATH` operator takes a `text_query` on the left and a Text Path, enclosed in parentheses, on the right. Example 9–18 finds `purchaseOrders` that contain the word "electric" in the path `/purchaseOrder/items/item/comment`.

Example 9–18 Structure Inside Full Text Predicate: INPATH

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS(DOC, 'electric INPATH (/purchaseOrder/items/item/comment)')>0;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

The scope of the search is the section indicated by the Text Path. If you choose a broader path, such as `/purchaseOrder/items`, you will still get 1 row returned, as shown in Example 9–19.

Example 9–19 Structure Inside Full Text Predicate: INPATH

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'electric INPATH (/purchaseOrder/items)' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

The Text Path The syntax and semantics of the Text Path are based on the w3c XPath 1.0 recommendation. Simple path expressions are supported (abbreviated syntax only), but functions are not. The following examples are meant to give the general flavor.

See Also:

- `http://www.w3.org/TR/xpath` for information on the w3c XPath 1.0 recommendation
- "Text Path BNF" on page 9-30 for the Text Path grammar

Example 9–20 finds all purchase orders that contain the word "electric" in a "comment" which is the direct child of an "item" with an attribute `partNum` equal to "872-AA", which in turn is the direct child of an "items", which is any number of levels down from the root node.

Example 9–20 INPATH with Complex Path Expression (1)

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'electric INPATH
      (//items/item[@partNum="872-AA"]/comment)' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

Example 9–21 finds all purchase orders that contain the word "lawnmower" in a third-level "item" (or any of its descendants) that has a "comment" descendant at any level. This query returns 1 row. Note that the scope of the query is **not** a "comment", but the set of "items" that have a "comment" as a descendant.

Example 9–21 INPATH with Complex Path Expression (2)

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'lawnmower INPATH (/*/*/item[.//comment])' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

Text Path Compared to XPath The Text Path language differs from the XPath language in the following ways:

- Not all XPath operators are included in the Text Path language.
- XPath built-in functions are not included in the Text Path language.
- Text Path language operators are case-insensitive.
- If you use "=" inside a filter (inside square brackets), matching follows text-matching rules.

  Rules for case-sensitivity, normalization, stopwords and whitespace depend on the text index definition. To emphasize this difference, this kind of equality is referred to here as text-equals.
- Namespace support is not included in the Text Path language.

  The name of an element, including a namespace prefix if it exists, is treated as a string. So two namespace prefixes that map to the same namespace URI will not be treated as equivalent in the Text Path language.
- In a Text Path the context is always the root node of the document.

  So in the purchaseOrder data `purchaseOrder/items/item`, `/purchaseOrder/items/item` and `./purchaseOrder/items/item` are equivalent.

- If you want to search within an attribute value, then the direct parent of the attribute must be specified (wildcards cannot be used).
- A Text Path may not end in a wildcard (*).

> **See Also:** "Text Path BNF" on page 9-30 for the Text Path grammar

Nested INPATH You can nest `INPATH` expressions. The context for the Text Path is always the root node. It is not changed by a nested `INPATH`.

Example 9–22 finds purchase orders that contain the word "electric" in a "comment" section at any level, where the occurrence of that word is also in an "items" section that is the direct child of the top-level `purchaseOrder`.

Example 9–22 Nested INPATH

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, '(electric INPATH (//comment)) INPATH
     (/purchaseOrder/items)' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

This nested `INPATH` query could be written more concisely as shown in Example 9–23.

Example 9–23 Nested INPATH Rewritten

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'electric INPATH (/purchaseOrder/items//comment)' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

HASPATH

The `HASPATH` operator takes only one operand: a Text Path, enclosed in parentheses, on the right. Use `HASPATH` when you want to find documents that contain a particular section in a particular path, possibly with an "=" predicate. Note that this is a path search rather than a Full Text search. You can check for existence of a section, or you can match the contents of a section, but you cannot do word searches. If your data is of type `XMLType`, then consider using `existsNode` instead of `HASPATH`.

Example 9–24 finds `purchaseOrders` that have some item that has a `USPrice`.

Example 9–24 Simple HASPATH

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'HASPATH (/purchaseOrder//item/USPrice)' )>0 ;
```

This example uses table `PURCHASE_ORDERS` and index `po_index-path-section`.

Example 9–25 finds `purchaseOrders` that have some item that has a `USPrice` that text-equals "148.95".

> **See Also:** "Text Path Compared to XPath" on page 9-11 for an explanation of text-equals

Example 9–25 HASPATH Equality

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'HASPATH (/purchaseOrder//item/USPrice="148.95")' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index-path-section.

HASPATH can be combined with other CONTAINS operators such as INPATH. Example 9–26 finds purchaseOrders that contain the word "electric" anywhere in the document AND have some "item" that has a USPrice that text-equals "148.95" AND contain "10" in the purchaseOrder attribute orderDate.

Example 9–26 HASPATH with Other Operators

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'electric AND HASPATH
     (/purchaseOrder//item/USPrice="148.95") AND 10 INPATH
     (/purchaseOrder/@orderDate)' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index-path-section.

Structure: Projecting the Result

The result of a SQL query with a CONTAINS predicate in the WHERE clause is always a set of rows (and possibly SCORE information), or a projection over the rows that match the query. If you want to return only a part of each XML document that satisfies the CONTAINS predicate, then use the SQL/XML extensions extract and extractValue. Note that extract and extractValue operate on the XMLType type, so the following examples use the table PURCHASE_ORDERS_xmltype.

> **See Also:** *Oracle XML DB Developer's Guide* for more information on extract and extractValue

Example 9–27 finds purchaseOrders that contain the word "electric" in a "comment" that is a descendant of the top-level purchaseOrder. Instead of returning the ID of the row for each result, extract is used to return only the "comment".

Example 9–27 Using Extract to Scope the Results of a CONTAINS Query

```
SELECT
  extract( DOC,
           '/purchaseOrder//comment',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) "Item Comment"
  FROM PURCHASE_ORDERS_xmltype
  WHERE CONTAINS( DOC, 'electric INPATH (/purchaseOrder//comment) ' )>0 ;
```

This example uses table PURCHASE_ORDERS_xmltype and index po_index_xmltype.

Note that the result of Example 9–27 is **two** instances of "comment". CONTAINS tells us which rows contain the word "electric" in a "comment" (the row with ID=1), and extract extracts all the instances of "comment" in the document at that row. There are two instances of "comment" in our purchaseOrder, and the query returns both of them.

This might not be what you want. If you want the query to return only the instances of "comment" that satisfy the CONTAINS predicate, then you must repeat that predicate in the extract. You do that with ora:contains, which is an XPath function.

Example 9–28 returns only the "comment" that matches the CONTAINS predicate.

Example 9–28 Using Extract Plus ora:contains to Project the Results of a CONTAINS Query

```
SELECT
  extract( DOC,
       '/purchaseOrder/items/item/comment[ora:contains(text(), "electric")>0]',
       'xmlns:ora="http://xmlns.oracle.com/xdb"') "Item Comment"
  FROM PURCHASE_ORDERS_xmltype
  WHERE CONTAINS( DOC, 'electric
       INPATH (/purchaseOrder/items/item/comment) ' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index-path-section.

Indexing

This section contains these topics:

- Introduction to the CONTEXT Index
- Effect of the CONTEXT Index on CONTAINS
- The CONTEXT Index: Preferences
- Introduction to Section Groups

Introduction to the CONTEXT Index

The Oracle general purpose Full Text indextype is the CONTEXT indextype, owned by the database user CTXSYS. To create a default Full Text index, use the regular SQL CREATE INDEX command, and add the clause INDEXTYPE IS CTXSYS.CONTEXT, as shown in Example 9–29.

Example 9–29 Simple CONTEXT Index on PURCHASE_ORDERS Table

```
CREATE INDEX po_index
  ON PURCHASE_ORDERS( DOC )
  INDEXTYPE IS ctxsys.CONTEXT ;
```

This example uses table PURCHASE_ORDERS.

You have many choices available when building a Full Text index. These choices are expressed as indexing **preferences**. To use an indexing preference, add the PARAMETERS clause to CREATE INDEX, as shown in Example 9–30.

See Also: "The CONTEXT Index: Preferences" on page 9-16

Example 9–30 Simple CONTEXT Index on PURCHASE_ORDERS Table with Path Section Group

```
CREATE INDEX po_index
  ON PURCHASE_ORDERS( DOC )
  INDEXTYPE IS ctxsys.CONTEXT
  PARAMETERS ( 'section group ctxsys.PATH_SECTION_GROUP' ) ;
```

This example uses table PURCHASE_ORDERS.

Oracle Text provides other indextypes, such as CTXCAT and CTXRULE, which are outside the scope of this chapter.

> **See Also:** *Oracle Text Reference* for more information on CONTEXT indexes

CONTEXT Index on XMLType Table You can build a CONTEXT index on any data that contains text. Example 9–29 creates a CONTEXT index on a VARCHAR2 column. The syntax to create a CONTEXT index on a column of type CHAR, VARCHAR, VARCHAR2, BLOB, CLOB, BFILE, XMLType, or URIType is the same. Example 9–31 creates a CONTEXT index on a column of type XMLType.

Example 9–31 Simple CONTEXT Index on PURCHASE_ORDERS_xmltype Table (Defaults to PATH_SECTION_GROUP)

```
CREATE INDEX po_index_xmltype
  ON PURCHASE_ORDERS_xmltype( DOC )
  INDEXTYPE IS ctxsys.CONTEXT ;
```

This example uses table PURCHASE_ORDERS_xmltype.

If you have a table of type XMLType, then you need to use object syntax to create the CONTEXT index as shown in Example 9–32.

Example 9–32 Simple CONTEXT Index on XMLType Table

```
CREATE INDEX po_index_xmltype_table
  ON PURCHASE_ORDERS_xmltype_table T ( value(T) )
  INDEXTYPE IS ctxsys.CONTEXT ;
```

This example uses table PURCHASE_ORDERS_xmltype.

You can then query the table using the syntax in Example 9–33.

Example 9–33 CONTAINS Query on XMLType Table

```
SELECT
  extract( value(T), '/purchaseOrder/@orderDate' ) "Order Date"
  FROM
    PURCHASE_ORDERS_xmltype_table T
    WHERE CONTAINS( value(T), 'electric INPATH (/purchaseOrder//comment) ' )>0 ;
```

This example uses table PURCHASE_ORDERS_xmltype_table and index po_index_xmltype_table.

Maintaining the CONTEXT Index The CONTEXT index, like most Full Text indexes, is asynchronous. When indexed data is changed, the CONTEXT index might not change until you take some action, such as calling a procedure to synchronize the index. There are a number of ways to manage changes to the CONTEXT index, including some options that are new for this release.

The CONTEXT index might get fragmented over time. A fragmented index uses more space, and it leads to slower queries. There are a number of ways to optimize (defragment) the CONTEXT index, including some options that are new for this release.

> **See Also:** *Oracle Text Reference* for more information on CONTEXT index maintenance

Roles and Privileges You do not need any special privileges to create a CONTEXT index. You need the CTXAPP role to create and delete preferences and to use the Oracle Text PL/SQL packages. You must also have EXECUTE privilege on the ctxsys package CTX_DDL.

Effect of the CONTEXT Index on CONTAINS

You must create an index of type CONTEXT in order to use the CONTAINS function. If you call the CONTAINS function, and the column given in the first argument does not have an index of type CONTEXT, then you will get an error.

The syntax and semantics of text_query depend on the choices you make when you build the CONTEXT index. For example:

- What counts as a word?
- Are very common words processed?
- What is a common word?
- Is the text search case-sensitive?
- Can the text search include themes (concepts) as well as keywords?

The CONTEXT Index: Preferences

A preference can be considered a collection of indexing choices. Preferences include section group, datastore, filter, wordlist, stoplist and storage. This section shows how to set up a lexer preference to make searches case-sensitive.

You can use the procedure CTX_DDL.CREATE_PREFERENCE (CTX_DDL.CREATE_STOPLIST) to create a preference. Override default choices in that preference group by setting attributes of the new preference, using the CTX_DDL.SET_ATTRIBUTE procedure. Then use the preference in a CONTEXT index by including <preference type> <preference_name> in the PARAMETERS string of CREATE INDEX.

Once a preference has been created, you can use it to build any number of indexes.

Making Search Case-Sensitive Full Text searches with CONTAINS are **case-insensitive** by default. That is, when matching words in text_query against words in the document, case is not considered. Section names and attribute names, however, are always **case-sensitive**.

If you want Full Text searches to be case-sensitive, then you need to make that choice when building the CONTEXT index. Example 9–34 returns 1 row, because "HURRY" in text_query matches "Hurry" in the purchaseOrder with the default case-insensitive index.

Example 9–34 CONTAINS: Default Case Matching

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'HURRY INPATH (/purchaseOrder/comment)' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index-path-section.

Example 9–35 creates a new lexer preference my_lexer, with the attribute mixed_case set to TRUE. It also sets printjoin characters to "-" and "!" and ",". You can use the same preferences for building CONTEXT indexes and for building policies.

See Also: *Oracle Text Reference* for a full list of lexer attributes

Example 9–35 Create a Preference for Mixed Case

```
BEGIN
  Ctx_Ddl.Create_Preference (
    preference_name  =>  'my_lexer',
    object_name      =>  'BASIC_LEXER'
    ) ;

  Ctx_Ddl.Set_Attribute (
    preference_name  =>  'my_lexer',
    attribute_name   =>  'mixed_case',
    attribute_value  =>  'TRUE'
    ) ;

  Ctx_Ddl.Set_Attribute (
    preference_name  =>  'my_lexer',
    attribute_name   =>  'printjoins',
    attribute_value  =>  '-,!'
    ) ;

END ;
/
```

Example 9–36 builds a CONTEXT index using the new my_lexer lexer preference.

Example 9–36 CONTEXT Index on PURCHASE_ORDERS Table, Mixed Case

```
CREATE INDEX po_index
  ON PURCHASE_ORDERS( DOC )
  INDEXTYPE IS ctxsys.context
  PARAMETERS( 'lexer my_lexer section group ctxsys.PATH_SECTION_GROUP' ) ;
```

This example uses table PURCHASE_ORDERS and preference: preference-case-mixed.

Example 9–36 returns no rows, because "HURRY" in text_query no longer matches "Hurry" in the purchaseOrder. Example 9–37 returns 1 row, because the text_query term "Hurry" exactly matches the word "Hurry" in the purchaseOrder.

Example 9–37 CONTAINS: Mixed (Exact) Case Matching

```
SELECT ID
  FROM PURCHASE_ORDERS
  WHERE CONTAINS( DOC, 'Hurry INPATH (/purchaseOrder/comment)' )>0 ;
```

This example uses table PURCHASE_ORDERS and index po_index-case-mixed.

Introduction to Section Groups

One of the choices you make when creating a CONTEXT index is section group. A section group instance is based on a section group type. The section group type specifies the kind of structure in your documents, and how to index (and therefore search) that structure. The section group instance may specify which structure elements are indexed. Most users will either take the default section group or use a pre-defined section group.

Choosing a Section Group Type The section group types that are useful in XML searching are:

- PATH_SECTION_GROUP

Choose this when you want to use WITHIN, INPATH and HASPATH in queries, and you want to be able to consider all sections to scope the query.

- XML_SECTION_GROUP

  Choose this when you want to use WITHIN, but not INPATH and HASPATH, in queries, and you want to be able to consider only explicitly-defined sections to scope the query. XML_SECTION_GROUP section group type supports FIELD sections in addition to ZONE sections. In some cases FIELD sections offer significantly better query performance.

- AUTO_SECTION_GROUP

  Choose this when you want to use WITHIN, but not INPATH and HASPATH, in queries, and you want to be able to consider most sections to scope the query. By default all sections are indexed (available for query restriction). You can specify that some sections are **not** indexed (by defining STOP sections).

- NULL_SECTION_GROUP

  Choose this when defining no XML sections.

Other section group types include:

- BASIC_SECTION_GROUP
- HTML_SECTION_GROUP
- NEWS_SECTION_GROUP

Oracle recommends that most users with XML Full Text search requirements use PATH_SECTION_GROUP. Some users may prefer XML_SECTION_GROUP with FIELD sections. This choice will generally give better query performance and a smaller index, but it is limited to documents with fielded structure (searchable nodes are all non-repeating leaf nodes).

> **See Also:** *Oracle Text Reference* for a detailed description of the XML_SECTION_GROUP section group type

Choosing a Section Group When choosing a section group to use with your index, you can choose a supplied section group, take the default, or create a new section group based on the section group type you have chosen.

There are supplied section groups for section group types PATH_SECTION_GROUP, AUTO_SECTION_GROUP, and NULL_SECTION_GROUP. The supplied section groups are owned by CTXSYS and have the same name as their section group types. For example, the supplied section group of section group type PATH_SECTION_GROUP is CTXSYS.PATH_SECTION_GROUP.

There is no supplied section group for section group type XML_SECTION_GROUP, because a default XML_SECTION_GROUP would be empty and therefore meaningless. If you want to use section group type XML_SECTION_GROUP, then you must create a new section group and specify each node that you want to include as a section.

When you create a CONTEXT index on data of type XMLType, the default section group is the supplied section group CTXSYS.PATH_SECTION_GROUP. If the data is VARCHAR or CLOB, then the default section group is CTXSYS.NULL_SECTION_GROUP.

> **See Also:** *Oracle Text Reference* for instructions on creating your own section group

To associate a section group with an index, add `section group <section group name>` to the `PARAMETERS` string, as in Example 9–38.

Example 9–38 Simple CONTEXT Index on PURCHASE_ORDERS Table with Path Section Group

```
CREATE INDEX po_index
  ON PURCHASE_ORDERS( DOC )
  INDEXTYPE IS ctxsys.CONTEXT
  PARAMETERS ( 'section group ctxsys.PATH_SECTION_GROUP' ) ;
```

This example uses table `PURCHASE_ORDERS`.

ora:contains XPath Function

Function `ora:contains` is an Oracle-defined XPath function for use in the XPath argument to the SQL/XML functions `existsNode`, `extract`, and `extractValue`.

> **Note:** These functions are not yet a part of the SQL/XML standard. But these functions or very similar functions are expected to be part of a future version of SQL/XML.

The `ora:contains` function name consists of a name (`contains`) plus a namespace prefix (`ora:`). When you use `ora:contains` in `existsNode`, `extract` or `extractValue` you must also supply a namespace mapping parameter, `xmlns:ora="http://xmlns.oracle.com/xdb"`.

Full Text Search

The `ora:contains` argument `text_query` is a string that specifies the Full Text search. The `ora:contains` `text_query` is the same as the `CONTAINS` `text_query`, with the following restrictions:

- `ora:contains` `text_query` must not include the structure operators `WITHIN`, `INPATH`, or `HASPATH`
- `ora:contains` `text_query` may include the score weighting operator `weight(*)`, but weights will be ignored

If you include any of the following in the `ora:contains` `text_query`, the query cannot use a `CONTEXT` index:

- Score-based operators `MINUS(-)` or `threshold(>)`
- Selective, corpus-based expansion operators `FUZZY(?)` or `soundex(!)`

> **See Also:** "Query-Rewrite and the CONTEXT Index" on page 9-28

Example 9–39 shows a Full Text search using an arbitrary combination of Boolean operators and $ (stemming).

Example 9–39 ora:contains with an Arbitrarily Complex Text Query

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype
  WHERE existsNode( DOC,
           '/purchaseOrder/comment[ora:contains(text(),
           "($lawns AND wild) OR flamingo")>0]',
```

```
    'xmlns:ora="http://xmlns.oracle.com/xdb"'
  ) = 1 ;
```

This example uses table PURCHASE_ORDERS_xmltype.

See Also:

- "Full Text Search" on page 9-5 for a description of Full Text operators
- *Oracle Text Reference* for a full list of the operators you can use in CONTAINS and ora:contains

Matching rules are defined by the policy <policy_owner>.<policy_name>. If policy_owner is absent, then the policy owner defaults to the current user. If both policy_name and policy_owner are absent, then the policy defaults to CTXSYS.DEFAULT_POLICY_ORACONTAINS.

Score

ora:contains is an XPath function that returns a number. It returns a positive number if the text_query matches the input_text. Otherwise it returns zero. ora:contains does not return a score.

Structure: Restricting the Scope of the Query

When you use ora:contains in an XPath expression, the scope is defined by input_text. This argument is evaluated in the current XPath context. If the result is a single text node or an attribute, then that node is the target of the ora:contains search. If input_text does not evaluate to a single text node or an attribute, an error is raised.

The policy determines the matching rules for ora:contains. The section group associated with the default policy for ora:contains is of type NULL_SECTION_GROUP.

ora:contains can be used anywhere in an XPath expression, and its input_text argument can be any XPath expression that evaluates to a single text node or an attribute.

Structure: Projecting the Result

If you want to return only a part of each XML document, then use extract to project a node sequence or extractValue to project the value of a node.

Example 9–40 returns the orderDate for each purchaseOrder that has some comment that contains the word "electric".

Example 9–40 ora:contains in existsNode Plus Extract

```
SELECT extract( DOC, '/purchaseOrder/@orderDate' ) "Order date"
  FROM PURCHASE_ORDERS_xmltype
  WHERE existsNode( DOC,
           '/purchaseOrder/comment[ora:contains(text(), "lawn")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"' ) = 1 ;
```

This example uses table PURCHASE_ORDERS_xmltype.

In Example 9–40 `existsNode` restricts the result to rows (documents) where the `purchaseOrder` includes some comment that contains the word "electric". `extract` then returns the `PurchaseOrder` attribute `orderDate` from those `purchaseOrders`. Note that if we extracted `//comment` we would get both comments from the sample document, not just the comment that matched the `WHERE` clause.

> **See Also:** Example 9–27, "Using Extract to Scope the Results of a CONTAINS Query" on page 9-13

Policies

The `CONTEXT` index on a column determines the semantics of `CONTAINS` queries on that column. Because `ora:contains` does not rely on a supporting index, some other means must be found to provide many of the same choices when doing `ora:contains` queries. A policy is a collection of preferences that can be associated with an `ora:contains` query to give the same sort of semantic control as the indexing choices give to the `CONTAINS` user.

Introduction to Policies

When using `CONTAINS`, indexing preferences affect the semantics of the query. You create a preference, using the package `CTX_DDL.CREATE_PREFERENCE` (or `CTX_DDL.CREATE_STOPLIST`). You override default choices by setting attributes of the new preference, using the `CTX_DDL.SET_ATTRIBUTE` procedure. Then you use the preference in a `CONTEXT` index by including `preference_type preference_name` in the `PARAMETERS` string of `CREATE INDEX`.

> **See Also:** "The CONTEXT Index: Preferences" on page 9-16

Because `ora:contains` does not have a supporting index, a different mechanism is needed to apply preferences to a query. That mechanism is called a policy, consisting of a collection of preferences, and it is used as a parameter to `ora:contains`.

Policy Example: Supplied Stoplist Example 9–41 creates a policy with an empty stopwords list.

Example 9–41 Create a Policy to Use with ora:contains

```
BEGIN
  Ctx_Ddl.Create_Policy (
    policy_name  =>  'my_nostopwords_policy',
    stoplist     =>  'ctxsys.EMPTY_STOPLIST'
    ) ;
END ;
/
```

For simplicity, this policy consists of an empty stoplist, which is owned by the user `ctxsys`. You could create a new stoplist to include in this policy, or you could reuse a stoplist (or lexer) definition that you created for a `CONTEXT` index.

Refer to this policy in any `ora:contains` to search for all words, including the most common ones (stopwords). Example 9–42 returns 0 comments, because "is" is a stopword by default and cannot be queried.

Example 9–42 Query on a Common Word with ora:contains

```
SELECT ID
```

```
  FROM PURCHASE_ORDERS_xmltype
  WHERE existsNode( DOC,
           '/purchaseOrder/comment[ora:contains(text(), "is")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1 ;
```

This example uses table `PURCHASE_ORDERS_xmltype`.

Example 9–43 uses the policy created in Example 9–41 to specify an empty stopword list. This query finds "is" and returns 1 comment.

Example 9–43 Query on a Common Word with ora:contains and Policy my_nostopwords_policy

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype
  WHERE existsNode( DOC,
           '/purchaseOrder/comment[ora:contains(text(),
           "is", "my_nostopwords_policy")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"' ) = 1 ;
```

This example uses table `PURCHASE_ORDERS_xmltype` and policy `my_nostopwords_policy`.

Effect of Policies on ora:contains

The `ora:contains` policy affects the matching semantics of `text_query`. The `ora:contains` policy may include a lexer, stoplist, wordlist preference, or any combination of these. Other preferences that can be used to build a `CONTEXT` index are not applicable to `ora:contains`. The effects of the preferences are as follows:

- The wordlist preference tweaks the semantics of the stem operator.
- The stoplist preference defines which words are too common to be indexed (searchable).
- The lexer preference defines how words are tokenized and matched. For example, it defines which characters count as part of a word and whether matching is case-sensitive.

See Also:

- "Policy Example: Supplied Stoplist" on page 9-21 for an example of building a policy with a predefined stoplist
- "Policy Example: User-Defined Lexer" on page 9-22 for an example of a case-sensitive policy

Policy Example: User-Defined Lexer When you search for a document that contains a particular word, you usually want the search to be **case-insensitive**. If you do a search that is **case-sensitive**, then you will often miss some expected results. For example, if you search for `purchaseOrders` that contain the phrase "baby monitor", then you would not expect to miss our example document just because the phrase is written "Baby Monitor".

Full Text searches with `ora:contains` are **case-insensitive** by default. Section names and attribute names, however, are always **case-sensitive**.

If you want Full Text searches to be case-sensitive, then you need to make that choice when you create a policy. You can use this procedure:

1. Create a preference using the procedure CTX_DDL.CREATE_PREFERENCE (or CTX_DDL.CREATE_STOPLIST).
2. Override default choices in that preference object by setting attributes of the new preference, using the CTX_DDL.SET_ATTRIBUTE procedure.
3. Use the preference as a parameter to CTX_DDL.CREATE_POLICY.
4. Use the policy name as the third argument to ora:contains in a query.

Once you have created a preference, you can reuse it in other policies or in CONTEXT index definitions. You can use any policy with any ora:contains query.

Example 9–44 returns 1 row, because "HURRY" in text_query matches "Hurry" in the purchaseOrder with the default case-insensitive index.

Example 9–44 ora:contains, Default Case-Sensitivity

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype
  WHERE existsNode( DOC,
           '/purchaseOrder/comment[ora:contains(text(), "HURRY")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1 ;
```

This example uses table PURCHASE_ORDERS_xmltype.

Example 9–45 creates a new lexer preference my_lexer, with the attribute mixed_case set to TRUE. It also sets printjoin characters to "-" and "!" and ",". You can use the same preferences for building CONTEXT indexes and for building policies.

See Also: *Oracle Text Reference* for a full list of lexer attributes

Example 9–45 Create a Preference for Mixed Case

```
BEGIN

  Ctx_Ddl.Create_Preference (
    preference_name  =>  'my_lexer',
    object_name      =>  'BASIC_LEXER'
    ) ;

  Ctx_Ddl.Set_Attribute (
    preference_name  =>  'my_lexer',
    attribute_name   =>  'mixed_case',
    attribute_value  =>  'TRUE'
    ) ;

  Ctx_Ddl.Set_Attribute (
    preference_name  =>  'my_lexer',
    attribute_name   =>  'printjoins',
    attribute_value  =>  '-,!'
    ) ;
END ;
/
```

Example 9–46 creates a new policy my_policy and specifies only the lexer. All other preferences are defaulted.

Example 9–46 Create a Policy with Mixed Case (Case-Insensitive)

```
BEGIN
```

```
  Ctx_Ddl.Create_Policy
  (
    policy_name  => 'my_policy',
    lexer        => 'my_lexer'
 ) ;

END ;
/
```

This example uses preference-case-mixed.

Example 9–47 uses the new policy in a query. It returns no rows, because "HURRY" in `text_query` no longer matches "Hurry" in the `purchaseOrder`.

Example 9–47 ora:contains, Case-Sensitive (1)

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype
  WHERE existsNode( DOC,
           '/purchaseOrder/comment[ora:contains(text(),
           "HURRY", "my_policy")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1 ;
```

This example uses table `PURCHASE_ORDERS_xmltype`.

Example 9–48 returns 1 row, because the `text_query` term "Hurry" exactly matches the word "Hurry" in the `purchaseOrder`.

Example 9–48 ora:contains, Case-Sensitive (2)

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype
  WHERE existsNode( DOC,
           '/purchaseOrder/comment[ora:contains(text(), "is going wild")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1 ;
```

This example uses table `PURCHASE_ORDERS_xmltype`.

Policy Defaults

The policy argument to `ora:contains` is optional. If it is omitted, then the query uses the default policy `CTXSYS.DEFAULT_POLICY_ORACONTAINS`.

When you create a policy for use with `ora:contains`, you do not need to specify every preference. In Example 9–46 on page 9-23, for example, only the lexer preference was specified. For the preferences that are not specified, `CREATE_POLICY` uses the default preferences:

- `CTXSYS.DEFAULT_LEXER`
- `CTXSYS.DEFAULT_STOPLIST`
- `CTXSYS.DEFAULT_ WORDLIST`

Creating a policy follows copy semantics for preferences and their attributes, just as creating a `CONTEXT` index follows copy semantics for index metadata.

ora:contains Performance

The `ora:contains` XPath function does not depend on a supporting index. `ora:contains` is very flexible. But if you use it to search across large amounts of data without an index, then it can also be resource-intensive. In this section we discuss how to get the best performance from queries that include XPath expressions with `ora:contains`.

> **Note:** Function-based indexes can also be very effective in speeding up XML queries, but they are not generally applicable to Text queries.

The examples in this section use table `PURCHASE_ORDERS_xmltype_big`. This has the same table structure and XML Schema as `PURCHASE_ORDERS_xmltype`, but it has around 1,000 rows. Each row has a unique ID (in the "id" column), and some different text in `/purchaseOrder/items/item/comment`. Where an execution plan is shown, it was produced using the SQL*Plus `AUTOTRACE`. Execution plans can also be produced using SQL `trace` and `tkprof`. A description of `AUTOTRACE`, SQL `trace` and `tkprof` is outside the scope of this chapter.

This section contains these topics:

- Use a Primary Filter in the Query
- Use a CTXXPath Index
- Query-Rewrite and the CONTEXT Index

Use a Primary Filter in the Query

Because `ora:contains` is relatively expensive to process, Oracle recommends that you write queries that include a primary filter wherever possible. This will minimize the number of rows actually processed by `ora:contains`.

Example 9–49 examines every row in the table (does a full table scan), as we can see from the Plan in Example 9–50. In this example, `ora:contains` is evaluated for every row.

Example 9–49 ora:contains in existsNode, Big Table

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype_big
  WHERE existsNode( DOC,
           '/purchaseOrder/items/item/comment[ora:contains(text(),
           "constitution")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1 ;
```

Example 9–50 Explain Plan: existsNode

```
Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE
1     0   TABLE ACCESS (FULL) OF 'PURCHASE_ORDERS_XMLTYPE_BIG' (TABLE)
```

If you create an index on the ID column, as shown in Example 9–51, and add a selective ID predicate to the query, as shown in Example 9–52, then it is apparent from

Example 9–53 that Oracle will drive off the ID index. `ora:contains` will be executed only for the rows where the ID predicate is true (where ID is less than 5).

Example 9–51 B-Tree Index on ID

```
CREATE INDEX id_index
  ON PURCHASE_ORDERS_xmltype_big( ID ) ;
```

This example uses table PURCHASE_ORDERS.

Example 9–52 ora:contains in existsNode, Mixed Query

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype_big
  WHERE existsNode( DOC,
           '/purchaseOrder/items/item/comment[ora:contains(text(),
           "constitution")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1
         AND id>5 ;
```

Example 9–53 Explain Plan: existsNode

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0    TABLE ACCESS (BY INDEX ROWSELECT ID) OF 'PURCHASE_ORDERS_XMLTYPE_BIG' (TABLE)
   2    1         INDEX (RANGE SCAN) OF 'SELECT ID_INDEX' (INDEX)
```

Use a CTXXPath Index

The CTXXPATH index can be used as a primary filter for `existsNode`. CTXXPATH is not related to `ora:contains`. CTXXPATH can be a primary filter for any `existsNode` query.

The CTXXPATH index stores enough information about a document to produce a superset of the results of an XPath expression. For an `existsNode` query it is often helpful to interrogate the CTXXPATH index and then apply `existsNode` to that superset, rather than applying `existsNode` to each document in turn.

Example 9–54 produces the execution plan shown in Example 9–55.

Example 9–54 ora:contains in existsNode, Big Table

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype_big
  WHERE existsNode( DOC,
           '/purchaseOrder/items/item/comment[ora:contains(text(),
           "constitution")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1 ;
```

Example 9–55 Explain Plan: existsNode

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0    TABLE ACCESS (FULL) OF 'PURCHASE_ORDERS_XMLTYPE_BIG' (TABLE)
```

Now create a CTXXPATH index on the DOC column, as shown in Example 9–56. You can create a CTXXPATH index and a CONTEXT index on the same column.

Example 9–56 Create a CTXXPATH Index on PURCHASE_ORDERS_xmltype_big(DOC)

```
CREATE INDEX doc_xpath_index
  ON PURCHASE_ORDERS_xmltype_big( DOC )
  INDEXTYPE IS ctxsys.CTXXPATH ;
```

Run Example 9–54 again and you will see from the plan, shown in Example 9–57, that the query now uses the CTXXPATH index.

Example 9–57 Explain Plan: existsNode with CTXXPATH Index

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=2044)
   1    0     TABLE ACCESS (BY INDEX ROWSELECT ID) OF 'PURCHASE_ORDERS_XMLTYPE_BIG' (TABLE)
                (Cost=2 Card=1 Bytes=2044)

   2    1          DOMAIN INDEX OF 'DOC_XPATH_INDEX' (INDEX (DOMAIN))
                     (Cost=0)
```

When to Use CTXXPATH CTXXPATH processes only a part of the XPath expression, to give a guaranteed superset (a first-pass estimate) of the results of XPath evaluation.

CTXXPATH does not process:

- Functions, including `ora:contains`
- Range operators: <=, <, >=, >
- '..', "|"
- Attribute following '.', '*' or '//'
- Predicate following '.' or '*'
- '.' or '*' at the end of a path
- Any node with unabbreviated XPath syntax

So in Example 9–54, the CTXXPATH index cannot return results for `/purchaseOrder/items/item/comment[ora:contains(., "constitution")>0]`, because it cannot process the function `ora:contains`. But the CTXXPATH index **can** act as a primary filter by returning all documents that contain the path `/purchaseOrder/items/item/comment`. By calculating this superset of results, CTXXPATH can significantly reduce the number of documents considered by `existsNode` in this case.

There are two situations where a CTXXPATH index will give a significant performance boost:

- If the document collection is heterogeneous, then knowing which documents contain the path (some `purchaseOrder` with some `items` child with some `item` child with some `comment` child) is enough to significantly reduce the documents considered by `existsNode`.
- If many of the queries include XPath expressions with equality predicates rather than range predicates or functions (such as Example 9–58), then CTXXPATH will process those predicates and therefore will be a useful primary filter. CTXXPATH handles both string and number equality predicates.

Example 9–58 Equality Predicate in XPath, Big Table

```
SELECT count(*)
  FROM PURCHASE_ORDERS_xmltype_big
  WHERE existsNode( DOC,
           '/purchaseOrder/items/item[USPrice=148.9500]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1 ;
```

If you are not sure that CTXXPATH will be useful, then create a CTXXPATH index and gather statistics on it, as shown in Example 9–59. With these statistics in place, the Oracle Cost Based Optimizer can make an informed choice about whether to use the CTXXPATH index or to ignore it.

Example 9–59 Gathering Index Statistics

```
BEGIN
      DBMS_STATS.GATHER_INDEX_STATS (
        ownname  =>  'test',
        indname  =>  'doc_xpath_index'
        ) ;
END;
/
```

This example uses `index-ctxxpath-1`.

Maintaining the CTXXPATH Index The CTXXPATH index, like the CONTEXT index, is asynchronous. When indexed data is changed, the CTXXPATH index might not change until you take some action, such as calling a procedure to synchronize the index. There are a number of ways to manage changes to the CTXXPATH index, including some options that are new for this release.

If the CTXXPATH index is not kept in synch with the data, then the index gradually becomes less efficient. The CTXXPATH index still calculates a superset of the true result, by adding all unsynchronized (unindexed) rows to the result set. So `existsNode` must process all the rows identified by the CTXXPATH index plus all unsynchronized (unindexed) rows.

The CTXXPATH index may get fragmented over time. A fragmented index uses more space and leads to slower queries. There are a number of ways to optimize (defragment) the CTXXPATH index, including some options that are new for this release.

> **See Also:** *Oracle Text Reference* for information on CTXXPATH index maintenance

Query-Rewrite and the CONTEXT Index

`ora:contains` does not rely on a supporting index. But under some circumstances an `ora:contains` may use an existing CONTEXT index for better performance.

Introducing Query-Rewrite Oracle will, in some circumstances, rewrite a SQL/XML query into an object-relational query. This is done as part of query optimization and is transparent to the user. Two of the benefits of query-rewrite are:

- The re-written query can directly access the underlying object-relational tables instead of processing the whole XML document.
- The re-written query can make use of any available indexes.

Query-rewrite is a performance optimization. Query-rewrite only makes sense if the XML data is stored object-relationally, which in turn requires the XML data to be Schema-based.

> **See Also:** Chapter 1, "Introducing Oracle XML DB" for a full description of the query-rewrite process

From Documents to Nodes Consider Example 9–60, a simple `ora:contains` query. To naively process the XPath expression in this query, each cell in the `DOC` column must be considered, and each cell must be tested to see if it matches `/purchaseOrder/items/item/comment[ora:contains(text(), "electric")>0]`.

Example 9–60 ora:contains in existsNode

```
SELECT ID
  FROM PURCHASE_ORDERS_xmltype
  WHERE existsNode( DOC,
           '/purchaseOrder/items/item/comment[ora:contains(text(),
           "electric")>0]',
           'xmlns:ora="http://xmlns.oracle.com/xdb"'
         ) = 1 ;
```

This example uses table `PURCHASE_ORDERS_xmltype`.

But if `DOC` is schema-based, and the `purchaseOrder` documents are physically stored in object-relational tables, then it makes sense to go straight to the `/purchaseOrder/items/item/comment` column (if one exists) and test each cell there to see if it matches "electric".

This is the first query-rewrite step. If the first argument to `ora:contains` (`text_input`) maps to a single relational column, then `ora:contains` executes against that column. Even if there are no indexes involved, this can significantly improve query performance.

From ora:contains to CONTAINS As noted in "From Documents to Nodes" on page 9-29, Oracle may rewrite a query so that an XPath expression in `existsNode` may be resolved by applying `ora:contains` to some underlying column instead of applying the whole XPath to the whole XML document. In this section it will be shown how that query might make use of a `CONTEXT` index on the underlying column.

If you are running `ora:contains` against a text node or attribute that maps to a column with a `CONTEXT` index on it, why would you **not** use that index? One powerful reason is that a re-written query should give the same results as the original query. To ensure consistent results, the following conditions must be true before a `CONTEXT` index can be used.

First, the `ora:contains` target (`input_text`) must be either a single text node whose parent node maps to a column or an attribute that maps to a column. The column must be a single relational column (possibly in a nested table).

Second, as noted in "Policies" on page 9-21, the indexing choices (for `CONTAINS`) and policy choices (for `ora:contains`) affect the semantics of queries. A simple mismatch might be that the index-based `CONTAINS` would do a **case-sensitive** search, while `ora:contains` specifies a **case-insensitive** search. To ensure that the `ora:contains` and the rewritten `CONTAINS` have the same semantics, the `ora:contains` policy must exactly match the index choices of the `CONTEXT` index.

Both the ora:contains policy and the CONTEXT index must also use the NULL_SECTION_GROUP section group type. The default section group for an ora:contains policy is ctxsys.NULL_SECTION_GROUP.

Third, the CONTEXT index is generally asynchronous. If you add a new document that contains the word "dog", but do not synchronize the CONTEXT index, then a CONTAINS query for "dog" will not return that document. But an ora:contains query against the same data will. To ensure that the ora:contains and the rewritten CONTAINS will always return the same results, the CONTEXT index must be built with the TRANSACTIONAL keyword in the PARAMETERS string (see [Oracle Text Reference]).

See Also: *Oracle Text Reference*

Query-Rewrite: Summary A query with existsNode, extract or extractValue, where the XPath includes ora:contains, may be considered for query-rewrite if:

- The XML is schema-based
- The first argument to ora:contains (text_input) is either a single text node whose parent node maps to a column, or an attribute that maps to a column. The column must be a single relational column (possibly in a nested table).

The rewritten query will use a CONTEXT index if:

- There is a CONTEXT index on the column that the parent node (or attribute node) of text_input maps to.
- The ora:contains policy exactly matches the index choices of the CONTEXT index.
- The CONTEXT index was built with the TRANSACTIONAL keyword in the PARAMETERS string.

Query-rewrite can speed up queries significantly, especially if there is a suitable CONTEXT index.

Text Path BNF

```
HasPathArg           ::=   LocationPath
                         |  EqualityExpr
InPathArg            ::=   LocationPath
LocationPath         ::=   RelativeLocationPath
                         |  AbsoluteLocationPath
AbsoluteLocationPath ::=   ("/" RelativeLocationPath)
                         |  ("//" RelativeLocationPath)
RelativeLocationPath ::=   Step
                         |  (RelativeLocationPath "/" Step)
                         |  (RelativeLocationPath "//" Step)
Step                 ::=   ("@" NCName)
                         |  NCName
                         |  (NCName Predicate)
                         |  Dot
                         |  "*"
Predicate            ::=   ("[" OrExp "]")
                         |  ("[" Digit+ "]")
OrExpr               ::=   AndExpr
                         |  (OrExpr "or" AndExpr)
AndExpr              ::=   BooleanExpr
                         |  (AndExpr "and" BooleanExpr)
BooleanExpr          ::=   RelativeLocationPath
```

```
                        |  EqualityExpr
                        |  ("(" OrExpr ")")
                        |  ("not" "(" OrExpr ")")
EqualityExpr        ::=    (RelativeLocationPath "=" Literal)
                        |  (Literal "=" RelativeLocationPath)
                        |  (RelativeLocationPath "=" Literal)
                        |  (Literal "!=" RelativeLocationPath)
                        |  (RelativeLocationPath "=" Literal)
                        |  (Literal "!=" RelativeLocationPath)
Literal             ::=    (DoubleQuote [~"]* DoubleQuote)
                        |  (SingleQuote [~']* SingleQuote)
NCName              ::=    (Letter |  Underscore) NCNameChar*
NCNameChar          ::=    Letter
                        |  Digit
                        |  Dot
                        |  Dash
                        |  Underscore
Letter              ::=    ([a-z] | [A-Z])
Digit               ::=    [0-9]
Dot                 ::=    "."
Dash                ::=    "-"
Underscore          ::=    "_"
```

Example Support

This section contains these topics:

- Purchase Order po001.xml
- Create Table Statements
- An XML Schema for the Sample Data

Purchase Order po001.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="xmlschema/po.xsd" orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
```

```
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

Create Table Statements

Example 9–61 CREATE TABLE PURCHASE_ORDERS

```
CREATE TABLE PURCHASE_ORDERS (ID   NUMBER,
                              DOC  VARCHAR2(4000));
INSERT INTO PURCHASE_ORDERS  (ID,  DOC)
  VALUES (1,
          '<?xml version="1.0" encoding="UTF-8"?>
           <purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="xmlschema/po.xsd"
           orderDate="1999-10-20">
               <shipTo country="US">
                   <name>Alice Smith</name>
                   <street>123 Maple Street</street>
                   <city>Mill Valley</city>
                   <state>CA</state>
                   <zip>90952</zip>
               </shipTo>
               <billTo country="US">
                   <name>Robert Smith</name>
                   <street>8 Oak Avenue</street>
                   <city>Old Town</city>
                   <state>PA</state>
                   <zip>95819</zip>
               </billTo>
               <comment>Hurry, my lawn is going wild!</comment>
               <items>
                   <item partNum="872-AA">
                       <productName>Lawnmower</productName>
                       <quantity>1</quantity>
                       <USPrice>148.95</USPrice>
                       <comment>Confirm this is electric</comment>
                   </item>
                   <item partNum="926-AA">
                       <productName>Baby Monitor</productName>
                       <quantity>1</quantity>
                       <USPrice>39.98</USPrice>
                       <shipDate>1999-05-21</shipDate>
                   </item>
               </items>
           </purchaseOrder>') ;
COMMIT ;
```

Example 9–62 CREATE TABLE PURCHASE_ORDERS_xmltype

```
CREATE TABLE PURCHASE_ORDERS_xmltype (ID   NUMBER ,
                                      DOC  XMLType);
INSERT INTO PURCHASE_ORDERS_xmltype  (ID,  DOC)
  VALUES (1,
          XMLTYPE ('<?xml version="1.0" encoding="UTF-8"?>
```

```
                    <purchaseOrder
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:noNamespaceSchemaLocation="po.xsd"
                    orderDate="1999-10-20">
                        <shipTo country="US">
                            <name>Alice Smith</name>
                            <street>123 Maple Street</street>
                            <city>Mill Valley</city>
                            <state>CA</state>
                            <zip>90952</zip>
                        </shipTo>
                        <billTo country="US">
                            <name>Robert Smith</name>
                            <street>8 Oak Avenue</street>
                            <city>Old Town</city>
                            <state>PA</state>
                            <zip>95819</zip>
                        </billTo>
                        <comment>Hurry, my lawn is going wild!</comment>
                        <items>
                            <item partNum="872-AA">
                                <productName>Lawnmower</productName>
                                <quantity>1</quantity>
                                <USPrice>148.95</USPrice>
                                <comment>Confirm this is electric</comment>
                            </item>
                            <item partNum="926-AA">
                                <productName>Baby Monitor</productName>
                                <quantity>1</quantity>
                                <USPrice>39.98</USPrice>
                                <shipDate>1999-05-21</shipDate>
                            </item>
                        </items>
                    </purchaseOrder>'));
COMMIT ;
```

Example 9–63 CREATE TABLE PURCHASE_ORDERS_xmltype_table

```
CREATE TABLE PURCHASE_ORDERS_xmltype_table OF XMLType;
INSERT INTO PURCHASE_ORDERS_xmltype_table
  VALUES (
    XMLTYPE ('<?xml version="1.0" encoding="UTF-8"?>
              <purchaseOrder
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:noNamespaceSchemaLocation="xmlschema/po.xsd"
              orderDate="1999-10-20">
                  <shipTo country="US">
                      <name>Alice Smith</name>
                      <street>123 Maple Street</street>
                      <city>Mill Valley</city>
                      <state>CA</state>
                      <zip>90952</zip>
                  </shipTo>
                  <billTo country="US">
                      <name>Robert Smith</name>
                      <street>8 Oak Avenue</street>
                      <city>Old Town</city>
                      <state>PA</state>
                      <zip>95819</zip>
                  </billTo>
```

```
                    <comment>Hurry, my lawn is going wild!</comment>
                    <items>
                        <item partNum="872-AA">
                            <productName>Lawnmower</productName>
                            <quantity>1</quantity>
                            <USPrice>148.95</USPrice>
                            <comment>Confirm this is electric</comment>
                        </item>
                        <item partNum="926-AA">
                            <productName>Baby Monitor</productName>
                            <quantity>1</quantity>
                            <USPrice>39.98</USPrice>
                            <shipDate>1999-05-21</shipDate>
                        </item>
                    </items>
                </purchaseOrder>' ));
COMMIT ;
```

An XML Schema for the Sample Data

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:annotation>
      <xsd:documentation xml:lang="en">
       Purchase order schema for Example.com.
       Copyright 2000 Example.com. All rights reserved.
      </xsd:documentation>
   </xsd:annotation>
   <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
   <xsd:element name="comment" type="xsd:string"/>
   <xsd:complexType name="PurchaseOrderType">
      <xsd:sequence>
         <xsd:element name="shipTo" type="USAddress"/>
         <xsd:element name="billTo" type="USAddress"/>
         <xsd:element ref="comment" minOccurs="0"/>
         <xsd:element name="items" type="Items"/>
      </xsd:sequence>
      <xsd:attribute name="orderDate" type="xsd:date"/>
   </xsd:complexType>
   <xsd:complexType name="USAddress">
      <xsd:sequence>
         <xsd:element name="name" type="xsd:string"/>
         <xsd:element name="street" type="xsd:string"/>
         <xsd:element name="city" type="xsd:string"/>
         <xsd:element name="state" type="xsd:string"/>
         <xsd:element name="zip" type="xsd:decimal"/>
      </xsd:sequence>
      <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
   </xsd:complexType>
   <xsd:complexType name="Items">
      <xsd:sequence>
         <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="productName" type="xsd:string"/>
                  <xsd:element name="quantity">
                     <xsd:simpleType>
                        <xsd:restriction base="xsd:positiveInteger">
                           <xsd:maxExclusive value="100"/>
                        </xsd:restriction>
```

```
                  </xsd:simpleType>
               </xsd:element>
               <xsd:element name="USPrice" type="xsd:decimal"/>
               <xsd:element ref="comment" minOccurs="0"/>
               <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="partNum" type="SKU" use="required"/>
         </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
</xsd:complexType>
<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
   </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Part III

Using APIs for XMLType to Access and Operate on XML

Part III of this manual introduces you to ways you can use Oracle XML DB XMLType PL/SQL, Java, C APIs, and Oracle Data Provider for .NET (ODP.NET) to access and manipulate XML data. It contains the following chapters:

10

PL/SQL API for XMLType

This chapter describes the use of the APIs for `XMLType` in PL/SQL.

This chapter contains these topics:

Introducing PL/SQL APIs for XMLType

This chapter describes the PL/SQL Application Program Interfaces (APIs) for `XMLType`. These include the following:

- PL/SQL Document Object Model (DOM) API for XMLType (package DBMS_XMLDOM): For accessing `XMLType` objects. You can access both XML schema-based and non-schema-based documents. Before database startup, you must specify the read-from and write-to directories in 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.

  DOM is 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 APIs for programmatic access. DOM views the parsed document as a tree of objects.
- PL/SQL XML Parser API for XMLType (package DBMS_XMLPARSER): For accessing the contents and structure of XML documents.
- PL/SQL XSLT Processor for XMLType (package DBMS_XSLPROCESSOR): For transforming XML documents to other formats using XSLT.

PL/SQL APIs For XMLType Features

The PL/SQL APIs for `XMLType` allow you to perform the following tasks:

- Create `XMLType` tables, columns, and views
- Construct `XMLType` instances from data encoded in different character sets.
- Access `XMLType` data

- Manipulate `XMLType` data

See Also:

- "Oracle XML DB Features", for an overview of the Oracle XML DB architecture and new features.
- Chapter 4, "XMLType Operations"
- *Oracle XML API Reference*

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 when they are requested. It throws away previously-referenced sections of the document if memory usage grows too large. Lazy XML loading supports highly scalable applications that have many concurrent users needing to access large XML documents.

XMLType 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.

XMLType Supports Data in Different Character Sets.

`XMLType` instances can now be created from data encoded in any Oracle-supported character sets by using the PL/SQL `XMLType` constructor or the `createXML()` methods. The source XML data must be supplied using either `BFILE` or `BLOB` datatype. The encoding of the data is specified through the `csid` argument. When this argument is zero then the encoding of the source data is determined from the XML prolog as specified in Appendix F of the XML 1.0 Reference.

In addition, a new `getBlobVal()` method is provided to retrieve the XML contents in the requested character set.

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 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 DTD is a set of rules that define the allowable

structure of an XML document. DTDs are text files that derive their format from SGML and can either be included in an XML document by using the DOCTYPE element or by using an external file through a DOCTYPE reference. A DOM parser also provides methods for traversing the DOM tree and return data from it.

If you use the PL/SQL DOM API, then 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)

This section describes PL/SQL DOM API for XMLType. This is the DBMS_XMLDOM PL/SQL API.

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 Oracle9*i* release 1 (9.0.1), the XDK for PL/SQL implemented DOM Level 1.0 and parts of DOM Level 2.0.
- In Oracle9*i* release 2 (9.2) and Oracle Database 10*g* release 1 (10.1), the PL/SQL API for `XMLType` implements DOM Levels 1.0 and Level 2.0 Core, and is fully integrated in the database through extensions to the `XMLType` API.

The following briefly 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). CSS are a simple mechanism for adding style (fonts, colors, spacing, and so on) to Web documents.
- 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:

```
<img src="employee_jdoe.gif" ID="0123jdoe">
```

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 action 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 Oracle9*i* release 1(9.0.1). The only difference is that the validation occurs when the object document is manifested. For example, if lazy manifestation is used, then 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 10-6, 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 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 requirement 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 12, "Java API 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. Class generators accept 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.
- 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: *Oracle XML Developer's Kit Programmer's Guide*

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 10-8.
- 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, "XML Schema Storage and Query: The Basics" 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:**
>
> - *PL/SQL Packages and Types Reference*
> - *Oracle XML API Reference*

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 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 16, "XMLType Views"

PL/SQL DOM API for XMLType (DBMS_XMLDOM) Methods

Table 10–1 lists the PL/SQL DOM API for `XMLType` (`DBMS_XMLDOM`) methods supported in this release. Character data (CDATA) refers to the section in an XML document used to indicate the text that should not be parsed. 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.

Non-Supported DBMS_XMLDOM Methods in This Release

The following `DBMS_XMLDOM` methods are *not* supported in this release:

- `hasFeature`
- `getDocType`
- `setDocType`
- `writeExternalDTDToFile`
- `writeExternalDTDToBuffer`
- `writeExternalDTDToClob`

Table 10–1 lists additional methods supported in this release.

Table 10–1 Summary of 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.

Table 10–1 (Cont.) Summary of DBMS_XMLDOM Methods

Group/Method	Description
`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.
`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.
`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.

Table 10–1 (Cont.) Summary of DBMS_XMLDOM Methods

Group/Method	Description
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.
`getLocalName()`	Retrieves the local name of the attribute.
`getExpandedName()`	Retrieves the expanded name of the attribute.
`getName()`	Retrieves the name of the attribute.
`getSpecifiied()`	Tests if attribute was specified in the owning element.
`getValue()`	Retrieves the value of the attribute.
`setValue()`	Sets the value of the attribute.
CData 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. [Not supported in this release]

Table 10–1 (Cont.) Summary of DBMS_XMLDOM Methods

Group/Method	Description
Document fragment methods	--
`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.
`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`.

Table 10–1 (Cont.) Summary of DBMS_XMLDOM Methods

Group/Method	Description
`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.
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.

Table 10–1 (Cont.) Summary of DBMS_XMLDOM Methods

Group/Method	Description
`writeExternalDTDToFile()`	Writes the DTD of the document to a file. [Not supported in this release]
`writeExternalDTDToBuffer()`	Writes the DTD of the document to a buffer. [Not supported in this release]
`writeExternalDTDToClob()`	Writes the DTD of the document to a clob. [Not supported in this release]
`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.
`adoptNode()`	`FUNCTION adoptNode(doc DOMdocument, adoptednode domnode)RETURN DOMNode;` Adopts the given node: removes it from its owner document and adds it to the given document.
`createDocument()`	`FUNCTION createDocument (namspaceURI IN VARCHAR2, qualifiedName IN VARCHAR2, doctype IN DOMType) RETURN DocDocument;`
`getPrefix()`	`FUNCTION getPrefix(n DOMNode) RETURN VARCHAR2;`
`setPrefix()`	`PROCEDURE setPrefix (n DOMNode) RETURN VARCHAR2;`
`hasAttributes()`	`FUNCTION hasAttributes (n DOMNode) RETURN BOOLEAN;`
`getNamedItem()`	`FUNCTION getNamedItem (nnm DOMNamedNodeMap, name IN VARCHAR2, ns IN VARCHAR2) RETURN DOMNode;`
`setNamedItem()`	`FUNCTION getNamedItem (nnm DOMNamedNodeMap, arg IN DOMNode, ns IN VARCHAR2) RETURN DOMNode;`
`removeNamedItem()`	`FUNCTION removeNamedItem (nnm DOMNamesNodeMap, name in VARCHAR2, ns IN VARCHAR2) RETURN DOMNode;`
`getOwnerElement()`	`FUNCTION getOwnerElement (a DOMAttr) RETURN DOMElement;`

Table 10–1 (Cont.) Summary of DBMS_XMLDOM Methods

Group/Method	Description
getAttribute()	FUNCTION getAttribute (elem DOMElement, name IN VARCHAR2, ns IN VARCHAR2) RETURN VARCHAR2;
hasAttribute()	FUNCTION hasAttribute (elem DOMElement, name IN VARCHAR2) RETURN BOOLEAN;
hasAttribute()	FUNCTION hasAttribute (elem DOMElement, name IN VARCHAR2, ns IN VARCHAR2) RETURN BOOLEAN;
setAttribute()	PROCEDURE setAttribute (elem DOMElement, name IN VARCHAR2, newvalue IN VARCHAR2, ns IN VARCHAR2);
removeAttribute()	PROCEDURE removeAttribute (elem DOMElement, name IN VARCHAR2, ns IN VARCHAR2);
getAttributeNode()	FUNCTION getAttributeNode(elem DOMElement, name IN VARCHAR2, ns IN VARCHAR2) RETURN DOMAttr;
setAttributeNode()	FUNCTION setAttributeNode(elem DOMElement, newAttr IN DOMAttr, ns IN VARCHAR2) RETURN DOMAttr;
createElement()	FUNCTION createElement (doc DOMDocument, tagname IN VARCHAR2, ns IN VARCHAR2) RETURN DOMElement;
createAttribute()	FUNCTION createAttribute (doc DOMDocument, name IN VARCHAR2, ns IN VARCHAR2) RETURN DOMAttr;

PL/SQL DOM API for XMLType (DBMS_XMLDOM) Exceptions

The following lists the PL/SQL DOM API for XMLType (DBMS_XMLDOM) exceptions. For more information, see *Oracle XML Developer's Kit Programmer's Guide*.

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 10–2 shows the node types for XML and HTML and the allowed corresponding children node types.

Table 10–2 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
`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 Schema-Based XML Instances

Oracle Database has several extensions for character-set conversion and input and output to and from a file system. PL/SQL API for `XMLType` is optimized to operate on XML schema-based XML instances. Function `newDOMDocument()` constructs a DOM document handle, given an `XMLType` value.

A typical usage scenario would be for a PL/SQL application to:

1. Fetch or construct an `XMLType` instance
2. Construct a `DOMDocument` node over the `XMLType` instance
3. Use the DOM API to access and manipulate the XML data

Note: For `DOMDocument`, node types represent handles to XML fragments but do not represent the data itself.

For example, if you copy a node value, `DOMDocument` clones the handle to the same underlying data. Any data modified by one of the handles is visible when accessed by the other handle. The `XMLType` value from which the `DOMDocument` handle is constructed is the actual data, and reflects the results of all DOM operations on it.

DOM NodeList and NamesNodeMap Objects

`NodeList` and `NamedNodeMap` objects in the DOM are active; 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), then those changes are automatically propagated in the `NodeList`, without additional action from the user. Likewise, changes to a node in the tree are propagated throughout all references to that node in `NodeList` and `NamedNodeMap` objects.

The interfaces: `Text`, `Comment`, and `CDATASection`, all inherit from the `CharacterData` interface.

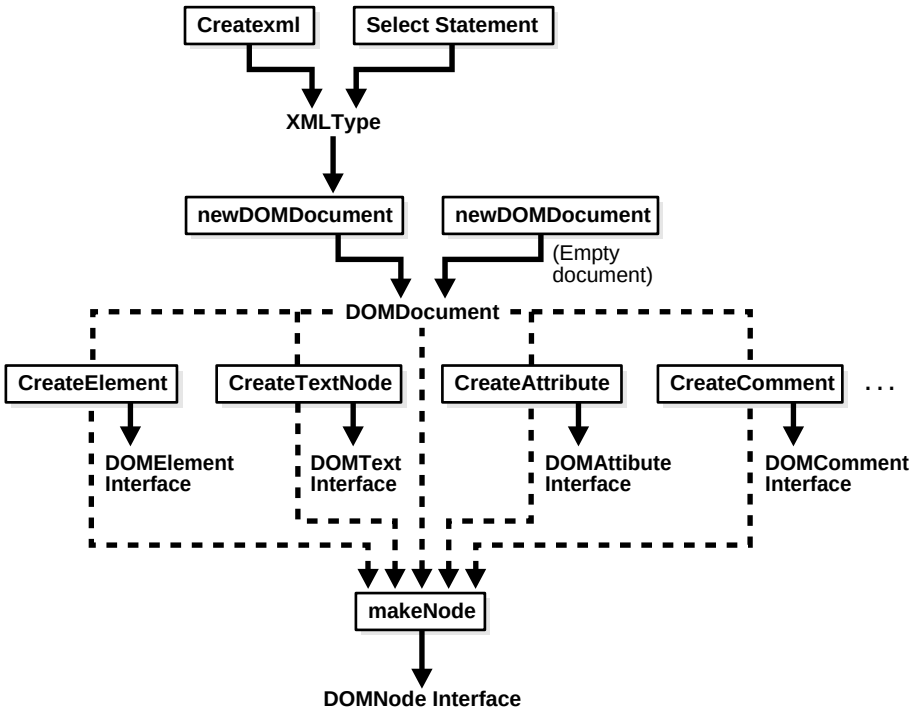
PL/SQL DOM API for XMLType (DBMS_XMLDOM): Calling Sequence

Figure 10–1 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 10–1 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 10–1 PL/SQL DOM API for XMLType: Calling Sequence



PL/SQL DOM API for XMLType Examples

Example 10–1 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 10–2 Creating a DOM Document Using sys.xmltype

This example creates a DOM document from an XMLType value:

```
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 10–3 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('Node name = '||dbms_xmldom.getNodeName(nelem));
       dbms_output.put_line('Node value = '||
                            dbms_xmldom.getNodeValue(nelem));
       dbms_output.put_line('Node type = '||dbms_xmldom.getNodeType(nelem));
```

```
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 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 10–3 lists the PL/SQL Parser API for `XMLType` (`DBMS_XMLPARSER`) methods.

Table 10–3 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

Table 10–3 (Cont.) DBMS_XMLPARSER Methods

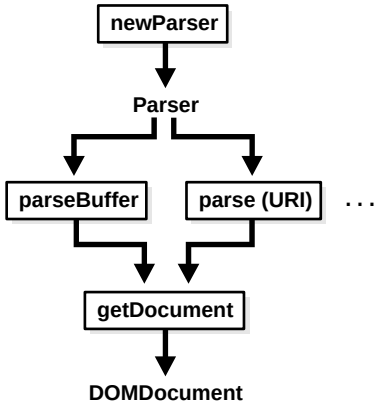
Method	Arguments, Return Values, and Results
`showWarnings`	Argument: `(p Parser, yes BOOLEAN)` Result: Turns warnings on or off
`setErrorLog`	Argument: `(p Parser, fileName VARCHAR2)` Result: Sets errors to be sent to the specified file
`setPreserveWhitespace`	Argument: (`p Parser`, `yes BOOLEAN`) Result: Sets white space preserve mode
`setValidationMode`	Argument: (`p Parser`, `yes BOOLEAN`) Result: Sets validation mode
`getValidationMode`	Argument: (`p Parser`) Result: Gets validation mode
`setDoctype`	[Not supported.] Argument: (`p Parser`, `dtd DOMDocumentType`) Result: Sets DTD
`getDoctype`	[Not supported.] 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 10–2 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 10–3 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 must call `getDocument` on the Parser to obtain a `DOMDocument` interface.

Figure 10–2 PL/SQL Parser API for XMLType: Calling Sequence



PL/SQL Parser API for XMLType Example

Example 10–4 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 style sheet. The transformation specified is achieved by associating patterns with templates defined in the XSLT style sheet. A template is instantiated to create part of the result tree.

Enabling Transformations and Conversions with XSLT

The Oracle XML DB PL/SQL DOM API for `XMLType` also supports 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

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 PL/SQL package `DBMS_XSLPROCESSOR` provides a convenient and efficient way of applying a single style sheet to multiple documents. The performance of this package will be better than `transform()` because the style sheet will be parsed only once.

Note: The XML Parser for PL/SQL in Oracle XDK parses an XML document (or a standalone DTD) so that the XML document can be processed by an application, typically running on the client. PL/SQL APIs for `XMLType` are used for applications that run on the server and are natively integrated in the database. Benefits include performance improvements and enhanced access and manipulation options.

See Also:

- Appendix D, "XSLT Primer"
- Chapter 8, "Transforming and Validating XMLType Data"

PL/SQL XSLT Processor for XMLType: Features

PL/SQL XSLT Processor for `XMLType` (`DBMS_XSLPROCESSOR`) is the Oracle XML DB implementation of the XSL processor. This follows the W3C XSLT final recommendation (REC-xslt-19991116). It includes the required action of an XSL processor in terms of how it must read XSLT style sheets and the transformations it must achieve.

The types and methods of 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 10–4 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 10–4 DBMS_XSLPROCESSOR Methods

Method	Argument or Return Values or Result
`newProcessor`	Returns: a new processor instance
`processXSL`	Argument: `(p Processor, ss Stylesheet, xmldoc DOMDocument)` Result: Transforms input XML document using given DOMDocument and style sheet
`processXSL`	Argument: `(p Processor, ss Stylesheet, xmldoc DOMDocumentFragment)` Result: Transforms input XML document using the given `DOMDocumentFragment` and style sheet
`showWarnings`	Argument: `(p Processor, yes BOOLEAN)` Result: Turn warnings on or off
`setErrorLog`	Argument: `(p Processor, Filename VARCHAR2)` Result: Sets errors to be sent to the specified file
`NewStylesheet`	Argument: `(Input VARCHAR2, Reference VARCHAR2)` Result: Sets errors to be sent to the specified file
`transformNode`	Argument: `(n DOMNode, ss Stylesheet)` Result: Transforms a node in a DOM tree using the given style sheet
`selectNodes`	Argument: `(n DOMNode, pattern VARCHAR2)` Result: Selects nodes from a DOM tree that match the given pattern
`selectSingleNodes`	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 given style sheet
`removeParam`	Argument: `(ss Stylesheet, name VARCHAR2)` Result: Removes a top-level style-sheet parameter
`ResetParams`	Argument: `(ss Stylesheet)` Result: Resets the top-level style-sheet parameters
`freeStylesheet`	Argument: `(ss Stylesheet)` Result: Frees the given `Stylesheet` object
`freeProcessor`	Argument: `(p Processor)` Result: Frees the given `Processor` object

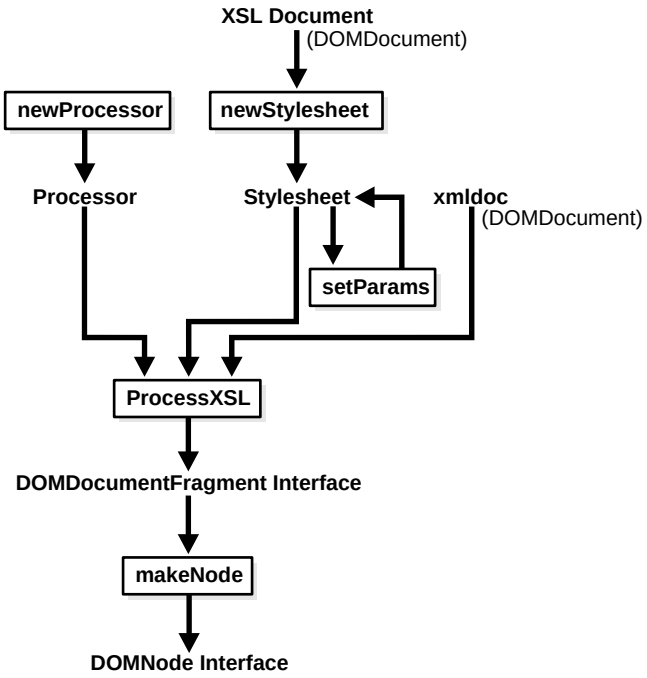
PL/SQL Parser API for XMLType (DBMS_XSLPROCESSOR): Calling Sequence

Figure 10–3 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 10–3 PL/SQL XSLT Processor for XMLType: Calling Sequence



PL/SQL XSLT Processor for XMLType Example

Example 10–5 Transforming an XML Document Using an XSL Style Sheet

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;
/
```

11

DBMS_XMLSTORE

This chapter introduces you to the PL/SQL package DBMS_XMLSTORE. This package is used to insert, update, and delete data from XML documents in object-relational tables.

This chapter contains these topics:

Overview of DBMS_XMLSTORE

The DBMS_XMLSTORE package enables DML operations to be performed on relational tables using XML. It takes a canonical XML mapping, similar to the one produced by DBMS_XMLGEN, converts it to object relational constructs, and inserts, updates or deletes the value from relational tables.

The functionality of the DBMS_XMLSTORE package is similar to that of the DBMS_XMLSAVE package which is part of the Oracle XML SQL Utility. There are, however, several key differences: DBMS_XMLSTORE is written in C and compiled into the kernel and hence provides higher performance.

- DBMS_XMLSTORE uses SAX to parse the input XML document and hence has higher scalability and lower memory requirements. DBMS_XMLSTORE allows input of XMLType in addition to CLOBs and VARCHAR.
- While DBMS_XMLSAVE is a wrapper around a Java class, DBMS_XMLSTORE is implemented in C inside the database. This should significantly improve performance.
- DBMS_XMLSTORE uses SAX parsing of the incoming XML documents, which provides much greater scalability than the DOM parsing used in DBMS_XMLSAVE.
- The insertXML(), updateXML(), and deleteXML() functions, which are also present in DBMS_XMLSAVE, have been enhanced in DBMS_XMLSTORE to take XMLTypes in addition to CLOBs and strings. This provides for better integration with Oracle XML DB functionality.

Using DBMS_XMLSTORE

To use DBMS_XMLSTORE follow these steps:

- Create a context handle by calling the DBMS_XMLSTORE.newContext() function and supplying it with the table name to use for the DML operations. For case sensitivity, double-quote the string which is passed to the function.

  By default, XML documents are expected to identify rows with the <ROW> tag. This is the same default used by DBMS_XMLGEN when generating XML. This may be overridden by calling the setRowTag function.

- For Inserts: You can set the list of columns to insert using the setUpdateColumn function for each column. This is highly recommended since it will improve performance. The default is to insert values for all the columns whose corresponding elements are present in the XML document.

- For Updates: You must specify one or more key columns using the setKeyColumn function. The key columns are used to specify which rows are to be updated, like the where clause in a SQL update statement. For example, if you set EMPLOYEE_ID as a key column, and the XML document contains "<EMPLOYEE_ID>2176</EMPLOYEE_ID>", then rows where EMPLOYEE_ID equals 2176 are updated. The list of update columns can also be specified and is recommended for performance. The default is to update all the columns whose corresponding elements are present in the XML document.

- For Deletes: Key columns may be set to specify which columns are used for the where clause. The default is for all columns present to be used. Specifying the columns is recommended for performance.

- Provide a document to one of insertXML, updateXML, or deleteXML.

- This last step may be repeated multiple times, with several XML documents.

- Close the context with the closeContext function.

Insert Processing with DBMS_XMLSTORE

To insert an XML document into a table or view, simply supply the table or the view name and then the document. DBMS_XMLSTORE parses the document and then creates an INSERT statement into which it binds all the values. By default, DBMS_XMLSTORE inserts values into all the columns represented by elements in the XML document. The following example shows you how the XML document generated from the Employees table, can be stored in the table with relative ease.

Example 11–1 Inserting data with specified columns

```
DECLARE
  insCtx DBMS_XMLStore.ctxType;
  rows NUMBER;
  xmldoc CLOB :=
    '<ROWSET>
       <ROW num="1">
         <EMPNO>7369</EMPNO>
         <SAL>1800</SAL>
         <HIREDATE>27-AUG-1996</HIREDATE>
       </ROW>
       <ROW>
         <EMPNO>2290</EMPNO>
         <SAL>2000</SAL>
         <HIREDATE>31-DEC-1992</HIREDATE>
```

```
      </ROW>
    </ROWSET>';
BEGIN
  insCtx := DBMS_XMLStore.newContext('scott.emp'); -- get saved context
  DBMS_XMLStore.clearUpdateColumnList(insCtx); -- clear the update settings
  -- set the columns to be updated as a list of values
  DBMS_XMLStore.setUpdateColumn(insCtx,'EMPNO');
  DBMS_XMLStore.setUpdateColumn(insCtx,'SAL');
  DBMS_XMLStore.setUpdatecolumn(insCtx,'HIREDATE');
  -- Now insert the doc.
  -- This will only insert into EMPNO, SAL and HIREDATE columns
  rows := DBMS_XMLStore.insertXML(insCtx, xmlDoc);
  -- Close the context
  DBMS_XMLStore.closeContext(insCtx);
END;
/
```

Update Processing with DBMS_XMLSTORE

Now that you know how to insert values into the table from XML documents, let us see how to update only certain values. If you get an XML document to update the salary of an employee and also the department that she works in:

```
<ROWSET>
  <ROW num="1">
    <EMPNO>7369</EMPNO>
    <SAL>1800</SAL>
    <DEPTNO>30</DEPTNO>
  </ROW>
  <ROW>
    <EMPNO>2290</EMPNO>
    <SAL>2000</SAL>
    <HIRE_DATE>31-DEC-1992</HIRE_DATE>
  <!-- additional rows ... -->
</ROWSET>
```

you can call the update processing to update the values. In the case of update, you need to supply the list of key column names. These form part of the WHERE clause in the UPDATE statement. In the `employees` table shown earlier, the employee number `EMPLOYEE_ID` column forms the key that you use for updates.

Example 11–2 Updating Data With Key Columns

Consider the following PL/SQL procedure:

```
CREATE OR REPLACE PROCEDURE testUpdate (xmlDoc IN CLOB) IS
  updCtx DBMS_XMLStore.ctxType;
  rows NUMBER;
BEGIN
   updCtx := DBMS_XMLStore.newContext('scott.emp');  -- get the context
   DBMS_XMLStore.clearUpdateColumnList(updCtx);      -- clear the update settings
   DBMS_XMLStore.setKeyColumn(updCtx,'EMPNO');       -- set EMPNO as key column
   rows := DBMS_XMLStore.updateXML(updCtx,xmlDoc);   -- update the table
   DBMS_XMLStore.closeContext(updCtx);               -- close the context
END;
/
```

In this example, when the procedure is executed with a CLOB value that contains the document described earlier, two UPDATE statements are generated. For the first `ROW`

element, you would generate an UPDATE statement to update the SALARY and JOB_ID fields as follows:

```
UPDATE scott.emp SET SAL = 1800 AND DEPTNO = 30 WHERE EMPNO = 7369;
```

and for the second ROW element:

```
UPDATE scott.emp SET SAL = 2000 AND HIREDATE = 12/31/1992 WHERE EMPNO = 2290;
```

Delete Processing with DBMS_XMLSTORE

For deletes, you can set the list of key columns. These columns are used in the WHERE clause of the DELETE statement. If the key column names are not supplied, then a new DELETE statement is created for each ROW element of the XML document where the list of columns in the WHERE clause of the DELETE matches those in the ROW element.

Example 11–3 Simple deleteXML Example

Consider the following PL/SQL example:

```
CREATE OR REPLACE PROCEDURE testDelete(xmlDoc IN CLOB) IS
  delCtx DBMS_XMLStore.ctxType;
  rows NUMBER;
BEGIN
  delCtx  := DBMS_XMLStore.newContext('scott.emp');
  DBMS_XMLStore.setKeyColumn(delCtx,'EMPNO');
  rows := DBMS_XMLStore.deleteXML(delCtx, xmlDoc);
  DBMS_XMLStore.closeContext(delCtx);
END;
/
```

If you use the same XML document as in the preceding update example, you end up with the following two DELETE statements:

```
DELETE FROM scott.emp WHERE EMPNO=7369;
DELETE FROM scott.emp WHERE EMPNO=2200;
```

The DELETE statements are formed based on the tag names present in each ROW element in the XML document.

12

Java API for XMLType

This chapter describes how to use `XMLType` in Java, including fetching `XMLType` data through JDBC.

This chapter contains these topics:

- Introducing Java DOM API for XMLType
- Java DOM API for XMLType
- Loading a Large XML Document into the Database With JDBC
- Java DOM API for XMLType Features
- Java DOM API for XMLType Classes

Introducing Java DOM API for XMLType

Oracle XML DB supports the Java Document Object Model (DOM) Application Program Interface (API) for XMLType. This is a generic API for client and server, for both XML schema-based and non- schema-based documents. It is implemented using the Java package `oracle.xdb.dom`. DOM is an in-memory tree-based object representation of XML documents that enables programmatic access to their elements and attributes. The DOM object and interface are part of a W3C recommendation. DOM views the parsed document as a tree of objects.

To access `XMLType` data using JDBC use the class `oracle.xdb.XMLType`.

For XML documents that do not conform to any XML schema, you can use the Java DOM API for `XMLType` because it can handle *any* valid XML document.

See Also:

- *Oracle XML API Reference*
- Appendix E, "Java APIs: Quick Reference"

Java DOM API for XMLType

Java DOM API for `XMLType` handles all kinds of valid XML documents irrespective of how they are stored in Oracle XML DB. It presents to the application a uniform view of the XML document 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 10, "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.

Java DOM API for `XMLType` can be used to construct an `XMLType` instance from data encoded in different character sets. It also provides a new `getBlobVal()` method to retrieve the XML contents in the requested character set.

See Also: Appendix E, "Java APIs: Quick Reference"

Accessing XML Documents in Repository

Oracle XML DB resource API for Java 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 Also: Chapter 22, "Java Access to Repository Data Using Resource API for Java"

Accessing XML Documents Stored in Oracle Database (Java)

Oracle XML DB provides the following way (part of the Java Resource APIs) for Java applications to access XML data stored in a database:

Using JDBC to Access XMLType Data

This is a SQL-based approach for Java applications for accessing any data in Oracle Database, including XML documents in Oracle XML DB. Use the `oracle.xdb.XMLType` class, `createXML()` method.

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.

Example 12–1 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 12–2 XMLType Java: Selecting XMLType Data

You can select the `XMLType` data in JDBC in one of two ways:

- Use the `getClobVal()`, `getStringVal()` or `getBlobVal(csid)` in SQL and get the result as an `oracle.sql.CLOB, java.lang.String` or

`oracle.sql.BLOB` 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 `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 12–3 XMLType Java: Directly Returning XMLType Data

This example shows the use of `getObject` to directly get the `XMLType` from the `ResultSet`. This code snippet is the easiest way to get the `XMLType` from the `ResultSet`.

```
import oracle.xdb.XMLType;
...
PreparedStatement stmt =  conn.prepareStatement(
          "select e.poDoc from po_xml_tab e");
ResultSet rset = stmt.executeQuery();
```

```
while(rset.next())
{
// get the XMLType
XMLType poxml = (XMLType)rset.getObject(1);

// get the XML as a string...
String poString = poxml.getStringVal();
 }
```

Example 12–4 XMLType Java: Returning XMLType Data and Registering the Output Parameter as XMLType

This example illustrates how to bind an `OUT` variable of `XMLType` to a SQL statement.

```
public void doCall (String[] args)
  throws Exception
  {

//  create or replace function getPurchaseOrder(reference varchar2)
//  return XMLType
//  as
//    xml XMLType;
//  begin
//    select value(p)
//    into xml
//    from PURCHASEORDER p
//    where extractValue(value(p),'/PurchaseOrder/Reference') = reference;
//    return xml;
//  end;

    String SQLTEXT = "{? = call
getPurchaseOrder('BLAKE-2002100912333601PDT')}";
    CallableStatement sqlStatement = null;
    XMLType xml = null;

    super.doSomething(args);
    createConnection();
    try
    {
      System.out.println("SQL := " + SQLTEXT);
      sqlStatement = getConnection().prepareCall(SQLTEXT);
      sqlStatement.registerOutParameter (1, OracleTypes.OPAQUE,"SYS.XMLTYPE");
      sqlStatement.execute();
      xml = (XMLType) sqlStatement.getObject(1);
      System.out.println(xml.getStringVal());
    }
    catch (SQLException SQLe)
    {
      if (sqlStatement != null)
      {
        sqlStatement.close();
        throw SQLe;
      }
    }
```

Using JDBC to Manipulate XML Documents Stored in a Database

You can also update, insert, and delete `XMLType` data using JDBC.

Note: `extract()`, `transform()`, and `existsNode()` methods only work with the Thick JDBC driver.

Not all `oracle.xdb.XMLType` functions are supported by the Thin JDBC driver. However, if you do not use `oracle.xdb.XMLType` classes and OCI driver, you could loose performance benefits associated with the intelligent handling of XML.

Example 12–5 XMLType Java: Updating, Inserting, or Deleting XMLType Data

You can insert an `XMLType` in Java in one of two ways:

- Bind a CLOB or a string to an `INSERT` or UPDATE or `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 12–6 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.exuecuteQuery();

// 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 12–7 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, or delete `XMLType`s 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 xdb.jar in CLASSPATH

*/

import java.sql.*;
import java.io.*;

import oracle.xml.parser.v2.*;
import org.xml.sax.*;
import org.w3c.dom.*;

import oracle.jdbc.driver.*;
import oracle.sql.*;

import oracle.xdb.XMLType;

public class tkxmtpje
{

  static String conStr = "jdbc:oracle:oci8:@";
  static String user = "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>
   <CUSTNO>2</CUSTNO>
   <CUSTNAME>John Nike</CUSTNAME>
   <ADDRESS>
    <STREET>323 College Drive</STREET>
    <CITY>Edison</CITY>
    <STATE>NJ</STATE>
    <ZIP>08820</ZIP>
```

```
   </ADDRESS>
   <PHONELIST>
    <VARCHAR2>609-555-1212</VARCHAR2>
    <VARCHAR2>201-555-1212</VARCHAR2>
   </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>20-APR-97</ORDERDATE>
  <SHIPDATE>20-MAY-97 12.00.00.000000 AM</SHIPDATE>
  <LINEITEMS>
   <LINEITEM_TYP LineItemNo="1">
    <ITEM StockNo="1004">
     <PRICE>6750</PRICE>
     <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>1</QUANTITY>
    <DISCOUNT>10</DISCOUNT>
   </LINEITEM_TYP>
   <LINEITEM_TYP LineItemNo="2">
    <ITEM StockNo="1011">
     <PRICE>4500.23</PRICE>
     <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>2</QUANTITY>
    <DISCOUNT>10</DISCOUNT>
   </LINEITEM_TYP>
  </LINEITEMS>
  <SHIPTOADDR>
   <STREET>55 Madison Ave</STREET>
   <CITY>Madison</CITY>
   <STATE>WI</STATE>
   <ZIP>53715</ZIP>
  </SHIPTOADDR>
</PurchaseOrder>
```

Example 12–8 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,
            null)).getStringVal());
          }
      catch (SQLException e) {
        System.out.println("Thin driver Expected exception: " + e);
      }

      System.out.println("Testing createXML(conn, doc) ...");
      try {
        XMLType xt1 = XMLType.createXML(conn, doc);
        System.out.println(xt1.getStringVal());
      }
     catch (SQLException e) {
        System.out.println("Got exception: " + e);
      }

    }
}
catch( Exception e )
{
  e.printStackTrace(System.out);
}
}
}
```

Loading a Large XML Document into the Database With JDBC

If a large XML document (greater than 4000 characters, typically) is inserted into an `XMLType` table or column using a `String` object in JDBC, this run-time error occurs:

```
"java.sql.SQLException: Data size bigger than max size for this type"
```

This error can be avoided by using a Java `CLOB` object to hold the large XML document. Example 12–9 demonstrates this technique, loading a large document into an `XMLType` column; the same approach can be used for `XMLType` tables. The `CLOB` object is created using class `oracle.sql.CLOB` on the client side. This class is the Oracle JDBC driver implementation of the standard JDBC interface `java.sql.Clob`.

Example 12–9 Loading a Large XML Document

In this example, method `insertXML()` inserts a large XML document into the `purchaseOrder` `XMLType` column of table `poTable`. It uses a `CLOB` object containing the XML document to do this. The `CLOB` object is bound to a JDBC prepared statement, which inserts the data into the `XMLType` column.

Prerequisites for running this example are as follows:

- Oracle Database, version 9.2.0.1 or later.
- `Classes12.zip` or `Classes12.jar`, available in `ORACLE_HOME\jdbc\lib`, should be included in the `CLASSPATH` environment variable.
- The target database table. Execute the following SQL before running the example:

  ```
  CREATE TABLE poTable (purchaseOrder XMLType);
  ```

Method insertXML()

The formal parameters of method `insertXML()` are as follows:

- `xmlData` – XML data to be inserted into the `XMLType` column
- `conn` – database connection object (Oracle Connection Object)

```
...
import oracle.sql.CLOB;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.PreparedStatement;
...

private void insertXML(String xmlData, Connection conn) {
  CLOB clob = null;
  String query;
    // Initialize statement Object
  PreparedStatement pstmt = null;
  try{
    query = "INSERT INTO potable (purchaseOrder) VALUES (XMLType(?)) ";
    // Get the statement Object
    pstmt = conn.prepareStatement(query);

    // xmlData is the string that contains the XML Data.
    // Get the CLOB object using the getCLOB method.
    clob = getCLOB(xmlData, conn);
    // Bind this CLOB with the prepared Statement
    pstmt.setObject(1, clob);
    // Execute the Prepared Statement
    if (pstmt.executeUpdate () == 1) {
    System.out.println ("Successfully inserted a Purchase Order");
    }
  } catch(SQLException sqlexp){
    sqlexp.printStackTrace();
  } catch(Exception exp){
    exp.printStackTrace();
  }
}
```

Method getCLOB()

Method `insertXML()` calls method `getCLOB()` to create and return the `CLOB` object that holds the XML data. The formal parameters of `getCLOB()` are as follows:

- `xmlData` – XML data to be inserted into the `XMLType` column
- `conn` – database connection object (Oracle Connection Object)

```
...
import oracle.sql.CLOB;
import java.sql.Connection;
import java.sql.SQLException;
import java.io.Writer;
...

private CLOB getCLOB(String xmlData, Connection conn) throws SQLException{
  CLOB tempClob = null;
  try{
    // If the temporary CLOB has not yet been created, create one
    tempClob = CLOB.createTemporary(conn, true, CLOB.DURATION_SESSION);
```

```
    // Open the temporary CLOB in readwrite mode, to enable writing
    tempClob.open(CLOB.MODE_READWRITE);
    // Get the output stream to write
    Writer tempClobWriter = tempClob.getCharacterOutputStream();
    // Write the data into the temporary CLOB
    tempClobWriter.write(xmlData);

    // Flush and close the stream
    tempClobWriter.flush();
    tempClobWriter.close();

    // Close the temporary CLOB
    tempClob.close();
  } catch(SQLException sqlexp){
    tempClob.freeTemporary();
    sqlexp.printStackTrace();
  } catch(Exception exp){
    tempClob.freeTemporary();
    exp.printStackTrace();
  }
  return tempClob;
}
```

See Also: *Oracle Database Application Developer's Guide - Large Objects*

Java DOM API for XMLType Features

When using Java DOM API to retrieve XML data from Oracle XML DB, you get the following results:

- If the connection is thin, then you get an `XMLDocument` instance
- If the Connection is thick or kprb, then you get an `XDBDocument` instance

Both of these are instances of the W3C Document Object Model (DOM) interface. From this document interface you can access the document elements and perform all the operations specified in the W3C DOM Recommendation. The DOM works on:

- Any type of XML document:
  - XML schema-based
  - Non-XML schema-based
- Any type of underlying storage used by the document:
  - Character Large Object (CLOB)
  - Binary Large Object (BLOB)
  - Object-relational.

The Java DOM API for `XMLType` supports deep 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. The XML document is stored in the appropriate format:

- As a BLOB for non-XML schema-based documents.
- In the format specified by the XML schema for XML schema-based documents.

Example 12–10 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_XMLTypetab set poDoc = ? ");

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

Element rootElem = poDOM.createElement("PO");
poDOM.insertBefore(poDOM, rootElem, null);

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

JDBC or SQLJ

An `XMLType` instance is represented in Java by `oracle.xdb.XMLType`. When an instance of `XMLType` is fetched using JDBC, it is automatically manifested as an object of the provided `XMLType` class. Similarly, objects of this class can be bound as values to Data Manipulation Language (DML) statements where an `XMLType` is expected. The same action is supported in SQLJ clients.

Note: The SQLJ precompiler has been desupported from Oracle Database 10*g* release 1 (10.1) and Oracle Application Server 10*g* release 1 (10.1). Oracle9*i* release 2 (9.2) and Oracle9*i*AS release 9.0.4 are the last Oracle products to offer SQLJ support. From this release, only the SQLJ precompiler that generates `.java` files from `.sqlj` files is desupported through both command-line and JDeveloper. However, both the client-side and server-side SQLJ runtimes are maintained to support JPublisher and existing precompiled SQLJ applications.

No new SQLJ application development is possible using Oracle products. Customer priority one (P1) bugs will continue to be fixed. Although the term SQLJ runtime has been renamed to JPublisher runtime, the term SQLJ Object Types is still used.

Java DOM API for XMLType Classes

Oracle XML DB supports the W3C DOM Level 2 Recommendation. In addition to the W3C Recommendation, Oracle XML DB DOM API also provides Oracle-specific extensions, to facilitate your application interfacing with Oracle XDK for Java. A list of the Oracle extensions is found at:

`http://otn.oracle.com/tech/xml/index.html`

`XDBDocument` 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 XDB`Document` class.

Table 12–1 lists the Java DOM API for `XMLType` classes and the W3C DOM interfaces they implement.

Table 12–1 Java DOM API for XMLType: Classes

Java DOM API for XMLType Class	W3C DOM Interface Recommendation Class
`oracle.xdb.dom.XDBDocument`	`org.w3c.dom.Document`
`oracle.xdb.dom.XDBCData`	`org.w3c.dom.CDataSection`
`oracle.xdb.dom.XDBComment`	`org.w3c.dom.Comment`
`oracle.xdb.dom.XDBProcInst`	`org.w3c.dom.ProcessingInstruction`
`oracle.xdb.dom.XDBText`	`org.w3c.dom.Text`
`oracle.xdb.dom.XDBEntity`	`org.w3c.dom.Entity`
`oracle.xdb.dom.DTD`	`org.w3c.dom.DocumentType`
`oracle.xdb.dom.XDBNotation`	`org.w3c.dom.Notation`
`oracle.xdb.dom.XDBNodeList`	`org.w3c.dom.NodeList`
`oracle.xdb.dom.XDBAttribute`	`org.w3c.dom.Attribute`
`oracle.xdb.dom.XDBDOMImplementation`	`org.w3c.dom.DOMImplementation`
`oracle.xdb.dom.XDBElement`	`org.w3c.dom.Element`
`oracle.xdb.dom.XDBNamedNodeMap`	`org.w3c.dom.NamedNodeMap`
`oracle.xdb.dom.XDBNode`	`org.w3c.dom.Node`

Java Methods Not Supported

The following are methods documented in release 2 (9.2.0.1) but not currently supported:

- `XDBDocument.getElementByID`
- `XDBDocument.importNode`
- `XDBNode.normalize`
- `XDBNode.isSupported`
- `XDBDomImplementation.hasFeature`

Java DOM API for XMLType: Calling Sequence

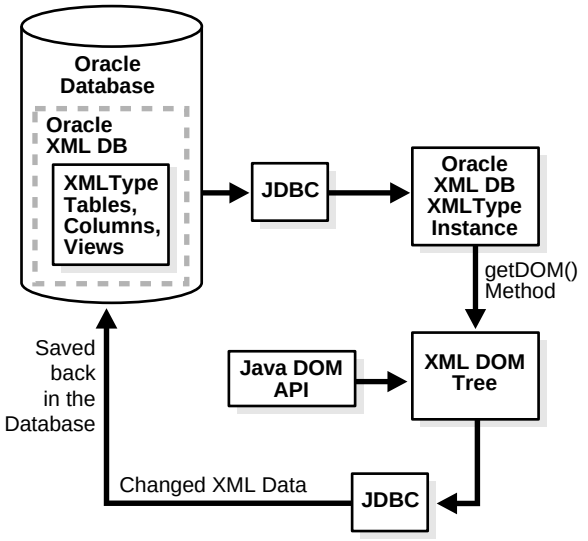
The following Java DOM API for `XMLType` calling sequence description assumes that your XML data is pre-registered with an XML schema and that it is stored in an `XMLType` datatype column. To use the Java DOM API for `XMLType`, follow these steps:

1. Retrieve the XML data from the `XMLType` table or `XMLType` column in the table. When you fetch XML data, Oracle creates an instance of an `XMLType`. You can then use the `getDom()` method to retrieve a Document instance. You can then manipulate elements in the DOM tree using Java DOM API for `XMLType`.
2. Use the Java DOM API for `XMLType` to perform operations and manipulations on elements of the DOM tree.
3. The `XMLType` holds the modified data, but the data is sent back using a JDBC update.

The `XMLType` and `XDBDocument` objects should be closed using the `close()` method in the respective classes. This releases any underlying memory that is held.

Figure 12–1 illustrates the Java DOM API for `XMLType` calling sequence.

Figure 12–1 Java DOM API for XMLType: Calling Sequence



13

Using C API for XML With Oracle XML DB

This chapter provides a guideline for using the C API for XML with Oracle XML DB.

This chapter contains these topics:

- Introducing the C API for XML (XDK and Oracle XML DB)
- Using OCI and the C API for XML with Oracle XML DB
- XML Context
- How to Use Oracle XML DB Functions
- OCI Usage
- Common XMLType Operations in C

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide* "XML Parser for C"

Introducing the C API for XML (XDK and Oracle XML DB)

The C API for XML is used for both XDK (XML Developer's Kit) and Oracle XML DB. It is a C-based DOM API for XML. It can be used for XML either inside or outside the database. Here DOM refers to compliance with the World Wide Web Consortium (W3C) DOM 2.0 Recommendation. This API also includes non-standard performance improving extensions used as follows:

- In XDK, for traditional XML storage
- In Oracle XML DB, for XML stored as an `XMLType` column in a table

> **Note:** Use this new C API for XML for any new XDK and Oracle XML DB applications. C DOM functions from prior releases are supported only for backward compatibility, but will not be enhanced.

The C API for XML is implemented on `XMLType` in Oracle XML DB. In the W3C DOM Recommendation, the term document is used in a broad sense (URI, file system, memory buffer, standard input and output).

The C API for XML is a combined programming interface that includes all functionality needed by both XDK and Oracle XML DB applications. It provides for XSLT and XML Schema implementations. Although DOM 2.0 Recommendation was followed closely, some naming changes were required for mapping from the

objected-oriented DOM 2.0 Recommendation to the flat C namespace. For example, the method `getName()` was renamed to `getAttrName()`.

C API for XML supersedes existing APIs. In particular, the `oraxml` interface (top-level, DOM, SAX, and XSLT) and `oraxsd.h` (Schema) interfaces will be deprecated in a future release.

Using OCI and the C API for XML with Oracle XML DB

Oracle XML DB provides support for storing and manipulating XML instances using the XMLType datatype. These XML instances can be accessed and manipulated using the Oracle Call Interface (OCI) in conjunction with the C DOM API for XML.

The main flow for an application program would involve initializing the usual OCI handles such as server handle and statement handle followed by initialization of an XML Context. The user program can then either operate on XML instances in the back end or create new instances on the client side. The initialized *XML context* can be used with all the C DOM functions.

XML data stored in Oracle XML DB can be accessed on the client side using the C DOM structure `xmldocnode`. This structure can be used for binding, defining and operating on XML values in OCI statements.

XML Context

An *XML context* is a required parameter to all the C DOM API functions. This context encapsulates information pertaining to data encoding, error message language and such. This contents of this opaque context are different for XDK applications and Oracle XML DB.

For Oracle XML DB, there are two OCI functions provided to initialize an XML context. The function `OCIXmlDbInitXmlCtx()` is used to initialize the context while `OCIXmlDbFreeXmlCtx()` tears it down.

OCIXmlDbFreeXmlCtx() Syntax

Here is the `OCIXmlDbFreeXmlCtx()` syntax:

```
void OCIXmlDbFreeXmlCtx ( xmlctx *xctx);
```

where parameter, `xctx (IN)` is the XML context to terminate.

OCIXmlDbInitXmlCtx() Syntax

Here is the `OCIXmlDbInitXmlCtx()` syntax:

```
xmlctx *OCIXmlDbInitXMlCtx (  OCIEnv               *envhp,
                              OCISvcHp             *svchp,
                              OCIError             *errhp,
                              ocixmldbparam        *params,
                              ub4                  num_params );
```

where Table 13–1 lists the parameters.

Table 13–1 OCIXmlDbInitXMlCtx Parameters

Parameter	Description
`envhp (IN)`	The OCI environment handle.

Table 13–1 (Cont.) OCIXmlDbInitXMlCtx Parameters

Parameter	Description
svchp (IN)	The OCI service handle.
errhp (IN)	The OCI error handle
params (IN)	Optional values in this array as follows: OCI duration. Default value is OCI_DURATION_SESSION. An error handler which is a user-registered callback of prototype: void (*err_handler) (sword errcode, (CONST OraText *) errmsg);void (*err_handler) (sword errcode, (CONST OraText *) errmsg);
num_params (IN)	Number of parameters to be read from params.

How to Use Oracle XML DB Functions

In Oracle XML DB, to initialize and terminate the XML context, use the functions OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx(), respectively, as shown in the following example. These examples are found in header file ocixmldb.h and used with the C API for XML:

Example 13–1 Using OCI and C API for XML with Oracle XML DB

```
#ifndef XML_ORACLE
#include <xml.h>
#endif

#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif
...

typedef enum {
   XCTXINIT_OCIDUR = 1,
   XCTXINIT_ERRHDL = 2
} ocixmldbpname;

typdef struct ocixmldbparam {
  ociXmlDbPName name_ocixml dbpname;
  void *value_ocixmldbparam;
} ocixmldbparam;
```

The following code shows how to perform operations with the C API for XML:

```
{
OCIStmt *stmthp = (OCIStmt *)0;
xmlctx *xctx = (xmlctx *)0;
ocixmldbparam params[NUM_PARAMS];
OCIType *xmltdo = (OCIType *)0;
OCIDuration dur = OCI_DURATION_SESSION;
text *sel_xml_stmt = (text*)"SELECT xml_col FROM tkpgxucm_tab";
OraText *xpathexpr = (OraText *)"/name";
sword status = 0;
```

```
/* Allocate statement handle for SQL executions */
if (status=OCIHandleAlloc((dvoid *)ctxptr->envhp, (dvoid **)&stmthp,
                          (ub4)OCI_HTYPE_STMT, (CONST size_t)0, (dvoid **)0))
{

   return OCI_ERROR;
}

/* Get an XML context */
params[0].name_xmlctx_param = XCTXINIT_OCIDUR;
params[0].value_xmlctx_param = &dur;

/* Initialize an XML context */
xctx = OCIXmlDbInitXMlCtx (ctxptr->envhp, ctxptr->svchp, ctxptr->errhp,
       params, 1);

/* Do unified C API operations next */
...

/* Free the allocations associated with the context */
OCIXmlDbFreeXMlCtx(xctx);
}
```

OCI Usage

OCI applications operating on XML typically operate on XML data stored in the server and also on XML data created on the client. This section explains these two access methods in more detail.

Accessing XMLType Data From the Back End

XML data on the server can be operated on the client using regular OCI statement calls. Similar to other object instances, users can bind and define XMLType values using `xmldocnode`. OCI statements can be used to select XML data from the server and this can be used in the C DOM functions directly. Similarly, the values can be bound back to SQL statements directly.

Creating XMLType Instances on the Client

New `XMLType` instances on the client can be constructed using the `XmlLoadDom()` calls. You first have to initialize the `xmlctx` as in the preceding example. The XML data itself can be constructed from a user buffer, local file, or URI. The return value from these is a (`xmldocnode *`) that can be used in the rest of the common C API. Finally, the (`xmldocnode *`) can be cast to a (`void*`) and directly provided as the bind value if required.

Empty `XMLType` instances can be constructed using the `XMLCreateDocument()` call. This would be equivalent to an `OCIObjectNew()` for other types. You can operate on the (`xmldocnode*`) returned by the preceding call and finally cast it to a (`void*`) if it needs to be provided as a bind value.

Common XMLType Operations in C

Table 13–2 provides the `XMLType` functional equivalent of common XML operations.

Table 13–2 Common XMLType Operations in C

Description	C API XMLType Function
Create empty `XMLType` instance	`XmlCreateDocument()`
Create from a source buffer	`XmlLoadDom()`
Extract an XPath expression	`XmlXPathEvalexpr()` and family
Transform using an XSLT style sheet	`XmlXslProcess()` and family
Check if an XPath exists	`XmlXPathEvalexpr()` and family
Is document schema-based?	`XmlDomIsSchemaBased()`
Get schema information	`XmlDomGetSchema()`
Get document namespace	`XmlDomGetNodeURI()`
Validate using schema	`XmlSchemaValidate()`
Obtain DOM from XMLType	`Cast (void *) to (xmldocnode *)`
Obtain XMLType from DOM	`Cast (xmldocnode *) to (void *)`

See Also: *Oracle XML Developer's Kit Programmer's Guide* "XML Parser for C"

Example 13–2 Constructing an XML Schema-Based Document Using DOM API and Saving it to Oracle Database

```
static oratext tlpxml_test_sch[] =
"<TOP xmlns='example1.xsd'\n\
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' \n\
xsi:schemaLocation='example1.xsd example1.xsd'/>";
void example1()
{
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIDuration dur;
OCIType *xmltdo = (OCIType *) 0;
xmldocnode *doc;
ocixmldbparam params[1];
xmlnode *quux, *foo, *foo_data;
xmlerr err;
sword status = 0;

/* Initialize envhp, svchp, errhp, dur, stmthp */
/* ........ */
/* Get an xml context */
params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
params[0].value_ocixmldbparam = &dur;
xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, 1);
/* Start processing */
printf("Supports XML 1.0: %s\n",
XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
"YES" : "NO");
/* Parsing a schema-based document */
if (!(doc = XmlLoadDom(xctx, &err, "buffer", tlpxml_test_sch,
"buffer_length", sizeof(tlpxml_test_sch)-1,
"validate", TRUE, NULL)))
{
```

```
printf("Parse failed, code %d\n");
return;
}
/* Create some elements and add them to the document */
top = XmlDomGetDocElem(xctx, doc);
quux = (xmlnode *) XmlDomCreateElem(xctx ,doc, (oratext *) "QUUX");
foo = (xmlnode *) XmlDomCreateElem(xctx, doc, (oratext *) "FOO");
foo_data = (xmlnode *) XmlDomCreateText(xctx, doc, (oratext *) "foo's
data");
foo_data = XmlDomAppendChild(xctx, (xmlnode *) foo, (xmlnode *) foo_data);
foo = XmlDomAppendChild(xctx, quux, foo);
quux = XmlDomAppendChild(xctx, top, quux);
XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);
XmlSaveDom(xctx, &err, doc, "stdio", stdout, NULL);
/* Insert the document to my_table */
ins_stmt = "insert into my_table values (:1)";
status = OCITypeByName(envhp, errhp, svchp, (const text *) "SYS",
(ub4) strlen((char *)"SYS"), (const text *) "XMLTYPE",
(ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
(ub4) 0, dur, OCI_TYPEGET_HEADER,
(OCIType **) &xmltdo)) ;
if (status == OCI_SUCCESS)
{
exec_bind_xml(svchp, errhp, stmthp, (void *)doc, xmltdo, ins_stmt));
}

/* free xml ctx */
OCIXmlDbFreeXmlCtx(xctx);
}
/*-------------------------------------------------------*/
/* execute a sql statement which binds xml data */
/*-------------------------------------------------------*/
sword exec_bind_xml(svchp, errhp, stmthp, xml, xmltdo, sqlstmt)
OCISvcCtx *svchp;
OCIError *errhp;
OCIStmt *stmthp;
void *xml;
OCIType *xmltdo;
OraText *sqlstmt;
{

OCIBind *bndhp1 = (OCIBind *) 0;
sword status = 0;
OCIInd ind = OCI_IND_NOTNULL;
OCIInd *indp = &ind;
if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
(ub4)strlen((char *)sqlstmt),
(ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)) {
return OCI_ERROR;
}
if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *) 0,
(sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
(ub2 *)0, (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)) {
return OCI_ERROR;
}
if(status = OCIBindObject(bndhp1, errhp, (CONST OCIType *) xmltdo, (dvoid **)
&xml,
(ub4 *) 0, (dvoid **) &indp, (ub4 *) 0)) {
return OCI_ERROR;
}
```

```
if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
(CONST OCISnapshot*) 0, (OCISnapshot*) 0, (ub4) OCI_DEFAULT)) {
return OCI_ERROR;
}
return OCI_SUCCESS;
}
```

Example 13–3 Retrieving a Document From Oracle Database and Modifying it Using the DOM API

```
sword example2()
{
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIDuration dur;
OCIType *xmltdo_p=(OCIType *) 0;;
xmldocnode *doc;
xmlnodelist *item_list; ub4 ilist_l;
ocixmldbparam params[1];
text *sel_xml_stmt = (text *)"SELECT xml_col FROM my_table";
ub4 xmlsize = 0;
sword status = 0;
OCIDefine *defnp = (OCIDefine *) 0;
/* Initialize envhp, svchp, errhp, dur, stmthp */
/* ........ */
/* Get an xml context */
params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
params[0].value_ocixmldbparam = &dur;
xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, 1);
/* Start processing */
if(status = OCITypeByName(envhp, errhp, svchp, (const text *) "SYS",
(ub4) strlen((char *)"SYS"), (const text *) "XMLTYPE",
(ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
(ub4) 0, dur, OCI_TYPEGET_HEADER,
(OCIType **) &xmltdo_p)) {
return OCI_ERROR;
}
if(!(xmltdo_p)) {
printf("NULL tdo returned\n");
return OCI_ERROR;
}
if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)selstmt,
(ub4)strlen((char *)selstmt),
(ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)) {
return OCI_ERROR;
&&&&&&&&}

if(status = OCIDefineByPos(stmthp, &defnp, errhp, (ub4) 1, (dvoid *) 0,
(sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
(ub2 *)0, (ub4) OCI_DEFAULT)) {
return OCI_ERROR;
}
if(status = OCIDefineObject(defnp, errhp, (OCIType *) xmltdo_p,
(dvoid **) &doc,
&xmlsize, (dvoid **) 0, (ub4 *) 0)) {
return OCI_ERROR;
}
if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
```

```
(CONST OCISnapshot*) 0, (OCISnapshot*) 0, (ub4) OCI_DEFAULT)) {
return OCI_ERROR;
}
/* We have the doc. Now we can operate on it */
printf("Getting Item list...\n");
item_list = XmlDomGetElemsByTag(xctx, (xmlelemnode *) elem, (oratext *)
"Item");
ilist_l = XmlDomGetNodeListLength(xctx, item_list);
printf(" Item list length = %d \n", ilist_l);
for (i = 0; i < ilist_l; i++)
{
elem = XmlDomGetNodeListItem(xctx, item_list, i);
printf("Elem Name:%s\n", XmlDomGetNodeName(xctx, fragelem));
XmlDomRemoveChild(xctx, fragelem);
}
XmlSaveDom(xctx, &err, doc, "stdio", stdout, NULL);
/* free xml ctx */
OCIXmlDbFreeXmlCtx(xctx);
return OCI_SUCCESS;
}
```

14

Using ODP.NET With Oracle XML DB

This chapter describes how to use Oracle Data Provider for .NET (ODP.NET) with Oracle XML DB.

This chapter contains these topics:

Overview of Oracle Data Provider for .NET (ODP.NET)

Oracle Data Provider for .NET (ODP.NET) is an implementation of a data provider for Oracle Database. ODP.NET uses Oracle native APIs to offer fast and reliable access to Oracle data and features from any .NET application. ODP.NET also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library. The ODPNT supports the following LOBs as native datatypes with .NET: BLOBs, CLOBs, NCLOBs, and BFILEs.

ODP.NET XML Support

ODP.NET supports XML natively in the database, through Oracle XML DB. ODP.NET XML support includes the following features:

- Stores XML data natively in Oracle Database as `XMLType`.
- Accesses relational and object-relational data as XML data from Oracle Database to a Microsoft .NET environment, and processes the XML using Microsoft .NET framework.
- Saves changes to the database server using XML data.

For the .NET application developer, these features include the following:

- Enhancements to the `OracleCommand`, `OracleConnection`, and `OracleDataReader` classes. Provides the following XML-specific classes:
  - `OracleXmlType`
  - `OracleXmlStream`
  - `OracleXmlQueryProperties`
  - `OracleXmlSaveProperties`

ODP.NET Sample Code

This example retrieves `XMLType` data from the database to .NET and outputs the results:

```
//Create OracleCommand and query XMLType
OracleCommand xmlCmd = new OracleCommand();
poCmd.CommandText = "SELECT po FROM po_tab";
poCmd.Connection = conn;
// Execute OracleCommand and output XML results to an OracleDataReader
OracleDataReader poReader = poCmd.ExecuteReader();
// ODP.NET native XML data type object from XML DB
OracleXmlType poXml;
string str = ""; //read XML results
while (poReader.Read())
{
  // Return OracleXmlType object of the specified XmlType column
  poXml = poReader.GetOracleXmlType(0);
  // Concatenate output for all the records
  str = str + poXml.Value;
} //Output XML results to the screen
Console.WriteLine(str);
```

See Also: *Oracle Data Provider for .NET Developer's Guide* for complete information about Oracle .NET support for 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:

15

Generating XML Data from the Database

This chapter describes Oracle XML DB options for generating XML from the database. It explains the SQL/XML standard functions and Oracle Database-provided functions and packages for generating XML data from relational content.

This chapter contains these topics:

Oracle XML DB Options for Generating XML Data From Oracle Database

Oracle Database supports native XML generation. Oracle provides you with several options for generating or regenerating XML data when stored in:

- Oracle Database, in general
- Oracle Database in `XMLTypes` columns and tables

The following discussion illustrates the Oracle XML DB options you can use to generate XML from Oracle Database.

Generating XML Using SQL/XML Functions

The following SQL/XML functions are supported in Oracle XML DB:

- XMLElement() Function on page 15-3
- XMLForest() Function on page 15-8
- XMLConcat() Function on page 15-13
- XMLAgg() Function on page 15-14

Generating XML Using Oracle Database Extensions to SQL/XML

The following are Oracle Database extension functions to SQL/XML:

- XMLColAttVal() Function on page 15-18

Generating XML Using DBMS_XMLGEN

Oracle XML DB supports `DBMS_XMLGEN`, a PL/SQL supplied package. `DBMS_XMLGEN` generates XML from SQL queries.

See Also:

- "Generating XML from Oracle Database Using DBMS_XMLGEN" on page 15-19
- *PL/SQL Packages and Types Reference*, description of DBMS_XMLGEN

Generating XML Using SQL Functions

Oracle XML DB also supports the following Oracle Database-provided SQL functions that generate XML from SQL queries:

- XMLSequence() Function on page 15-9. Note that only the cursor version of this function generates XML. This function is also classified as a SQL/XML function.
- SYS_XMLGEN() Function on page 15-41. This operates on rows, generating XML documents.
- SYS_XMLAGG() Function on page 15-51. This operates on groups of rows, aggregating several XML documents into one.

Generating XML with XSQL Pages Publishing Framework

Generating XML Using XSQL Pages Publishing Framework on page 15-52 can also be used to generate XML from Oracle 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 15-54
- Chapter 3, "Using Oracle XML DB"
- Chapter 8, "Transforming and Validating XMLType Data"
- Chapter 10, "PL/SQL API for XMLType"
- Chapter 12, "Java API for XMLType"
- *Oracle XML API Reference*

Generating XML from the Database Using SQL/XML Functions

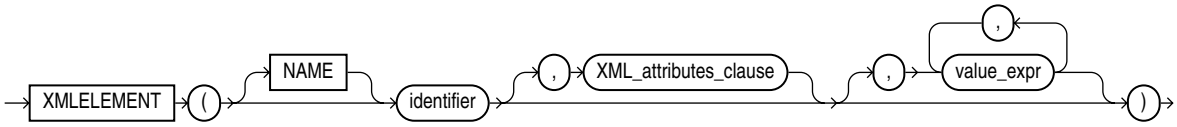
`XMLElement()`, `XMLForest()`, `XMLConcat()`, and `XMLAgg()` belong to the SQL/XML standard, an emerging SQL standard for XML. Because these are emerging standards the syntax and semantics of these functions are subject to change in the future in order to conform to the standard. The SQL/XML standard is being developed under the auspices of INCITS Technical Committee H2, the USA committee responsible for SQL and SQL/MM. ("INCITS" stands for "International Committee for Information Technology Standards") INCITS is an Accredited Standards Development Organization operating under the policies and procedures of ANSI, the American National Standards Institute. SQL/XML is being developed as a new part (Part 14) of the SQL standard and is aligned with SQL:2003.

All of the generation functions convert scalars and user-defined types (UDTs) to their canonical XML format. In canonical mapping the user-defined type 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 content and returns an instance of type `XMLType`. See Figure 15–1. The XML_attributes_clause is described in the following section.

Figure 15–1 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 a SQL identifier, characters that are disallowed in an XML element name are escaped. With p*artial escaping* the SQL identifiers other than the `":"` sign that are not representable in XML, are preceded by

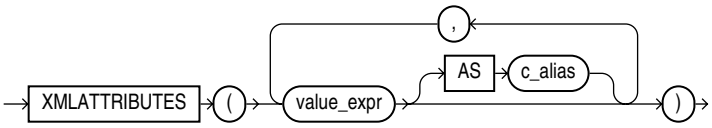
an escape character using the # sign followed by the unicode representation of that character in hexadecimal format. This can be used to specify namespace prefixes for the elements being generated.

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 15–2.

Figure 15–2 XMLAttributes 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 15–1 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:
-- EMPLOYEE_ID    result
-- --------------------
--        1001    <Emp>John Smith</Emp>
--        1206    <Emp>Mary Martin</Emp>

2 rows selected
```

`XMLElement()` can also be nested to produce XML data with a nested structure.

Example 15–2 XMLElement(): Generating Nested XML

To produce an `Emp` element for each employee, with elements that provide the employee's name and start date, do the following:

```
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>24-MAY-00</hiredate>
</Emp>
<Emp>
  <name>Mary Martin</name>
  <hiredate>01-FEB-96</hiredate>
</Emp>

2 rows selected
```

If NLS_DATE_FORMAT is set to YYYY-MM-DD, then the date is in XML schema date format. The same query then produces this result:

```
result
-----------------
<Emp>
  <name>John Smith</name>
  <hiredate>2000-05-24</hiredate>
</Emp>
<Emp>
  <name>Mary Martin</name>
  <hiredate>1996-02-01</hiredate>
</Emp>

2 rows selected
```

Note: Attributes, if they are specified, appear in the second argument of XMLElement() as:

"XMLATTRIBUTES (attribute, ...)".

Example 15–3 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.employee_id as "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 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 employee_id = 1001;
```

This query could produce the following XML result:

```
result
--------------
<Emp:Exempt FNAME="John" name:last="Smith" name_x003A_middle="Quincy"/>
</Emp:Exempt>

1 row selected.
```

> **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 15–4 XMLElement(): Using Namespaces to Create a Schema-Based XML Document

The following example illustrates the use of namespaces to create an XML schema-based document. Assuming that an XML schema "`http://www.oracle.com/Employee.xsd`" exists and has no target namespace, then the following query creates an `XMLType` instance conforming to that schema:

```
SET LONG 2000
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 = 10;
```

This creates an XML document that conforms to the `Employee.xsd` XMLSchema, result:

```
result
--------------
<Employee xmlns:xsi="http://www.w3.org/2001/XMLSchema"
          xsi:nonamespaceSchemaLocation="http://www.oracle.com/Employee.xsd">
   <EMPNO>7782</EMPNO>
   <ENAME>CLARK</ENAME>
   <SAL>2450</SAL>
</Employee>

<Employee xmlns:xsi="http://www.w3.org/2001/XMLSchema"
          xsi:nonamespaceSchemaLocation="http://www.oracle.com/Employee.xsd">
   <EMPNO>7839</EMPNO>
   <ENAME>KING</ENAME>
```

```
   <SAL>5000</SAL>
</Employee>

<Employee xmlns:xsi="http://www.w3.org/2001/XMLSchema"
          xsi:nonamespaceSchemaLocation="http://www.oracle.com/Employee.xsd">
   <EMPNO>7934</EMPNO>
   <ENAME>MILLER</ENAME>
   <SAL>1300</SAL>
</Employee>

3 rows selected.
```

Example 15–5 XMLElement(): Generating an Element from a User-Defined Type

Using the same example as given in the following `XMLSequence()` section for generating one XML document from another, you can generate a hierarchical XML for the employee, department example as follows:

```
CREATE OR REPLACE TYPE emp_t AS OBJECT ("@EMPNO" NUMBER(4),
                                        ENAME VARCHAR2(10));
/

CREATE OR REPLACE TYPE emplist_t AS TABLE OF emp_t;
/

CREATE OR REPLACE TYPE dept_t AS OBJECT ("@DEPTNO" NUMBER(2),
                                        DNAME VARCHAR2(14),
                                        EMP_LIST EMPLIST_T);
/

SELECT XMLElement("Department",
                  dept_t(deptno, dname,
                         CAST(MULTISET(select empno, ename from scott.emp e
                                       where e.deptno = d.deptno)
                              AS emplist_t)))
AS deptxml
FROM scott.dept d
WHERE d.deptno = 10;
```

This produces an XML document which contains the `Department` element and the canonical mapping of the `dept_t` type.

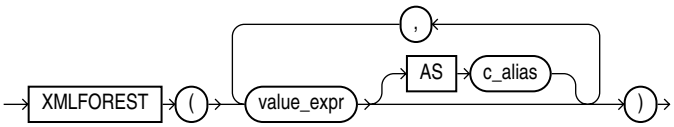
```
DEPTXML
-------------
<Department>
  <DEPT_T DEPTNO="10">
    <DNAME>ACCOUNTING</DNAME>
  <EMPLIST>
    <EMP_T EMPNO="7782">
      <ENAME>CLARK</ENAME>
    </EMP_T>
    <EMP_T EMPNO="7839">
      <ENAME>KING</ENAME>
    </EMP_T>
    <EMP_T EMPNO="7934">
      <ENAME>MILLER</ENAME>
    </EMP_T>
  </EMPLIST>
  </DEPT_T>
</Department>
```

```
1 row selected.
```

XMLForest() 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 15–3 describes the `XMLForest()` syntax.

Figure 15–3 XMLForest() Syntax



The list of value expressions are converted to XML format. For a given expression, if the AS clause is omitted, then 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. For other types, the AS clause can 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 15–6 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.department AS "department"))
AS "result"
FROM employees e;
```

This query might produce the following XML result:

```
result
--------------
<Emp name="John Smith">
  <HIRE>24-MAY-00</HIRE>
  <department>Accounting</department>
</Emp>
<Emp name="Mary Martin">
  <HIRE>FEB-01-96</HIRE>
  <department>Shipping</department>
</Emp>

2 rows selected.
```

If `NLS_DATE_FORMAT` is set to `YYYY-MM-DD`, then the date is in XML schema date format. The same query then produces this result:

```
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>

2 rows selected.
```

Example 15–7 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 on generating complex XML, 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 scott.emp e
                         where e.deptno = d.deptno)
               AS emplist_t))
         AS "Department")
AS deptxml
FROM scott.dept d
WHERE deptno=10;
```

This produces an XML document that contains the `Department` element and the canonical mapping of the `dept_t` type.

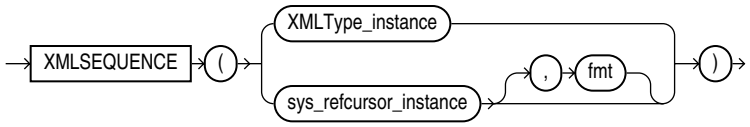
> **Note:** Unlike in the `XMLElement()` case, the `DEPT_T` element is missing.

```
DEPTXML
-------------
<Department>
  <DEPT_T DEPTNO="10">
    <DNAME>ACCOUNTING</DNAME>
    <EMP_LIST>
      <EMP_T EMPNO="7782">
        <ENAME>CLARK</ENAME>
      </EMP_T>
      <EMP_T EMPNO="7839">
        <ENAME>KING</ENAME>
      </EMP_T>
      <EMP_T EMPNO="7934">
        <ENAME>MILLER</ENAME>
      </EMP_T>
    </EMP_LIST>
  </DEPT_T>
</Department>

1 row selected.
```

XMLSequence() Function

`XMLSequence()` function returns a sequence of `XMLType`. The function returns an `XMLSequenceType` which is a VARRAY of `XMLType` instances. Because this function returns a collection, it can be used in the FROM clause of SQL queries. See Figure 15–4.

Figure 15–4 XMLSequence() Syntax



`XMLSequence()` only returns top-level element nodes. That is, it will not shred attributes or text nodes. For example:

```
SELECT value(T).getstringval() Attribute_Value
FROM TABLE(XMLSEQUENCE(extract(XMLType('<A><B>V1</B><B>V2</B><B>V3</B></A>'),
                                       '/A/B'))) T

ATTRIBUTE_VALUE
----------------------
<B>V1</B>
<B>V2</B>
<B>V3</B>

3 rows selected.
```

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 15–8 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>
    <EMPNAME>Jack</EMPNAME>
    <SALARY>40000</SALARY>
 </EMP>
</EMPLOYEES>
```

To create a new XML document containing only employees who make $50,000 or more a 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>

2 rows selected.
```

Notice how `extract()` was used to extract out all the employees:

1. `Extract()` 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 15–9 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() AS "xmltype"
FROM TABLE(XMLSequence(Cursor(SELECT * FROM scott.emp))) e;
```

might return the following XML:

```
xmltype
--------------------------------
  <ROW>
    <EMPNO>7369</EMPNO>
    <ENAME>SMITH</ENAME>
    <JOB>CLERK</JOB>
    <MGR>7902</MGR>
    <HIREDATE>17-DEC-80</HIREDATE>
    <SAL>800</SAL>
    <DEPTNO>20</DEPTNO>
  </ROW>

  <ROW>
    <EMPNO>7499</EMPNO>
    <ENAME>ALLEN</ENAME>
    <JOB>SALESMAN</JOB>
    <MGR>7698</MGR>
    <HIREDATE>20-FEB-81</HIREDATE>
```

```
    <SAL>1600</SAL>
    <DEPTNO>30</DEPTNO>
  </ROW>
...
14 rows selected.
```

The row tag used for each row can be changed using the `XMLFormat` object.

Example 15–10 XMLSequence(): Unnesting Collections in XML Documents into SQL Rows

`XMLSequence()` being a `TABLE` function, can be used to unnest the elements inside an XML document. For example, suppose you have XML documents such as the following stored in an `XMLType` table `dept_xml_tab`:

```
<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>
```

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>'));

COMMIT;

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.

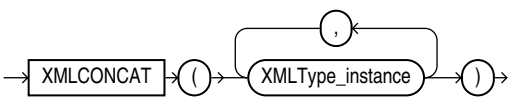
See Also: Chapter 4, "XMLType Operations"

XMLConcat() Function

`XMLConcat()` function concatenates all the arguments passed in to create a XML fragment. Figure 15–5 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, then it is ignored in the result. If all the values are NULL, then the result is NULL. This form is used to concatenate arbitrary number of `XMLType` instances in the same row. `XMLAgg()` can be used to concatenate `XMLType` instances across rows.

Figure 15–5 XMLConcat() Syntax



Example 15–11 XMLConcat(): Returning a Concatenation of XML Elements Used in the Argument Sequence

This example shows `XMLConcat()` returning the concatenation of `XMLType`s from the `XMLSequenceType`:

```
SELECT XMLConcat(XMLSequenceType(xmltype('<PartNo>1236</PartNo>'),
                                 xmltype('<PartName>Widget</PartName>'),
                                 xmltype('<PartPrice>29.99</PartPrice>')
                )).getClobVal()
  AS "result"
  FROM dual;
```

returns a single fragment of the form:

```
result
---------------
<PartNo>1236</PartNo>
<PartName>Widget</PartName>
<PartPrice>29.99</PartPrice>

1 row selected.
```

Example 15–12 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:

```
result
---------------
<first>Mary</first>
<last>Martin</last>
<first>John</first>
<last>Smith</last>

2 rows selected.
```

XMLAgg() Function

`XMLAgg()` is an aggregate function that produces a forest of XML elements from a collection of XML elements. Figure 15–6 describes the `XMLAgg()` syntax, where the order_by_clause is:

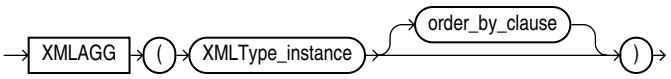
```
ORDER BY [list of: expr [ASC|DESC] [NULLS {FIRST|LAST}]]
```

and number literals are not interpreted as column positions. For example, `ORDER BY 1` does not mean order by the first column. Instead the number literals are interpreted just as any other literal.

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. It also allows an optional ORDER BY clause to order the XML values being aggregated.

XMLAgg() is an aggregation function and hence produces one aggregated XML result for each group. If there is no group by specified in the query, then it returns a single aggregated XML result for all the rows of the query.

Figure 15–6 XMLAgg() Syntax



Example 15–13 XMLAgg(): Generating Department Elements with a List of Employee Elements

The following example produces a Department element containing Employee elements with employee job ID and last name as the contents of the elements. It also orders the employee XML elements in the department by their last name.

```
SELECT XMLELEMENT("Department", XMLAGG(XMLELEMENT("Employee",
                                                  e.job||' '||e.ename)
                                       ORDER BY e.ename))
  AS "Dept_list"
  FROM scott.emp e
  WHERE e.deptno = 10;

Dept_list
------------------
<Department>
  <Employee>MANAGER CLARK</Employee>
  <Employee>PRESIDENT KING</Employee>
  <Employee>CLERK MILLER</Employee>
</Department>

1 row selected.
```

The result is a single row, because XMLAgg() aggregates the rows. You can use the GROUP BY clause to group the returned set of rows into multiple groups:

```
SELECT XMLELEMENT("Department", XMLAttributes(deptno AS "deptno"),
                  XMLAgg(XMLElement("Employee", e.job||' '||e.ename)))
   AS "Dept_list"
   FROM scott.emp e
   GROUP BY e.deptno;

Dept_list
------------------
<Department deptno="10">
  <Employee>MANAGER CLARK</Employee>
  <Employee>PRESIDENT KING</Employee>
  <Employee>CLERK MILLER</Employee>
</Department>

<Department deptno="20">
  <Employee>CLERK SMITH</Employee>
  <Employee>ANALYST FORD</Employee>
  <Employee>CLERK ADAMS</Employee>
  <Employee>ANALYST SCOTT</Employee>
  <Employee>MANAGER JONES</Employee>
</Department>
```

```
<Department deptno="30">
  <Employee>SALESMAN ALLEN</Employee>
  <Employee>MANAGER BLAKE</Employee>
  <Employee>SALESMAN MARTIN</Employee>
  <Employee>SALESMAN TURNER</Employee>
  <Employee>CLERK JAMES</Employee>
  <Employee>SALESMAN WARD</Employee>
</Department>

3 rows selected.
```

You can order the employees within each department by using the `ORDER BY` clause inside the `XMLAgg()` expression.

> **Note:** Within the *`order_by_clause`*, Oracle Database does not interpret number literals as column positions, as it does in other uses of this clause, but simply as number literals.

Example 15–14 XMLAgg(): Generating Department Elements, Employee Elements in Each 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 the employee dependents:

```
CREATE TABLE scott.dependents (id NUMBER(4) PRIMARY KEY,
                               empno NUMBER(4),
                               name VARCHAR2(10));
INSERT INTO scott.dependents values (1, 7369, 'MARK');
INSERT INTO scott.dependents values (2, 7369, 'JACK');
INSERT INTO scott.dependents values (3, 7499, 'JANE');
INSERT INTO scott.dependents values (4, 7521, 'HELLEN');
INSERT INTO scott.dependents values (5, 7521, 'FRANK');
INSERT INTO scott.dependents values (6, 7566, 'JANUS');
INSERT INTO scott.dependents values (7, 7654, 'KATE');
INSERT INTO scott.dependents values (8, 7654, 'JEFF');
INSERT INTO scott.dependents values (9, 7654, 'JENNIFER');
INSERT INTO scott.dependents values (10, 7654, 'JOHN');
INSERT INTO scott.dependents values (11, 7698, 'BUSH');
INSERT INTO scott.dependents values (12, 7782, 'BUSH W');
INSERT INTO scott.dependents values (13, 7788, 'WALLACE');
INSERT INTO scott.dependents values (14, 7788, 'FRED');
INSERT INTO scott.dependents values (15, 7839, 'GALE');
INSERT INTO scott.dependents values (16, 7839, 'GARY');
INSERT INTO scott.dependents values (17, 7876, 'JOE');
INSERT INTO scott.dependents values (18, 7902, 'NAISON');
INSERT INTO scott.dependents values (19, 7902, 'JOYCE');
INSERT INTO scott.dependents values (20, 7902, 'NAISON');
INSERT INTO scott.dependents values (21, 7934, 'JOHNSON');
INSERT INTO scott.dependents values (22, 7934, 'BUCKS');

COMMIT;

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:

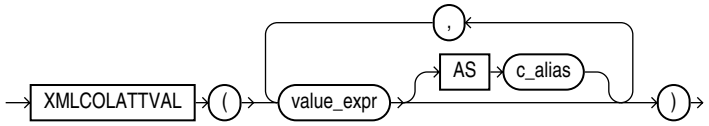
```
dept_list
----------------------
<Department name="ACCOUNTING">
  <emp NAME="CLARK">
    <dependent name="BUSH W"/dependent>
  </emp>
  <emp NAME="KING"/>
    <dependent name="GALE"/dependent>
    <dependent name="GARY"/dependent>
  </emp>
  <emp NAME="MILLER">
    <dependent name="JOHNSON"/dependent>
    <dependent name="BUCKS"/dependent>
  </emp>
</Department>
<Department name="RESEARCH">
  <emp NAME="SMITH">
    <dependent name="MARK"/dependent>
    <dependent name="JACK"/dependent>
  </emp>
  <emp NAME="JONES">
    <dependent name="JANUS"/dependent>
  </emp>
  <emp NAME="SCOTT">
    <dependent name="WALLACE"/dependent>
    <dependent name="FRED"/dependent>
  </emp>
  <emp NAME="ADAMS">
    <dependent name="JOE"/dependent>
  </emp>
  <emp NAME="FORD">
    <dependent name="NAISON"/dependent>
    <dependent name="JOYCE"/dependent>
  </emp>
</Department>
<Department name="SALES">
  <emp NAME="ALLEN">
    <dependent name="JANE"/dependent>
    <dependent name="HELLEN"/dependent>
  </emp>
  <emp NAME="WARD">
    <dependent name="FRANK"/dependent>
  </emp>
  <emp NAME="MARTIN">
    <dependent name="KATE"/dependent>
    <dependent name="JEFF"/dependent>
    <dependent name="JENNIFER"/dependent>
    <dependent name="JOHN"/dependent>
```

```
  </emp>
  <emp NAME="BLAKE">
    <dependent name="BUSH"/dependent>
  </emp>
  <emp NAME="TURNER"></emp>
  <emp NAME="JAMES"></emp>
</Department>
<Department name="OPERATIONS"></Department>

4 rows selected.
```

XMLColAttVal() Function

`XMLColAttVal()` function generates a forest of XML `column` elements containing the value of the arguments passed in. This function is an Oracle Database extension to the SQL/XML ANSI-ISO standard functions. Figure 15–7 shows the `XMLColAttVal()` syntax.

Figure 15–7 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 15–15 XMLColAttVal(): Generating an Emp Element For Each 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.department AS "department"))
  AS "result"
  FROM employees e;
```

This query might produce the following XML result:

```
result
-------------
<Emp name="John Smith">
   <column name="HIRE">24-MAY-00</column>
   <column name="department">Accounting</column>
</Emp>
<Emp name="Mary Martin">
   <column name="HIRE">01-FEB-96</column>
   <column name="department">Shipping</column>
</Emp>

2 rows selected.
```

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.

If `NLS_DATE_FORMAT` is set to `YYYY-MM-DD`, then the date is in XML schema date format. The same query then produces this result:

```
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>

2 rows selected.
```

Generating XML from Oracle 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 *Oracle XML Developer's Kit Programmer's Guide*, 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>7369</EMPNO>
  <ENAME>SMITH</ENAME>
  <JOB>CLERK</JOB>
  <MGR>7902</MGR>
  <HIREDATE>17-DEC-80</HIREDATE>
  <SAL>800</SAL>
  <DEPTNO>20</DEPTNO>
</ROW>
```

```
<ROW>
  <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>
</ROW>
...
</ROWSET>
```

The result of the `getXML()` using `DBMS_XMLGen` package is a `CLOB`. The default mapping is as follows:

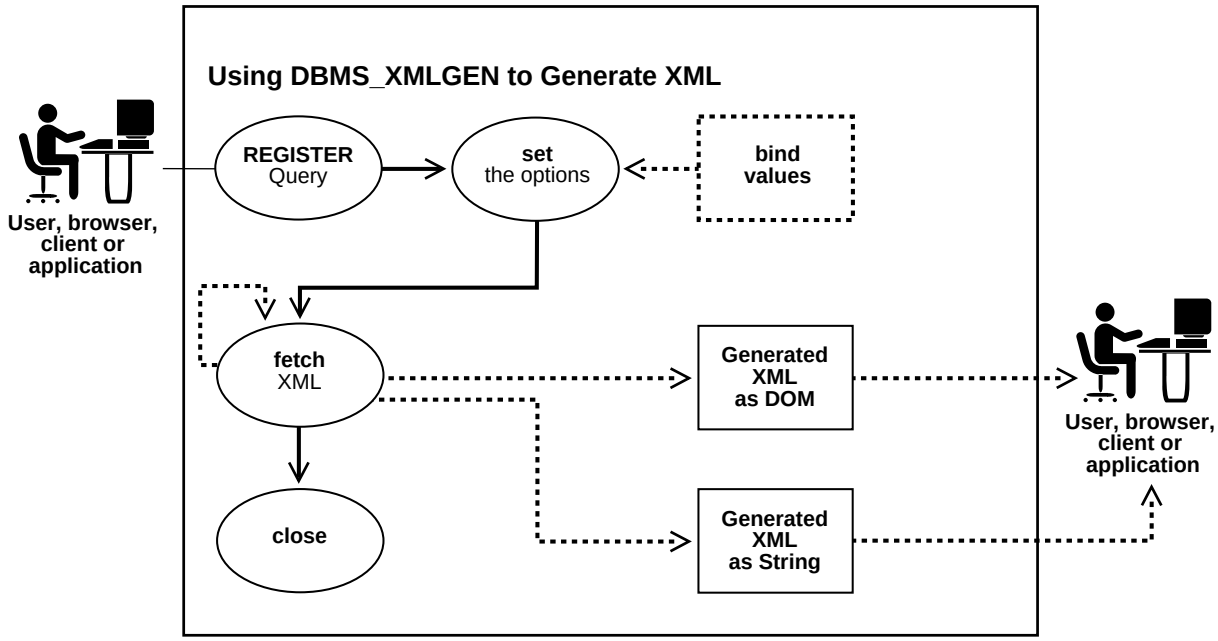
- Every row of the query result maps to an XML element with the default tag name `ROW`.
- The entire result is enclosed in a `ROWSET` element. These names are both configurable, using the `setRowTagName()` and `setRowSetTagName()` procedures in `DBMS_XMLGEN`.
- Each column in the SQL query result, maps 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 15–8 summarizes the `DBMS_XMLGEN` calling sequence.

Figure 15–8 DBMS_XMLGEN Calling Sequence



Here is the `DBMS_XMLGEN` 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 procedures or functions in the package to set the various options. For example, to set the `ROW` element name, use `setRowTag(ctx)`, where `ctx` is the context got from the previous `newContext()` call.
3. Get the XML result, using 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, then 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 15–1 summarizes `DBMS_XMLGEN` functions and procedures.

Table 15–1 DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
`DBMS_XMLGEN Type definitions` `SUBTYPE ctxHandle IS NUMBER`	The context handle used by all functions. Document Type Definition (DTD) or schema specifications: NONE CONSTANT NUMBER:= 0; -- supported for this release. DTD CONSTANT NUMBER:= 1; SCHEMA CONSTANT NUMBER:= 2; Can be used in `getXML` function to specify whether to generate a DTD or XML Schema or none. Only the NONE specification is supported in the `getXML` functions for this release.
`FUNCTION PROTOTYPES` `newContext()`	Given a query string, generate a new context handle to be used in subsequent functions.
`FUNCTION` `newContext(queryString IN VARCHAR2)`	Returns a new context PARAMETERS: `queryString (IN)`- the query string, the result of which must be converted to XML RETURNS: Context handle. Call this function first to obtain a handle that you can use in the `getXML()` and other functions to get the XML back from the result.
`FUNCTION` `newContext(queryString IN SYS_REFCURSOR) RETURN ctxHandle;`	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:

Table 15–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
FUNCTION newContextFromHierarchy(queryString IN VARCHAR2) RETURN ctxHandle; This is new in Oracle Database 10*g* release 1 (10.0.1).	Returns a new context PARAMETERS: queryString (IN)- the query string, the result of which must be converted to XML. The query is a hierarchical query typically formed using a `CONNECT BY` clause, and the result must have the same property as the result set generated by a `CONNECT BY` query. The result set must have only two columns, the level number and an XML value. The level number is used to determine the hierarchical position of the XML value within the result XML document. RETURNS: Context handle. Call this function first to obtain a handle that you can use in the `getXML()` and other functions to get a hierarchical XML with recursive elements back from the result.
`setRowTag()`	Sets the name of the element separating all the rows. The default name is `ROW`.
PROCEDURE `setRowTag(ctx IN ctxHandle,rowTag IN VARCHAR2);`	PARAMETERS: `ctx(IN)` - the context handle obtained from the newContext call, `rowTag(IN)` - the name of the `ROW` element. `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 root element. The default name is `ROWSET`
PROCEDURE `setRowSetTag(ctx IN ctxHandle, rowSetTag IN VARCHAR2);`	PARAMETERS: `ctx(IN)` - the context handle obtained from the newContext call, `rowSetTag(IN)` - the name of the document element. `NULL` indicates that you do not want the `ROW` element to be present. Call this to set the name of the document root element, if you do not want the default `ROWSET` name in the output. You can also set this to `NULL` to suppress the printing of this element. However, this is an error if both the `ROW` and the `ROWSET` are `NULL` and there is more than one column or row in the output.
`getXML()`	Gets the XML document by fetching the maximum number of rows specified. It appends the XML document to the `CLOB` passed in.

Table 15–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
PROCEDURE `getXML(ctx IN ctxHandle, clobval IN OUT NCOPY clob, dtdOrSchema IN number:= NONE);`	PARAMETERS: ctx(IN) - The context handle obtained from the newContext() call, `clobval(IN/OUT)` - the `CLOB` to which the XML document is to be appended, `dtdOrSchema(IN)` - whether 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.
`getXML()`	Generates the XML document and returns it as a CLOB.
`FUNCTION` `getXML(ctx IN ctxHandle, dtdOrSchema IN number:= NONE) RETURN clob;`	`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.
`FUNCTION` `getXMLType(ctx IN ctxHandle, dtdOrSchema IN number:= NONE) RETURN XMLType;`	`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.
`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 `getXML` call. This count does not include the number of rows skipped before generating the XML.
`FUNCTION` `getNumRowsProcessed(ctx IN ctxHandle) RETURN number;`	PARAMETERS: `queryString(IN)`- the query string, the result of which needs to be converted to XML RETURNS: This gets the number of SQL rows that were processed in the last call to getXML. You can call this to find out if the end of the result set has been reached. This does not include the number of rows skipped. Use this function to determine the terminating condition if you are calling getXML in a loop. Note that getXML would always generate a XML document even if there are no rows present.

Table 15–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
`setMaxRows()`	Sets the maximum number of rows to fetch from the SQL query result for every invocation of the `getXML` call. It is an error to call this function on a context handle created by newContextFromHierary() function
`PROCEDURE` setMaxRows(ctx IN ctxHandle, maxRows IN NUMBER);	PARAMETERS: ctx(IN) - the context handle corresponding to the query executed, `maxRows(IN)` - the maximum number of rows to get for each call to `getXML`. The maxRows parameter can be used when generating paginated results using this utility. For instance when generating a page of XML or HTML data, you can restrict the number of rows converted to XML and then in subsequent calls, you can get the next set of rows and so on. This also can provide for faster response times. It is an error to call this procedure on a context handle created by newContextFromHierary() function
`setSkipRows()`	Skips a given number of rows before generating the XML output for every call to the `getXML()` routine. It is an error to call this function on a context handle created by `newContextFormHierarchy()` function
`PROCEDURE` `setSkipRows(ctx IN ctxHandle, skipRows IN NUMBER);`	PARAMETERS:ctx(IN) - the context handle corresponding to the query executed, `skipRows(IN)` - the number of rows to skip for each call to getXML. The skipRows parameter can be used when generating paginated results for stateless web pages using this utility. For instance when generating the first page of XML or HTML data, you can set `skipRows` to zero. For the next set, you can set the `skipRows` to the number of rows that you got in the first case. It is an error to call this function on a context handle created by `newContextFromHierarchy()` function.
`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 must be preceded by an escape character. 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 it the type name itself. You can override that to use the name of the column with the `_ITEM` tag appended to it using this function.
`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 action and generate the collection column name with the `_ITEM` tag appended to it, by calling this procedure.

Table 15–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
`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 `DBMS_XMLGEN` 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. ■ Encoding refers to replacing entity references such as '<' to their escaped equivalent, such as '`<`'. ■ 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. ■ Encoding refers to replacing entity references such as '<' to their escaped equivalent, such as '`<`'. ■ Decoding refers to the reverse conversion.

NULL Handling
`PROCEDURE`
`setNullHandling(ctx IN ctxHandle, flag IN NUMBER);`
This is new in Oracle9*i* release 2 (9.2.0.2).
	The `setNullHandling` flag values are:
	■ DROP_NULLS CONSTANT NUMBER := 0; This is the default setting and leaves out the tag for null elements.
	■ NULL_ATTR CONSTANT NUMBER := 1; This sets xsi:nil="true".
	■ EMPTY_TAG CONSTANT NUMBER := 2; This sets, for example, <foo/>.

`PROCEDURE`
`useNullAttributeIndicator(ctx IN ctxHandle, attrind IN boolean := TRUE);`
This is new in Oracle9*i* release 2 (9.2.0.2).
	`useNullAttributeIndicator` is a short-cut for `setNullHandling(ctx, NULL_ATTR)`.

`PROCEDURE`
setBindValue(ctx IN ctxHandle, bindVariableName IN VARCHAR2, bindValue IN VARCHAR2);
This is new in Oracle Database 10*g* release 1 (10.0.1).
	This procedure allows one to set bind value for the bind variable appearing in the query string associated with the context handle. The query string with bind variables cannot be executed until all the bind variables are set values using `setBindValue()` call.

`PROCEDURE`
clearBindValue(ctx IN ctxHandle);
This is new in Oracle Database 10*g* release 1 (10.0.1).
	This procedure clears all the bind values for all the bind variables appearing in the query string associated with the context handle. Afterwards, all the bind variables have to rebind new values using `setBindValue()` call.

Example 15–16 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 15–17 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 Document Object Model (DOM) Application Program Interface (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
    -- get the result
    result := dbms_xmlgen.getXML(qryCtx);
    -- if no rows were processed, then quit
    exit when dbms_xmlgen.getNumRowsProcessed(qryCtx) = 0;

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

Here, for each set of 5 rows, you generate an XML document.

Example 15–18 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));
/
INSERT INTO new_departments VALUES(10, 'SALES');
INSERT INTO new_departments VALUES(20, 'ACCOUNTING');
INSERT INTO new_employees VALUES(30, 'Scott', 10);
INSERT INTO new_employees VALUES(31, 'Marry', 10);
INSERT INTO new_employees VALUES(40, 'John', 20);
INSERT INTO new_employees VALUES(41, 'Jerry', 20);

COMMIT;

CREATE TYPE emplist_t AS TABLE OF emp_t;
/

CREATE TYPE dept_t AS OBJECT("@department_id" NUMBER,
                             department_name VARCHAR2(20),
                             emplist emplist_t);
/
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  result CLOB;
```

```
BEGIN
  DBMS_XMLGEN.setRowTag(qryCtx, NULL);
  qryCtx := DBMS_XMLGEN.newContext
     ('SELECT
         dept_t(department_id, department_name,
                CAST(MULTISET
                      (SELECT e.employee_id, e.last_name
                       FROM new_employees e
                       WHERE e.department_id = d.department_id)
                     AS emplist_t))
         AS deptxml
         FROM new_departments d');
-- now get the result
result := DBMS_XMLGEN.getXML(qryCtx);
INSERT INTO temp_clob_tab VALUES (result);
-- close context
DBMS_XMLGEN.closeContext(qryCtx);
END;
/
SELECT * FROM temp_clob_tab;
```

Here is the resulting XML:

```
RESULT
----------------------
<?xml version="1.0"?>
<ROWSET>
  <ROW>
    <DEPTXML department_id="10">
      <DEPARTMENT_NAME>SALES</DEPARTMENT_NAME>
      <EMPLIST>
        <EMP_T employee_id="30">
          <LAST_NAME>Scott</LAST_NAME>
        </EMP_T>
        <EMP_T employee_id="31">
          <LAST_NAME>Marry</LAST_NAME>
        </EMP_T>
      </EMPLIST>
    </DEPTXML>
  </ROW>
  <ROW>
    <DEPTXML department_id="20">
      <DEPARTMENT_NAME>ACCOUNTING</DEPARTMENT_NAME>
      <EMPLIST>
        <EMP_T employee_id="40">
          <LAST_NAME>John</LAST_NAME>
        </EMP_T>
        <EMP_T employee_id="41">
          <LAST_NAME>Jerry</LAST_NAME>
        </EMP_T>
      </EMPLIST>
    </DEPTXML>
  </ROW>
</ROWSET>
```

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 section "Sample DBMS_XMLGEN Query Result" on page 15-19.

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, "XML Schema Storage and Query: The Basics".
- ***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 15–19 DBMS_XMLGEN: Generating Complex XML #2 - Inputting User Defined Types For Nested XML Documents

When you enter 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 *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:

```
-- empno is defined with '@' in front to indicate that it must
-- be mapped as an attribute of the enclosing Employee element.
CREATE TYPE EMP_T AS OBJECT("@empno" number,  -- empno defined as attribute
                            ename varchar2(20));
/
CREATE TYPE EMPLIST_T AS TABLE OF EMP_T;
/
CREATE TYPE DEPT_T AS OBJECT("@deptno" number, dname varchar2(20),
                            emplist emplist_t);
/

-- Department type DEPT_T contains a list of employees.
-- We 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 query context
  qryCtx := DBMS_XMLGEN.newContext(
 'SELECT
  dept_t(deptno, dname, CAST(MULTISET(SELECT empno, ename
                                      FROM emp e
                                      WHERE e.deptno = d.deptno)
                            AS emplist_t))
  AS deptxml
  FROM dept d');
  -- set maximum number of rows to 5,
```

```
  DBMS_XMLGEN.setMaxRows(qryCtx, 5);
  -- set no row tag for this result, since there is a single ADT column
  DBMS_XMLGEN.setRowTag(qryCtx, NULL);
  LOOP
    -- get result
    result := DBMS_XMLGEN.getXML(qryCtx);
    -- if there were no rows processed, then quit
    EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed(qryCtx) = 0;
    -- do something with the result
    -- ...
  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, assigns 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 15–20 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 attribute 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 number */
                     Custno     NUMBER  /*  Foreign KEY referencing customer */
                                CONSTRAINT po_cust_fk REFERENCES Customer_tab,
                     OrderDate  DATE, /*  date of order */
                     ShipDate   TIMESTAMP, /* date to be shipped */
                     ToStreet   VARCHAR2(200), /* shipto address */
                     ToCity     VARCHAR2(200),
                     ToState    CHAR(2),
                     ToZip      VARCHAR2(20),
                     CONSTRAINT po_pk PRIMARY KEY(PONo));

--Stock Table
CREATE TABLE Stock_tab (StockNo NUMBER CONSTRAINT stock_uk UNIQUE,
```

```
                        Price   NUMBER,
                        TaxRate NUMBER);

--Line Items table
CREATE TABLE LineItems_tab(LineItemNo NUMBER,
                           PONo       NUMBER
                                      CONSTRAINT li_po_fk REFERENCES po_tab,
                           StockNo    NUMBER,
                           Quantity   NUMBER,
                           Discount   NUMBER,
                           CONSTRAINT li_pk PRIMARY KEY (PONo, LineItemNo));

-- create Object views

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

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

-- Create table with XMLType column to store purchase order in XML format
CREATE TABLE po_xml_tab(poid  NUMBER,
                        poDoc XMLType) /* PO 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 purchase order in XML format
-- and store XMLType in po_xml table
-----------------------------------------------------
DECLARE
   qryCtx DBMS_XMLGEN.ctxHandle;
   pxml XMLType;
   cxml CLOB;
BEGIN
  -- get query context;
  qryCtx := DBMS_XMLGEN.newContext('SELECT pono,deref(cust_ref) customer,
                                           p.OrderDate,
                                           p.shipdate,
                                           lineitems_ntab lineitems,
                                           shiptoaddr
                                    FROM po p');
  -- set maximum number of rows to be 1,
  DBMS_XMLGEN.setMaxRows(qryCtx, 1);
  -- set ROWSET tag to null and ROW tag to PurchaseOrder
  DBMS_XMLGEN.setRowSetTag(qryCtx, NULL);
  DBMS_XMLGEN.setRowTag(qryCtx,'PurchaseOrder');
  LOOP
    -- get purchase order in XML format
    pxml := DBMS_XMLGEN.getXMLType(qryCtx);
    -- if there were no rows processed, then quit
    EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed(qryCtx) = 0;
    -- Store XMLType po in po_xml table (get the pono out)
    INSERT INTO po_xml_tab(poid, poDoc)
      VALUES(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 15–21 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;
/
```

Example 15–22 DBMS_XMLGEN: Specifying Null Handling

```
SQL> @tkxmgnd10
SQL> CONNECT system/manager
Connected.
SQL>
SQL> GRANT CONNECT, RESOURCE TO gnd10 IDENTIFIED BY gnd10;
Grant succeeded.
SQL> CONNECT gnd10/gnd10
Connected.
SQL> SET SERVEROUTPUT ON SIZE 200000
SQL> -- null_handle: 1 => NULL_ATTR, 2 => EMPTY_TAG
SQL> CREATE OR REPLACE FUNCTION getXML(sql_query VARCHAR2, null_handle NUMBER,
```

```
 2                                    rset_tag VARCHAR2 := 'ROWSET',
 3                                    rtag VARCHAR2 := 'ROW')
 4  RETURN CLOB IS
 5    ctx NUMBER;
 6    xmldoc CLOB;
 7  BEGIN
 8    ctx := DBMS_XMLGEN.newContext(sql_query);
 9
10    IF (nvl(rset_tag, 'X') != 'ROWSET') THEN
11       DBMS_XMLGEN.setRowSetTag(ctx, rset_tag);
12    END IF;
13
14    IF (nvl(rtag, 'Y') != 'ROW') THEN
15       DBMS_XMLGEN.setRowTag(ctx, rtag);
16    END IF;
17
18    DBMS_XMLGEN.setNullHandling(ctx, null_handle);
19
20    xmldoc := DBMS_XMLGEN.getXML(ctx);
21    DBMS_XMLGEN.closeContext(ctx);
22    RETURN xmldoc;
23  END;
24  /
Function created.
```

Example 15–23 DBMS_XMLGEN : Generating a Hierarchical XML Document Containing Recursive Elements From the Result of a Hierarchical Query (Formed by a CONNECT BY Clause)

The `DBMS_XMLGEN` package contains the function `newContextFromHierarchy()`. It takes a hierarchical query string, which is typically formulated with a CONNECT BY clause, as an argument and returns a context to be used to generate a hierarchical XML document with recursive elements.

The hierarchical query returns two columns, the level number (a pseudo-column generated by CONNECT BY query) and an `XMLType`. The level is used to determine the position of the `XMLType` value within the hierarchy of the result XML document.

Setting skip number of rows or maximum number of rows for a context created from `newContextFromHierarchy()` is an error.

For example, you can generate a Manager employee hierarchy by using `DBMS_XMLGEN.newContextFormHierarchy()`.

```
SQL> set serveroutput on size 200000
SQL> set long 200000
SQL>

SQL> create table sqlx_display(id number, xmldoc XMLType);

Table created.

SQL>
SQL> -- Test 2: XMLelement with scott schema
SQL> declare
  2         qryctx dbms_xmlgen.ctxhandle;
  3         result XMLType;
  4  begin
  5    qryctx := dbms_xmlgen.newcontextFromHierarchy(
  6    'select level, xmlelement("emp", xmlelement("enumber", empno),
  7           xmlelement("name", ename),
```

```
  8           xmlelement("Salary", sal),
  9           xmlelement("Hiredate", hiredate)) from scott.emp
 10         start with ename=''KING'' connect by prior empno=mgr
 11         order siblings by hiredate');
 12    result := dbms_xmlgen.getxmltype(qryctx);
 13    dbms_output.put_line('<result num rows>');
 14    dbms_output.put_line(to_char(dbms_xmlgen.getNumRowsProcessed(qryctx
)));
 15    dbms_output.put_line('</result num rows>');
 16    insett into sqlx_display values  (2, result);
 17    commit;
 18    dbms_xmlgen.closecontext(qryctx);
 19  end;
 20  /
<result num rows>
14
</result num rows>

PL/SQL procedure successfully completed.

SQL>
SQL> select xmldoc from sqlx_display where id = 2;

XMLDOC
--------------------------------------------------------------------------------
<?xml version="1.0"?>
<emp>
  <enumber>7839</enumber>
  <name>KING</name>
  <Salary>5000</Salary>
  <Hiredate>17-NOV-81</Hiredate>
  <emp>
    <enumber>7566</enumber>
    <name>JONES</name>
    <Salary>2975</Salary>
    <Hiredate>02-APR-81</Hiredate>
    <emp>
      <enumber>7902</enumber>
      <name>FORD</name>
      <Salary>3000</Salary>
      <Hiredate>03-DEC-81</Hiredate>
      <emp>
        <enumber>7369</enumber>
        <name>SMITH</name>
        <Salary>800</Salary>
        <Hiredate>17-DEC-80</Hiredate>
      </emp>
    </emp>
    <emp>
      <enumber>7788</enumber>
      <name>SCOTT</name>
      <Salary>3000</Salary>
      <Hiredate>19-APR-87</Hiredate>
      <emp>
        <enumber>7876</enumber>
        <name>ADAMS</name>
        <Salary>1100</Salary>
        <Hiredate>23-MAY-87</Hiredate>
      </emp>
    </emp>
```

```
  </emp>
  <emp>
    <enumber>7698</enumber>
    <name>BLAKE</name>
    <Salary>2850</Salary>
    <Hiredate>01-MAY-81</Hiredate>
    <emp>
      <enumber>7499</enumber>
      <name>ALLEN</name>
      <Salary>1600</Salary>
      <Hiredate>20-FEB-81</Hiredate>
    </emp>
    <emp>
      <enumber>7521</enumber>
      <name>WARD</name>
      <Salary>1250</Salary>
      <Hiredate>22-FEB-81</Hiredate>
    </emp>
    <emp>
      <enumber>7844</enumber>
      <name>TURNER</name>
      <Salary>1500</Salary>
      <Hiredate>08-SEP-81</Hiredate>
    </emp>
    <emp>
      <enumber>7654</enumber>
      <name>MARTIN</name>
      <Salary>1250</Salary>
      <Hiredate>28-SEP-81</Hiredate>
    </emp>
    <emp>
      <enumber>7900</enumber>
      <name>JAMES</name>
      <Salary>950</Salary>
      <Hiredate>03-DEC-81</Hiredate>
    </emp>
  </emp>
  <emp>
    <enumber>7782</enumber>
    <name>CLARK</name>
    <Salary>2450</Salary>
    <Hiredate>09-JUN-81</Hiredate>
    <emp>
      <enumber>7934</enumber>
      <name>MILLER</name>
      <Salary>1300</Salary>
      <Hiredate>23-JAN-82</Hiredate>
    </emp>
  </emp>
</emp>

1 row selected.
```

By default, the `RowSet` tag is `NULL`. That is, there is no default rowset tag used to enclose the XML result. However, you can explicitly set the rowset tag by using the `setRowSetTag()` procedure, as follows:

```
SQL>
SQL> create table gg(x XMLType);
```

```
Table created.
SQL> declare
  2          qryctx dbms_xmlgen.ctxhandle;
  3          result clob;
  4  begin
  5          qryctx := dbms_xmlgen.newcontextFromHierarchy('select level,
  6                  xmlelement("NAME", name) as myname from tc
  7                  connect by prior id = pid start with id = 1');
  8          dbms_xmlgen.setRowSetTag(qryctx, 'mynum_hierarchy');
  9          result:=dbms_xmlgen.getxml(qryctx);
 10
 11          dbms_output.put_line('<result num rows>');
 12          dbms_output.put_line(to_char(dbms_xmlgen.getNumRowsProcessed(qryctx
)));
 13          dbms_output.put_line('</result num rows>');
 14          insert into gg values(xmltype(result));
 15          commit;
 16          dbms_xmlgen.closecontext(qryctx);
 17  end;
 18  /
<result num rows>
6
</result num rows>

PL/SQL procedure successfully completed.

SQL> select * from gg;

X
--------------------------------------------------------------------------------
<?xml version="1.0"?>
<mynum_hierarchy>
  <NAME>top
    <NAME>second1
      <NAME>third3</NAME>
    </NAME>
    <NAME>second2
      <NAME>third1</NAME>
      <NAME>third2</NAME>
    </NAME>
  </NAME>
</mynum_hierarchy>
```

Example 15–24 DBMS_XMLGEN : Using setBindValue() to Bind Variables in the Query String for DBMS_XMLGEN

The query string that is used to create context can contain host variables and then bind values to it by using `setBindValue()` before the execution of the query.

```
SET SERVEROUTPUT ON SIZE 200000
SET LONG 200000

-- bind one variable
DECLARE
  2  ctx NUMBER;
  3  xmldoc CLOB;
  4  BEGIN
  5    ctx := DBMS_XMLGEN.newContext('SELECT * FROM emp WHERE empno = :NO');
  6
  7    DBMS_XMLGEN.setBindValue(ctx, 'NO', '7369');
  8    xmldoc := DBMS_XMLGEN.getXML(ctx);
```

```
  9    printClobOut(xmldoc);
 10    DBMS_XMLGEN.closeContext(ctx);
 11  EXCEPTION
 12    WHEN OTHERS THEN
 13    DBMS_XMLGEN.closeContext(ctx);
 14    RAISE;
 15  END;
 16  /
| <?xml version="1.0"?>
| <ROWSET>
|   <ROW>
|     <EMPNO>7369</EMPNO>
|     <ENAME>SMITH</ENAME>
|     <JOB>CLERK</JOB>
|     <MGR>7902</MGR>
|     <HIREDATE>17-DEC-80</HIREDATE>
|     <SAL>800</SAL>
|     <DEPTNO>20</DEPTNO>
|   </ROW>
| </ROWSET>

SQL> --bind one variable twice with different values
SQL> DECLARE
  2  ctx NUMBER;
  3  xmldoc CLOB;
  4  BEGIN
  5    ctx := DBMS_XMLGEN.newContext('SELECT * FROM emp
                                      WHERE hiredate = :MDATE');
  6
  7    DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '17-DEC-80');
  8    xmldoc := DBMS_XMLGEN.getXML(ctx);
  9    printClobOut(xmldoc);
 10
 11    DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '20-FEB-81');
 12    xmldoc := DBMS_XMLGEN.getXML(ctx);
 13    printClobOut(xmldoc);
 14    DBMS_XMLGEN.closeContext(ctx);
 15  EXCEPTION
 16    WHEN OTHERS THEN
 17    DBMS_XMLGEN.closeContext(ctx);
 18    RAISE;
 19  END;
 20  /
| <?xml version="1.0"?>
| <ROWSET>
|   <ROW>
|     <EMPNO>7369</EMPNO>
|     <ENAME>SMITH</ENAME>
|     <JOB>CLERK</JOB>
|     <MGR>7902</MGR>
|     <HIREDATE>17-DEC-80</HIREDATE>
|     <SAL>800</SAL>
|     <DEPTNO>20</DEPTNO>
|   </ROW>
| </ROWSET>
| <?xml version="1.0"?>
| <ROWSET>
|   <ROW>
|     <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>
|   </ROW>
| </ROWSET>

PL/SQL procedure successfully completed.
SQL> -- bind two variables
SQL> DECLARE
  2  ctx NUMBER;
  3  xmldoc CLOB;
  4  BEGIN
  5    ctx := DBMS_XMLGEN.newContext('SELECT * FROM emp
  6                                   WHERE empno = :NO
  7                                     AND hiredate = :MDATE');
  8    DBMS_XMLGEN.setBindValue(ctx, 'NO', '7369');
  9    DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '17-DEC-80');
 10    xmldoc := DBMS_XMLGEN.getXML(ctx);
 11    printClobOut(xmldoc);
 12    DBMS_XMLGEN.closeContext(ctx);
 13  EXCEPTION
 14    WHEN OTHERS THEN
 15    DBMS_XMLGEN.closeContext(ctx);
 16    RAISE;
 17  END;
 18  /
| <?xml version="1.0"?>
| <ROWSET>
|   <ROW>
|     <EMPNO>7369</EMPNO>
|     <ENAME>SMITH</ENAME>
|     <JOB>CLERK</JOB>
|     <MGR>7902</MGR>
|     <HIREDATE>17-DEC-80</HIREDATE>
|     <SAL>800</SAL>
|     <DEPTNO>20</DEPTNO>
|   </ROW>
| </ROWSET>
```

Generating XML Using Oracle Database-Provided SQL Functions

In addition to the SQL standard functions, Oracle Database provides the `SYS_XMLGEN` and `SYS_XMLAGG` functions to aid in generating XML.

SYS_XMLGEN() Function

This Oracle Database-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 15–25 Using SQL_XMLGEN to Create XML

`SYS_XMLGEN()` creates and queries XML instances in SQL queries, as follows:

```
SELECT SYS_XMLGEN(employee_id) AS "result"
   FROM employees WHERE fname LIKE 'John%';
```

The resulting XML document is:

```
result
--------------
<?xml version="1.0"?>
<EMPLOYEE_ID>1001</EMPLOYEE_ID>

1 row selected.
```

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 15–9.

Figure 15–9 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, then the function returns an XML element containing the scalar value.
- If *expr* is a type, then 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, then the enclosing XML element will have the same name as the column. If you want to format the XML document differently, then specify *fmt*, which is an instance of the `XMLFormat` object.

The formatting argument for `SYS_XMLGEN()` accepts the schema and element name, and generates the XML document conforming to that registered schema.

```
CREATE OR REPLACE TYPE scott.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 scott.emplist_t AS TABLE OF emp_t;
/
CREATE OR REPLACE TYPE scott.dept_t AS OBJECT(DEPTNO NUMBER(2),
                                             DNAME VARCHAR2(14),
                                             LOC VARCHAR2(13),
                                             EMPLIST EMPLIST_T);
/
```

```
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 scott.emp e
                                WHERE e.deptno = d.deptno)
                              AS emplist_t)),
                  xmlformat.createformat('Department'))
FROM scott.dept d;
SELECT SYS_XMLGEN(x) FROM table_name WHERE x IS NOT NULL;
```

Suppressing <ROW/> Tags

To suppress <ROW/> tags with SYS_XMLGEN() if you do not want NULL values represented, just use a WHERE clause: as follows:

```
SELECT sys_xmlgen(x) from table_name WHERE x is NOT NULL;
```

Example 15–26 SYS_XMLGEN(): Retrieving Employee fname From Employees Table and Generating XML with FNAME Element

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

```
SELECT SYS_XMLGEN(fname).getStringVal()
   FROM employees
   WHERE employee_id = 1001;

SYS_XMLGEN(FNAME).GETSTRINGVAL()
----------------------------------------
<?xml version="1.0"?>
<FNAME>John</FNAME>
```

What are the Advantages of Using SYS_XMLGEN(?

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. For example, when you use an XMLType view that is a SYS_XMLGEN() on top of an object type, Oracle XML DB rewrites these queries when possible. See also Chapter 6, "XML Schema Storage and Query: Advanced Topics".

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 action 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. Oracle Database provides the `XMLFormat` object, which lets you format the output of the `SYS_XMLGEN` function.

Table 15–2 lists the `XMLFormat` attributes. of the `XMLFormat` object. The function that implements this type follows the table.

Table 15–2 Attributes of the XMLFormat Object

Attribute	Datatype	Purpose
`enclTag`	`VARCHAR2(100)`	The name of the enclosing tag for the result of the `SYS_XMLGEN` function. If the input to the function is a column name, then the default is the column name. Otherwise the default is `ROW`. When `schemaType` is set to `USE_GIVEN_SCHEMA`, this attribute also gives the name of the XMLSchema element.
`schemaType`	`VARCHAR2(100)`	The type of schema generation for the output document. Valid values are '`NO_SCHEMA`' and '`USE_GIVEN_SCHEMA`'. The default is '`NO_SCHEMA`'.
`schemaName`	`VARCHAR2(4000)`	The name of the target schema Oracle Database uses if the value of the `schemaType` is '`USE_GIVEN_SCHEMA`'. If you specify `schemaName`, then Oracle Database uses the enclosing tag as the element name.
`targetNameSpace`	`VARCHAR2(4000)`	The target namespace if the schema is specified (that is, `schemaType` is `GEN_SCHEMA_*`, or `USE_GIVEN_SCHEMA`)
`dburl`	`VARCHAR2(2000)`	The URL to the database to use if `WITH_SCHEMA` is specified. If this attribute is not specified, then Oracle Database declares the URL to the types as a relative URL reference.
`processingIns`	`VARCHAR2(4000)`	User-provided processing instructions, which are appended to the top of the function output before the element.

Example 15–27 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 either name.

Example 15–28 SYS_XMLGEN(): Converting a Scalar Value to an XML Document Element Contents

When you enter 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:

```
SYS_XMLGEN(EMPNO)
-------------------------
<?xml version="1.0"?>
<EMPNO>7369</EMPNO>

1 row selected.
```

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 15–29 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:

```
SYS_XMLGEN(EMPNO).GETCLOBVAL()
--------------------------------------
<?xml version="1.0"?>
<EMPNO>7369</EMPNO>
```

because 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 15–30 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, then you can supply a formatting argument to the function, as follows:

```
SELECT SYS_XMLGEN(empno *2,
                  xmlformat.createformat('EMPNO')).getClobVal()
FROM scott.emp
WHERE ROWNUM < 2;
```

This results in the following XML:

```
SYS_XMLGEN(EMPNO*2,XMLFORMAT.CREATEFORMAT('EMPNO')).GETCLOBVAL()
--------------------------------------------------------------------
<?xml version="1.0"?>
<EMPNO>14738</EMPNO>

1 row selected.
```

Example 15–31 SYS_XMLGEN(): Converting a User-Defined Type to XML

When you enter 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 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, you can generate a hierarchical XML for the employee, department example as follows:

```
CREATE OR REPLACE TYPE scott.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 scott.emplist_t AS TABLE OF emp_t;
/
CREATE OR REPLACE TYPE scott.dept_t AS OBJECT(deptno  NUMBER(2),
                                              dname   VARCHAR2(14),
                                              loc     VARCHAR2(13),
                                              emplist emplist_t);
/
SELECT SYS_XMLGEN(dept_t(deptno, dname, d.loc,
                         CAST(MULTISET(SELECT emp_t(e.empno, e.ename, e.job,
                                                    e.mgr, e.hiredate, e.sal,
                                                    e.comm)
                                       FROM scott.emp e
                                       WHERE e.deptno = d.deptno)
                              AS emplist_t))).getClobVal()
AS deptxml
FROM scott.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, assigns 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:

```
DEPTXML
------------------
<?xml version="1.0"?>
<ROW>
  <DEPTNO>10</DEPTNO>
  <DNAME>Accounting</DNAME>
  <LOC>NEW YORK</LOC>
  <EMPLIST>
```

```
    <EMP_T>
      <EMPNO>7782</EMPNO>
      <ENAME>CLARK</ENAME>
      <JOB>MANAGER</JOB>
      <MGR>7839</MGR>
      <HIREDATE>09-JUN-81</HIREDATE>
      <SAL>2450</SAL>
    </EMP_T>
    <EMP_T>
      <EMPNO>7839</EMPNO>
      <ENAME>KING</ENAME>
      <JOB>PRESIDENT</JOB>
      <HIREDATE>17-NOV-81</HIREDATE>
      <SAL>5000</SAL>
    </EMP_T>
  </EMPLIST>
</ROW>
<?xml version="1.0"?>
<ROW>
  <DEPTNO>20</DEPTNO>
  <DNAME>RESEARCH</DNAME>
  <LOC>DALLAS</LOC>
  <EMPLIST>
    <EMP_T>
      <EMPNO>7369</EMPNO>
      <ENAME>SMITH</ENAME>
      <JOB>CLERK</JOB>
      <MGR>7902</MGR>
      <HIREDATE>17-DEC-80</HIREDATE>
      <SAL>800</SAL>
    </EMP_T>

    <EMP_T>
      <EMPNO>7566</EMPNO>
      <ENAME>JONES</ENAME>
      <JOB>MANAGER</JOB>
      <MGR>7839</MGR>
      <HIREDATE>02-APR-87</HIREDATE>
      <SAL>3000</SAL>
    </EMP_T>
    <EMP_T>
      <EMPNO>7788</EMPNO>
      <ENAME>SCOTT</ENAME>
      <JOB>ANALYST</JOB>
      <MGR>7566</MGR>
      <HIREDATE>19-APR-87</HIREDATE>
      <SAL>3000</SAL>
    </EMP_T>
    <EMP_T>
      <EMPNO>7876</EMPNO>
      <ENAME>ADAMS</ENAME>
      <JOB>CLERK</JOB>
      <MGR>7788</MGR>
      <HIREDATE>23-MAY-87</HIREDATE>
      <SAL>1100</SAL>
    </EMP_T>
    <EMP_T>
      <EMPNO>7902</EMPNO>
      <ENAME>FORD</ENAME>
      <JOB>ANALYST</JOB>
```

```
      <MGR>7566</MGR>
      <HIREDATE>03-DEC-81</HIREDATE>
      <SAL>3000</SAL>
    </EMP_T>
</EMPLIST>
</ROW>
<?xml version="1.0"?>
<ROW>
  <DEPTNO>30</DEPTNO>
  <DNAME>SALES</DNAME>
  <LOC>CHICAGO</LOC>
  <EMPLIST>
    <EMP_T>
      <EMPNO>7499</EMPNO>
      <ENAME>ALLEN</ENAME>
      <JOB>SALESMAN</JOB>
      <MGR>7698</MGR>
      <HIREDATE>20-FEB-81</HIREDATE>
      <SAL>1600</SAL>
      <COMM>300</COMM>
    </EMP_T>

    <EMP_T>
      <EMPNO>7521</EMPNO>
      <ENAME>WARD</ENAME>
      <JOB>SALESMAN</JOB>
      <MGR>7698</MGR>
      <HIREDATE>22-FEB-81</HIREDATE>
      <SAL>1250</SAL>
      <COMM>500</COMM>
    </EMP_T>
    <EMP_T>
      <EMPNO>7654</EMPNO>
      <ENAME>MARTIN</ENAME>
      <JOB>SALESMAN</JOB>
      <MGR>7698</MGR>
      <HIREDATE>28-SEP-81</HIREDATE>
      <SAL>1250</SAL>
      <COMM>1400</COMM>
    </EMP_T>
    <EMP_T>
      <EMPNO>7698</EMPNO>
      <ENAME>BLAKE</ENAME>
      <JOB>MANAGER</JOB>
      <MGR>7839</MGR>
      <HIREDATE>01-MAY-81</HIREDATE>
      <SAL>2850</SAL>
    </EMP_T>
    <EMP_T>
      <EMPNO>7844</EMPNO>
      <ENAME>TURNER</ENAME>
      <JOB>SALESMAN</JOB>
      <MGR>7698</MGR>
      <HIREDATE>08-SEP-81</HIREDATE>
      <SAL>1500</SAL>
      <COMM>0</COMM>
    </EMP_T>
    <EMP_T>
      <EMPNO>7900</EMPNO>
      <ENAME>JAMES</ENAME>
```

```
      <JOB>CLERK</JOB>
      <MGR>7698</MGR>
      <HIREDATE>03-DEC-81</HIREDATE>
      <SAL>950</SAL>
    </EMP_T>
  </EMPLIST>
</ROW>

<?xml version="1.0"?>
<ROW>
 <DEPTNO>40</DEPTNO>
 <DNAME>OPERATIONS</DNAME>
 <LOC>BOSTON</LOC>
 <EMPLIST/>
</ROW>

4 rows selected.
```

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 15–32 SYS_XMLGEN(): Converting an XMLType Instance

If you pass an XML document into `SYS_XMLGEN()`, then `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;
```

then 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 15-44.

Example 15–33 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(MULTISET(
                    SELECT
                      LineItem_typ(L.LineItemNo,
                                   StockItem_typ(L.StockNo,S.Price,S.TaxRate),
                                                 L.Quantity, L.Discount)
                      FROM LineItems_tab L, Stock_tab S
                      WHERE L.PONo = P.PONo AND S.StockNo=L.StockNo)
                  AS LineItems_ntabtyp),
             Address_typ(P.ToStreet, P.ToCity, P.ToState, P.ToZip)
        FROM PO_tab P, Customer C
        WHERE P.CustNo=C.custNo;

-----------------------------------------------------
-- Use SYS_XMLGEN() to generate PO in XML format
-----------------------------------------------------
SET LONG 20000
SET PAGES 100
SELECT
```

```
  SYS_XMLGEN(value(p),
             sys.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>
    <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, use the 'fmt' parameter.

Figure 15–10 SYS_XMLAGG() Syntax

See Also: *Oracle Database SQL Reference*

Generating XML Using XSQL Pages Publishing Framework

Oracle9*i* 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 15–34 Using XSQL Servlet <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 15–35 Using XSQL Servlet <xsql:include-xml>, XMLElement(), and XMLAgg() to Generate XML from Oracle Database

Because 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`, then 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 Oracle Database 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 in your XSQL page,
-- 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 recommendation. 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:** *Oracle XML Developer's Kit Programmer's Guide*, the chapter, 'XSQL Page Publishing Framework'

Using XSLT and XSQL

With XSQL Pages, you have control of whether XSLT is executed by the database, the middle-tier, or the client. For the database option, use the `XMLTransform()` operator (or equivalent) technique in your query. For the middle-tier option, add the `<?xml-stylesheet?>` line at the top of your template page. For the client option, just add the client="yes" attribute to your `<?xml-stylesheet?>` line.

With XSQL Pages, you can even build pages that conditionally off-load the style-sheet processing to the client (for example, if you detect the requesting user agent is Internet Explorer 6.0), while other browsers will get the middle-tier fallback transform behavior.

XSQL caches and pools XSLT style sheets in the middle tier (as well as database connections) to improve performance and throughput. Depending on the application you can further improve performance by avoiding transformation using Web Cache or other techniques as well as a further performance optimization to avoid retransforming the same (or static) data over and over.

Also, XSQL Pages can include a mix of static XML and dynamically produced XML, so it already gives you some flexibility to only make the dynamic part of the page hit the database.

Generating XML Using XML SQL Utility (XSU)

Oracle XML SQL Utility (XSU) can still be used with Oracle Database 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 15–36 Generating XML Using XSU for Java getXML

For example, if you have table, parts:

```
CREATE TABLE parts (PartNo NUMBER, PartName VARCHAR2(20), PartDesc XMLType);
```

then 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 : *Oracle XML Developer's Kit Programmer's Guide* for more information on XSU

Guidelines for Generating XML With Oracle XML DB

This section describes additional guidelines for generating XML using Oracle XML DB.

Using XMLAgg ORDER BY Clause to Order Query Results Before Aggregation

To use the `XMLAgg ORDER BY` clause before aggregation, specify the ORDER BY clause following the first `XMLAGG` argument.

Example 15–37 Using XMLAgg ORDER BY Clause

For example, in the following expression the result is aggregated according to the order of the `dev` field:

```
SELECT XMLAgg(XMLElement("Dev",
                          XMLAttributes(dev AS "id",
                                        dev_total AS "total"),
                         devname AS "name")
              ORDER BY dev)
  FROM tab1 dev_total;
```

Using XMLSequence in the TABLE Clause to Return a Rowset

To use `XMLSequence()` with `extract()` to return a rowset with relevant portions of a document extracted as multiple rows, use `XMLSequence()` in the `TABLE()` clause as shown in the following example.

Example 15–38 Returning a Rowset using XMLSequence in the TABLE Clause

```
SELECT extractValue(value(t), '/*/Last') as LAST,
       extractValue(value(t), '/*/First') as FIRST
   FROM movies m,
        table(xmlsequence(extract(value(m),
                                  '/Movie/Cast/Actor | '/Movie/Cast/Actress)))
   ORDER BY LAST;
```

This returns a rowset with just the first and last names of the actors and actresses, ordered by last name.

16

XMLType Views

This chapter describes how to create and use `XMLType` views.

This chapter contains these topics:

What Are XMLType Views?

`XMLType` views wrap existing relational and object-relational data in XML formats. The major advantages of using `XMLType` views are:

- You can exploit Oracle XML DB XML features that use XML schema functionality without having to migrate your base legacy data.
- With `XMLType` views, you can experiment with various other forms of storage, besides the object-relational 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()` with `getNumberVal()` on the `XMLType` value . Oracle recommends that you use the `extract()` operator rather than the member function in the `OBJECT IDENTIFIER` clause.

Throughout this chapter XML schema refers to the W3C XML Schema 1.0 recommendation, `http://www.w3.org/xml/Schema`.

There are two types of XMLType views:

- **Non-schema-based XMLType views.** These views do not confirm to a particular XML schema.

- **XML schema-based XMLType views.** As with `XMLType` tables, `XMLType` views that conform to a particular XML schema are called XML schema-based `XMLType` views. These provide stronger typing than non-schema-based `XMLType` views.

Optimization of queries over `XMLType` views are enabled for both XML schema-based and non-schema-based `XMLType` views. This is known as XPath rewrite and is described in the section, "XPath Rewrite on XMLType Views" on page 16-19.

To create an XML schema-based `XMLType` view, first register your XML schema. If the XML schema-based `XMLType` view is constructed using an object type -- object view, then the XML schema should have annotations that represent the bi-directional mapping from XML to SQL object types. `XMLType` views conforming to this registered XML schema can then be created by providing an underlying query that constructs instances of the appropriate SQL object type.

See Also:

- "Relational Access to XML Content Stored in Oracle XML DB Using Views" on page 3-55
- Chapter 5, "XML Schema Storage and Query: The Basics"
- Appendix B, "XML Schema Primer"

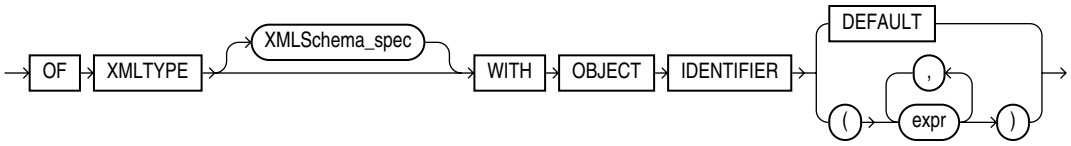
`XMLType` views can be constructed in the following ways:

- Based on SQL/XML generation functions, such as `XMLElement()`, `XMLForest()`, `XMLConcat()`, `XMLAgg()` and Oracle Database extension function `XMLColAttVal()`. SQL/XML generation functions can be used to construct both non-schema-based `XMLType` views and XML schema-based `XMLType` views. This enables construction of `XMLType` view from the underlying relational tables directly without physically migrating those relational legacy data into XML. However, to construct XML schema-based `XMLType` view, the XML schema must be registered and the XML value generated by SQL/XML functions must be constrained to the XML schema.
- Based on object types, object views and the `SYS_XMLGEN()` function. Non-schema-based `XMLType` views can be constructed using object types, object views, and `SYS_XMLGEN()` function and XML schema-based `XMLType` view can be constructed using object types and object views. This enables the construction of the `XMLType` view from underlying relational or object relational tables directly without physically migrating the relational or object relational legacy data into XML. Creating non-schema-based `XMLType` view requires the use of `SYS_XMLGEN()` function over existing object types or object views. Creating XML-schema-based `XMLType` view requires to annotate the XML schema with a mapping to existing object types or to generate the XML schema from the existing object types.
- XML schema-based `XMLType` views can also be constructed directly from an `XMLType` table.

Creating XMLType Views: Syntax

Figure 16–1 shows the `CREATE VIEW` clause for creating `XMLType` views. See *Oracle Database SQL Reference* for details on the `CREATE VIEW` syntax.

Figure 16–1 Creating XMLType Views Clause: Syntax



Creating Non-Schema-Based XMLType Views

Non-schema-based `XMLType` views are `XMLType` views whose resultant XML value is not constrained to be a particular element in a registered XML schema. There are two main ways to create non-schema-based `XMLType` views:

- Using SQL/XML generation functions, such as `XMLElement`, `XMLForest`, `XMLConcat`, `XMLAgg`, and `XMLColAttVal`. Here you create the `XMLType` view using simple SQL/XML generation functions, without creating object types. Creating `XMLType` views using SQL/XML functions is simple as you do not have to create object types or object views.

  See Also: Chapter 15, "Generating XML Data from the Database", for details on SQL/XML generation functions

- Using object types and object views with `SYS_XMLGEN()` function. Here you create the `XMLType` view using object types with `SYS_XMLGEN()` function. This method for creating `XMLType` views is convenient when you already have an object-relational schema, such as object types, views, and tables, and want to map it directly to XML without the overhead of creating XML schema.

  See Also: "Using Object Types and Views" on page 16-10

Using SQL/XML Generation Functions

Example 16–1 illustrates how to create an `XMLType` view using the SQL/XML function `XMLElement()`.

Example 16–1 Creating an XMLType View Using the XMLElement() Function

The following statement creates an `XMLType` view using the `XMLElement()` generation function:

```
CREATE OR REPLACE VIEW Emp_view OF XMLType WITH OBJECT ID
     (EXTRACT(OBJECT_VALUE,'/Emp/@empno').getnumberval())
       AS SELECT XMLELEMENT("Emp", XMLAttributes(employee_id),
            XMLForest(e.first_name ||' '|| e.last_name AS "name",
                        e.hire_date AS "hiredate")) AS "result"
          FROM employees e
          WHERE salary > 15000;

SELECT * FROM Emp_view;

SYS_NC_ROWINFO$

--------------------------------------------------------------------------------
<Emp EMPLOYEE_ID="100"><name>Steven King</name><hiredate>1987-06-17</hiredate></Emp>
<Emp EMPLOYEE_ID="101"><name>Neena Kochhar</name><hiredate>1989-09-21</hiredate></Emp>
<Emp EMPLOYEE_ID="102"><name>Lex De Haan</name><hiredate>1993-01-13</hiredate></Emp>
```

The empno attribute in the document will be used as the unique identifier for each row. As the result of the XPath rewrite, `/Emp/@empno` can refer directly to the empno column. An attribute is a property of an element that consists of a name and value separated by an equals sign and contained within the start-tags after the element name. In `<Price units='USD'>5</Price>`, units is the attribute and USD is its value, which must be in single or double quotes.

Elements may have many attributes but their retrieval order is not defined.

Existing data in relational tables or views can be exposed as XML using this mechanism. In addition, queries using `extract()`, `extractValue()` and `existsNode()` involving simple XPath traversal over views generated by SQL/XML generation functions are candidates for XPath rewrite to directly access the underlying relational columns or expressions based on those relational columns. See "XPath Rewrite on XMLType Views" on page 16-19 for details.

You can perform DML operations on these `XMLType` views, but, in general, you must write instead-of triggers to handle the DML operation.

Using Object Types with SYS_XMLGEN()

You can also create `XMLType` views using `SYS_XMLGEN()` with object types. `SYS_XMLGEN` inputs object type and generates an `XMLType`. Here is an equivalent query that produces the same query results using `SYS_XMLGEN`:

Example 16–2 Creating an XMLType View Using Object Types and SYS_XMLGEN()

```
CREATE TYPE Emp_t AS OBJECT
    ("@empno"   number(6),
      fname     varchar2(20),
      lname     varchar2(25),
      hiredate  date);
/

CREATE OR REPLACE VIEW employee_view OF XMLType WITH OBJECT ID
 (EXTRACT(OBJECT_VALUE,'/Emp/@empno').getnumberval())
 AS SELECT SYS_XMLGEN(emp_t(e.employee_id, e.first_name, e.last_name,
 e.hire_date),
 XMLFORMAT('EMP'))
 FROM employees e
 WHERE salary > 15000;

SELECT * FROM employee_view;

SYS_NC_ROWINFO$
------------------------------------------------------------
<?xml version="1.0"?
<EMP empno="100">
 <FNAME>Steven</FNAME>
 <LNAME>King</LNAME>
 <HIREDATE>1987-06-17</HIREDATE>
</EMP>

<?xml version="1.0"?>
<EMP empno="101">
 <FNAME>Neena</FNAME>
 <LNAME>Kochhar</LNAME>
 <HIREDATE>1989-09-21</HIREDATE>
</EMP>
```

```
<?xml version="1.0"?>
<EMP empno="102">
 <FNAME>Lex</FNAME>
 <LNAME>De Haan</LNAME>
 <HIREDATE>1993-01-13</HIREDATE>
</EMP>
```

Existing data in relational or object-relational tables or views can be exposed as XML using this mechanism. In addition, queries using `extract()`, `extractValue()` and `existsNode()` operators that involve simple XPath traversal over views generated by `SYS_XMLGEN()`, are candidates for XPath rewrite. XPath rewrite facilitates direct access to underlying object attributes or relational columns.

Creating XML Schema-Based XMLType Views

XML schema-based `XMLType` views are `XMLType` views whose resultant XML value is constrained to be a particular element in a registered XML schema. There are two main ways to create XML schema-based `XMLType` views:

- Using SQL/XML generation functions, such as `XMLElement`, `XMLForest`, `XMLConcat`, `XMLAgg` and `XMLColAttVal`: Here you create the `XMLType` view using simple XML generation functions, without needing to create any object types. This mechanism is simple as you do not have to create any object types or object views.

  See Also: "Using SQL/XML Generation Functions" on page 16-5

- Using object types and or object views. Here you create the `XMLType` view either using object types or from object views. This mechanism for creating `XMLType` views is convenient when you already have an object-relational schema and want to map it directly to XML.

  See Also: "Using Object Types and Views" on page 16-10

Using SQL/XML Generation Functions

You can use SQL/XML generation functions to create XML schema-based `XMLType` views in a similar way as for the non-schema based case described in section "Creating Non-Schema-Based XMLType Views". To create XML schema-based `XMLType` views perform these steps:

1. Create and register the XML schema document that contains the necessary XML structures. Note that since the `XMLType` view is constructed using SQL/XML generation functions, you do not need to annotate the XML schema to present the bidirectional mapping from XML to SQL object types.
2. Create an `XMLType` view conforming to the XML schema by using SQL/XML functions.

Step 1. Register XML Schema, emp_simple.xsd

Assume that you have an XML schema `emp_simple.xsd` that contains XML structures defining an employee. First register the XML schema and identify it using a URL:

```
BEGIN
 dbms_xmlschema.registerSchema('http://www.oracle.com/emp_simple.xsd',
   '<schema xmlns="http://www.w3.org/2001/XMLSchema"
```

```
      targetNamespace="http://www.oracle.com/emp_simple.xsd" version="1.0"
      xmlns:xdb="http://xmlns.oracle.com/xdb"
      elementFormDefault="qualified">
 <element name = "Employee">
  <complexType>
   <sequence>
    <element name = "EmployeeId" type = "positiveInteger"/>
    <element name = "Name" type = "string"/>
    <element name = "Job" type = "string"/>
    <element name = "Manager" type = "positiveInteger"/>
    <element name = "HireDate" type = "date"/>
    <element name = "Salary" type = "positiveInteger"/>
    <element name = "Commission" type = "positiveInteger"/>
    <element name = "Dept">
       <complexType>
          <sequence>
            <element name = "DeptNo" type = "positiveInteger" />
            <element name = "DeptName" type = "string"/>
            <element name = "Location" type = "positiveInteger"/>
          </sequence>
       </complexType>
    </element>
   </sequence>
  </complexType>
 </element>
</schema>', TRUE, TRUE, FALSE);
END;
```

This registers the XML schema with the target location:

```
http://www.oracle.com/emp_simple.xsd
```

Step 2. Create XMLType View Using SQL/XML Functions

You can create an XML schema-based `XMLType` view using SQL/XML functions. The resultant XML must conform to the XML schema specified for the view.

When using SQL/XML functions to generate XML schema-based content, you must specify the appropriate namespace information for all the elements and also indicate the location of the schema using the `xsi:schemaLocation` attribute. These can be specified using the `XMLAttributes` clause.

```
CREATE OR REPLACE VIEW emp_simple_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/emp_simple.xsd" ELEMENT "Employee"
   WITH OBJECT ID (extract(object_value,
                           '/Employee/EmployeeId/text()').getnumberval()) AS
   SELECT XMLElement("Employee",
     XMLAttributes( 'http://www.oracle.com/emp_simple.xsd' AS "xmlns" ,
              'http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
              'http://www.oracle.com/emp_simple.xsd
               http://www.oracle.com/emp_simple.xsd' AS "xsi:schemaLocation"),
      XMLForest(e.employee_id     AS "EmployeeId",
                e.last_name       AS "Name",
                e.job_id          AS "Job",
                e.manager_id      AS "Manager",
                e.hire_date       AS "HireDate",
                e.salary          AS "Salary",
                e.commission_pct  AS "Commission",
                XMLForest(d.department_id     AS "DeptNo",
                          d.department_name  AS "DeptName",
                          d.location_id      AS "Location") AS "Dept"))
```

```
   FROM employees e, departments d
   WHERE e.department_id = d.department_id;
```

In the preceding example, `XMLElement()` created the `Employee` XML element and the inner `XMLForest()` function created the children of the `Employee` element. The `XMLAttributes` clause inside `XMLElement()` constructed the required XML `namespace` and schema `location` attributes so that the XML generated conforms to the XML schema of the view. The innermost `XMLForest()` function created the `department` XML element that is nested inside the `Employee` element.

The XML generation function normally generates a non-XML schema-based XML instance. However, when the schema information is specified using attributes `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation`, Oracle XML DB generates schema-based XML. In the case of `XMLType` views, as long as the names of the elements and attributes match those in the XML schema, Oracle Database converts the XML implicitly into a well-formed and valid XML schema-based document. Any errors in the generated XML are caught when further operations, such as validate or extract, are performed on the XML instance.

You can now query the view and get the XML result from the `employees` and `departments` relational tables with `NLS_DATE_FORMAT` setting to `'SYYYY-MM-DD'`:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT='SYYYY-MM-DD';
SQL> SELECT value(p) AS result FROM emp_simple_xml p WHERE rownum < 2;

RESULT
--------------------------------------------------------------------
Employee xmlns="http://www.oracle.com/emp_simple.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.
oracle.com/emp_simple.xsd
 http://www.oracle.com/emp_simple.xsd">
<EmployeeId>100</EmployeeId><Name>King</Name>
<Job>AD_PRES</Job><HireDate>1987-06-17</Hi
reDate><Salary>24000</Salary><Dept><DeptNo>90</DeptNo>
<DeptName>Executive</DeptName><Location>1700</Location></Dept></Employee>
```

Using Namespaces With SQL/XML Functions

If you have complicated XML schemas involving namespaces, you must use the partially escaped mapping provided in the SQL/XML functions and create elements with appropriate namespaces and prefixes.

Example 16–3 Using Namespace Prefixes in XMLType Views

```
SELECT  XMLElement("ipo:Employee",
          XMLAttributes('http://www.oracle.com/emp_simple.xsd' AS "xmlns:ipo",
                        'http://www.oracle.com/emp_simple.xsd
                         http://www.oracle.com/emp_simple.xsd' AS "xmlns:xsi"),
            XMLForest(e.employee_id AS "ipo:EmployeeId",
                      e.last_name   AS "ipo:Name",
                      e.job_id      AS "ipo:Job",
                      e.manager_id  AS "ipo:Manager",
                      TO_CHAR(e.hire_date,'YYYY-MM-DD') AS "ipo:HireDate",
                      e.salary  AS "ipo:Salary",
                      e.commission_pct AS "ipo:Commission",
                   XMLForest(d.department_id   AS "ipo:DeptNo",
                             d.department_name AS "ipo:DeptName", d.location_id
       AS "ipo:Location") AS "ipo:Dept"))
```

```
       FROM employees e, departments d
       WHERE e.department_id = d.department_id
             AND d.department_id = 20;

BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('emp-noname.xsd', 4);
END;
```

This SQL query creates the XML instances with the correct namespace, prefixes, and target schema location, and can be used as the query in the `emp_simple_xml` view definition. The instance created by this query looks like the following:

```
result
----------
<ipo:Employee
xmlns:ipo="http://www.oracle.com/emp_simple.xsd"
 xmlns:xsi="http://www.oracle.com/emp_simple.xsd
 http://www.oracle.com/emp_simple.xsd">
<ipo:EmployeeId>201</ipo:EmployeeId><ipo:Name>Hartstein</ipo:Name>
<ipo:Job>MK_MAN</ipo:Job><ipo:Manager>100</ipo:Manager>
<ipo:HireDate>1996-02-17</ipo:HireDate><ipo:Salary>13000</ipo:Salary>
<ipo:Dept><ipo:DeptNo>20</ipo:DeptNo><ipo:DeptName>Marketing</ipo:DeptName>
<ipo:Location>1800</ipo:Location></ipo:Dept></ipo:Employee>
<ipo:Employee xmlns:ipo="http://www.oracle.com/emp_simple.xsd"
 xmlns:xsi="http://www.oracle.com/emp_simple.xsd
 http://www.oracle.com/emp_simple.xsd"><ipo:EmployeeId>202</ipo:EmployeeId>
<ipo:Name>Fay</ipo:Name><ipo:Job>MK_REP</ipo:Job><ipo:Manager>201</ipo:Manager>
<ipo:HireDate>1997-08-17</ipo:HireDate><ipo:Salary>6000</ipo:Salary>
<ipo:Dept><ipo:DeptNo>20</ipo:Dept
No><ipo:DeptName>Marketing</ipo:DeptName><ipo:Location>1800</ipo:Location>
</ipo:Dept>
</ipo:Employee>
```

If the XML schema had no target namespace, then you 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.registerSchema('emp-noname.xsd',
   '<schema xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:xdb="http://xmlns.oracle.com/xdb"
       >
 <element name = "Employee">
  <complexType>
   <sequence>
    <element name = "EmployeeId" type = "positiveInteger"/>
    <element name = "Name" type = "string"/>
    <element name = "Job" type = "string"/>
    <element name = "Manager" type = "positiveInteger"/>
    <element name = "HireDate" type = "date"/>
    <element name = "Salary" type = "positiveInteger"/>
    <element name = "Commission" type = "positiveInteger"/>
    <element name = "Dept">
       <complexType>
          <sequence>
            <element name = "DeptNo" type = "positiveInteger" />
            <element name = "DeptName" type = "string"/>
            <element name = "Location" type = "positiveInteger"/>
          </sequence>
       </complexType>
```

```
    </element>
   </sequence>
  </complexType>
 </element>
</schema>', TRUE, TRUE, FALSE);
END;
```

The following 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(object_value,
                             '/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.employee_id    AS "EmployeeId",
                e.last_name      AS "Name",
                e.job_id         AS "Job",
                e.manager_id     AS "Manager",
                e.hire_date      AS "HireDate",
                e.salary         AS "Salary",
                e.commission_pct AS "Commission",
                XMLForest(d.department_id     AS "DeptNo",
                          d.department_name  AS "DeptName",
                          d.location_id      AS "Location") AS "Dept"))
   FROM employees e, departments d
   WHERE e.department_id = d.department_id;
```

The `XMLAttributes` clause creates an XML element that contains the `noNamespace` schema location attribute.

Example 16–4 Using SQL/XML Generation Functions in Schema-Based XMLType Views

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/dept.xsd', 4);
END;
/

BEGIN
dbms_xmlschema.registerSchema('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">
    <complexType>
      <sequence>
        <element name = "DeptNo" type = "positiveInteger"/>
        <element name = "DeptName" type = "string"/>
        <element name = "Location" type = "positiveInteger"/>
        <element name = "Employee" maxOccurs = "unbounded">
          <complexType>
           <sequence>
             <element name = "EmployeeId" type = "positiveInteger"/>
             <element name = "Name" type = "string"/>
             <element name = "Job" type = "string"/>
             <element name = "Manager" type = "positiveInteger"/>
             <element name = "HireDate" type = "date"/>
             <element name = "Salary" type = "positiveInteger"/>
```

```
            <element name = "Commission" type = "positiveInteger"/>
          </sequence>
         </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>', TRUE, FALSE, FALSE);
END;
/
CREATE OR REPLACE VIEW dept_xml OF XMLType
    XMLSCHEMA "http://www.oracle.com/dept.xsd" ELEMENT "Department"
 WITH OBJECT ID (EXTRACT(object_value, '/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(d.department_id "DeptNo",
                       d.department_name "DeptName",
                       d.location_id "Location"),
             (SELECT XMLAGG(XMLElement("Employee",
                              XMLForest(e.employee_id  "EmployeeId",
                                   e.last_name "Name",
                                   e.job_id "Job",
                                   e.manager_id "Manager",
               to_char(e.hire_date,'YYYY-MM-DD') "Hiredate",
                                   e.salary "Salary",
                                   e.commission_pct "Commission")))
                 FROM employees e
                 WHERE e.department_id = d.department_id))
     FROM departments d;
```

This SQL query creates the XML instances with the correct namespace, prefixes, and target schema location, and can be used as the query in the `emp_simple_xml` view definition. The instance created by this query looks like the following:

```
SELECT value(p) AS result FROM dept_xml p WHERE rownum < 2;

RESULT
--------------------------------------------------------------
<Department xmlns="http://www.oracle.com/emp.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.oracle.com/dept.xsd
 http://www.oracle.com/dept.xsd"><DeptNo>10</DeptNo>
<DeptName>Administration</DeptName><Location>1700</Location>
<Employee><EmployeeId>200</EmployeeId>
<Name>Whalen</Name><Job>AD_ASST</Job>
<Manager>101</Manager><Hiredate>1987-09-17</Hiredate>
<Salary>4400</Salary></Employee></Department>
```

Using Object Types and Views

To wrap relational or object-relational data with strongly-typed XML using the object view approach, 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, "XML Schema Storage and Query: The Basics".The XML schema can be

generated from the existing object types and must be annotated to contain the bidirectional mapping from XML to the object types.

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, "XML Schema Storage and Query: The Basics" 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 a VARCHAR2 or Large Object (LOB).
- ***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.

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 parts:
   a. Create an object view.
   b. Create an `XMLType` view over the object view.

Consider the following examples based on the employee -department relational tables and XML views of this data:

- Example 16–5, "Creating Schema-Based XMLType Views Over Object Views"
- Example 16–6, "XMLType View: View 2, Wrapping Relational Department Data with Nested Employee Data as XML"

Example 16–5 Creating Schema-Based XMLType Views Over Object 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 TYPE dept_t AS OBJECT
      (deptno        NUMBER(4),
       dname         VARCHAR2(30),
       loc           NUMBER(4));
/

CREATE TYPE emp_t AS OBJECT
      (empno         NUMBER(6),
       ename         VARCHAR2(25),
       job           VARCHAR2(10),
       mgr           NUMBER(6),
       hiredate      DATE,
```

```
       sal           NUMBER(8,2),
       comm          NUMBER(2,2),
       dept          dept_t );
/
```

Step 2. Create or Generate XMLSchema, emp.xsd

You can either create the XML schema manually or use the `DBMS_XMLSchema` package to generate the XML schema automatically from the existing object types as follows:

```
SELECT DBMS_XMLSchema.generateSchema('HR','EMP_T') AS result FROM DUAL;
```

This generates the XML schema for the `employee` type. You can supply various arguments to this function to add namespaces, and so on. You can also edit the XML schema to change the various default mappings that were generated. The `generateSchemas()` function in the DBMS_XMLSchema package generates a list of XML schemas one for each SQL database schema referenced by the object type and its attributes, embedded at any level.

Step 3. Register XML Schema, emp_complex.xsd

XML schema, `emp_complex.xsd` also specifies how the XML elements and attributes are mapped to their corresponding attributes in the object types, as follows. See also the `xdb:SQLType` annotation in the following example:

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/emp_complex.xsd', 4);
END;
/

COMMIT;

BEGIN
  DBMS_XMLSCHEMA.registerSchema('http://www.oracle.com/emp_complex.xsd',
    '<?xml version="1.0"?>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:xdb="http://xmlns.oracle.com/xdb"
               xsi:schemaLocation="http://xmlns.oracle.com/xdb
                                   http://xmlns.oracle.com/xdb/XDBSchema.xsd">
      <xsd:element name="Employee" type="EMP_TType" xdb:SQLType="EMP_T"
                   xdb:SQLSchema="HR"/>
      <xsd:complexType name="EMP_TType" xdb:SQLType="EMP_T" xdb:SQLSchema="HR"
                       xdb:maintainDOM="false">
        <xsd:sequence>
          <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="EMPNO"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="ENAME" xdb:SQLName="ENAME" xdb:SQLType="VARCHAR2">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="25"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="JOB" xdb:SQLName="JOB" xdb:SQLType="VARCHAR2">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="10"/>
              </xsd:restriction>
            </xsd:simpleType>
```

```
          </xsd:element>
          <xsd:element name="MGR" type="xsd:double" xdb:SQLName="MGR"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="HIREDATE" type="xsd:date" xdb:SQLName="HIREDATE"
                       xdb:SQLType="DATE"/>
          <xsd:element name="SAL" type="xsd:double" xdb:SQLName="SAL"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="COMM" type="xsd:double" xdb:SQLName="COMM"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="DEPT" type="DEPT_TType" xdb:SQLName="DEPT"
                       xdb:SQLSchema="HR" xdb:SQLType="DEPT_T"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="DEPT_TType" xdb:SQLType="DEPT_T" xdb:SQLSchema="HR"
                       xdb:maintainDOM="false">
        <xsd:sequence>
          <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="DNAME" xdb:SQLName="DNAME" xdb:SQLType="VARCHAR2">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="30"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
         <xsd:element name="LOC" type="xsd:double" xdb:SQLName="LOC"
                       xdb:SQLType="NUMBER"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>',
   TRUE,
   FALSE,
   FALSE);
END;
/
```

The preceding statement registers the XML schema with the target location:

```
"http://www.oracle.com/emp_complex.xsd"
```

Step 4a. 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_complex.xsd"
  ELEMENT "Employee"
    WITH OBJECT ID (EXTRACTVALUE(object_value, '/Employee/EMPNO')) AS
  SELECT emp_t(e.employee_id, e.last_name, e.job_id, e.manager_id, e.hire_date,
               e.salary, e.commission_pct,
               dept_t(d.department_id, d.department_name, d.location_id))
    FROM employees e, departments d
    WHERE e.department_id = d.department_id;
```

This example uses the `extractValue()` SQL function here in the `OBJECT ID` clause, because `extractValue()` can automatically calculate the appropriate SQL datatype mapping, in this case a SQL Number, using the XML schema information. Oracle Corporation recommends that you use the `extractValue()` operator rather than the `extractValue()` member function.

Step 4b. Using the Two-Step Process by First Creating an Object View

In the two-step process, first create an object view, then create an XMLType view on the object view, as follows:

```
CREATE OR REPLACE VIEW emp_v OF emp_t WITH OBJECT ID (empno) AS
  SELECT emp_t(e.employee_id, e.last_name, e.job_id, e.manager_id, e.hire_date,
               e.salary, e.commission_pct,
               dept_t(d.department_id, d.department_name, d.location_id))
    FROM employees e, departments d
    WHERE e.department_id = d.department_id;

CREATE OR REPLACE VIEW emp_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/emp_complex.xsd" ELEMENT "Employee"
  WITH OBJECT ID DEFAULT
  AS SELECT VALUE(p) FROM emp_v p;
```

Example 16–6 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

```
CREATE TYPE emp_t AS OBJECT (empno         NUMBER(6),
                             ename         VARCHAR2(25),
                             job           VARCHAR2(10),
                             mgr           NUMBER(6),
                             hiredate      DATE,
                             sal           NUMBER(8,2),
                             comm          NUMBER(2,2));
/

CREATE OR REPLACE TYPE emplist_t AS TABLE OF emp_t;
/

CREATE TYPE dept_t AS OBJECT (deptno        NUMBER(4),
                              dname         VARCHAR2(30),
                              loc           NUMBER(4),
                              emps          emplist_t);
/
```

Step 2. Register XML Schema, dept_complex.xsd

You can either use a pre-existing XML schema or you can generate an XML schema from the object type using the DBMS_XMLSCHEMA.generateSchema(s) functions:

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/dept_complex.xsd', 4);
END;
/

BEGIN
 DBMS_XMLSCHEMA.registerSchema('http://www.oracle.com/dept_complex.xsd',
   '<?xml version="1.0"?>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xdb="http://xmlns.oracle.com/xdb"
               xsi:schemaLocation="http://xmlns.oracle.com/xdb
                                   http://xmlns.oracle.com/xdb/XDBSchema.xsd">
```

```
      <xsd:element name="Department" type="DEPT_TType" xdb:SQLType="DEPT_T"
                   xdb:SQLSchema="HR"/>
      <xsd:complexType name="DEPT_TType" xdb:SQLType="DEPT_T" xdb:SQLSchema="HR"
                       xdb:maintainDOM="false">
        <xsd:sequence>
          <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="DNAME" xdb:SQLName="DNAME" xdb:SQLType="VARCHAR2">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="30"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="LOC" type="xsd:double" xdb:SQLName="LOC"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="EMPS" type="EMP_TType" maxOccurs="unbounded"
                       minOccurs="0" xdb:SQLName="EMPS"
                       xdb:SQLCollType="EMPLIST_T" xdb:SQLType="EMP_T"
                       xdb:SQLSchema="HR" xdb:SQLCollSchema="HR"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="EMP_TType" xdb:SQLType="EMP_T" xdb:SQLSchema="HR"
                       xdb:maintainDOM="false">
        <xsd:sequence>
          <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="EMPNO"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="ENAME" xdb:SQLName="ENAME" xdb:SQLType="VARCHAR2">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="25"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="JOB" xdb:SQLName="JOB" xdb:SQLType="VARCHAR2">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="10"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="MGR" type="xsd:double" xdb:SQLName="MGR"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="HIREDATE" type="xsd:date" xdb:SQLName="HIREDATE"
                       xdb:SQLType="DATE"/>
          <xsd:element name="SAL" type="xsd:double" xdb:SQLName="SAL"
                       xdb:SQLType="NUMBER"/>
          <xsd:element name="COMM" type="xsd:double" xdb:SQLName="COMM"
                       xdb:SQLType="NUMBER"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>',
  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_complex.xsd" ELEMENT "Department"
  WITH OBJECT ID (EXTRACTVALUE(object_value, '/Department/DEPTNO')) AS
  SELECT dept_t(d.department_id, d.department_name, d.location_id,
                CAST(MULTISET(SELECT emp_t(e.employee_id, e.last_name, e.job_id,
                                           e.manager_id, e.hire_date,
                                           e.salary, e.commission_pct)
                              FROM employees e
                              WHERE e.department_id = d.department_id)
                       AS emplist_t))
    FROM departments d;
```

Step 3b. Create XMLType Views Using SQL/XML Functions

You can also create the `dept_xml` `XMLType` view from the relational tables without using the object type definitions, that is using SQL/XML generation functions.

```
CREATE OR REPLACE VIEW dept_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/dept_complex.xsd" ELEMENT "Department"
  WITH OBJECT ID (EXTRACT(object_value, '/Department/DEPTNO').getNumberVal()) AS
  SELECT
    XMLElement(
      "Department",
      XMLAttributes('http://www.oracle.com/dept_complex.xsd' AS "xmlns",
                    'http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
                    'http://www.oracle.com/dept_complex.xsd
                     http://www.oracle.com/dept_complex.xsd'
                      AS "xsi:schemaLocation"),
      XMLForest(d.department_id "DeptNo", d.department_name "DeptName",
                d.location_id "Location"),
      (SELECT XMLAGG(XMLElement("Employee",
                                XMLForest(e.employee_id "EmployeeId",
                                          e.last_name "Name",
                                          e.job_id "Job",
                                          e.manager_id "Manager",
                                          e.hire_date "Hiredate",
                                          e.salary "Salary",
                                          e.commission_pct "Commission")))
                       FROM employees e WHERE e.department_id = d.department_id))
    FROM departments d;
```

Note: 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 From XMLType Tables

An `XMLType` view can be created on an `XMLType` table, for example, to transform the XML or to restrict the rows returned by using some predicates.

Example 16–7 Creating an XMLType View by Restricting Rows From an XMLType Table

Here is an example of creating an XMLType view by restricting the rows returned from an underlying XMLType table. This example uses the dept_complex.xsd XML schema, described in Example 16–6, to create the underlying table.

```
CREATE TABLE dept_xml_tab OF XMLType
    XMLSchema "http://www.oracle.com/dept_complex.xsd" ELEMENT "Department"
    nested table xmldata."EMPS" store as dept_xml_tab_tab1;

CREATE OR REPLACE VIEW dallas_dept_view OF XMLType
    XMLSchema "http://www.oracle.com/dept.xsd" ELEMENT "Department"
        AS SELECT value(p) FROM dept_xml_tab p
           WHERE extractValue(value(p), '/Department/Location') = 'DALLAS';
```

Here, dallas_dept_view restricts the XMLType table rows to those departments whose location is Dallas.

Example 16–8 Creating an XMLType View by Transforming an XMLType Table

You can create an XMLType view by transforming the XML data using a style sheet. For example, consider the creation of XMLType table po_tab. Refer to Example 8–1, "Transforming an XMLType Instance Using XMLTransform() and DBUriType to Get the XSL Style Sheet" on page 8-4 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_tab p;
```

XMLType view reference REF() is based on one of the following object IDs:

- System-generated OID — for views on XMLType tables or object views
- Primary key based OID -- for views with OBJECT ID expressions

These REFs can be used to fetch OCIXMLType instances in the OCI Object cache or can be used inside SQL queries. These REFs act 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 16–9 Identifying When a View is Implicitly Updatable

One way to identify when an XMLType view is implicitly updatable is to use an XMLType view query to determine if the view is based on an object view or an object constructor that is itself inherently updatable, as follows:

```
CREATE TYPE dept_t AS OBJECT
      (deptno        NUMBER(4),
       dname         VARCHAR2(30),
       loc           NUMBER(4));
/

BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/dept.xsd', 4);
END;
/

commit;

BEGIN
  DBMS_XMLSCHEMA.registerSchema('http://www.oracle.com/dept_t.xsd',
    '<?xml version="1.0"?>
     <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:xdb="http://xmlns.oracle.com/xdb"
                 xsi:schemaLocation="http://xmlns.oracle.com/xdb
                                     http://xmlns.oracle.com/xdb/XDBSchema.xsd">
     <xsd:element name="Department" type="DEPT_TType" xdb:SQLType="DEPT_T"
                    xdb:SQLSchema="HR"/>
     <xsd:complexType name="DEPT_TType" xdb:SQLType="DEPT_T" xdb:SQLSchema="HR"
                      xdb:maintainDOM="false">
         <xsd:sequence>
           <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"
                      xdb:SQLType="NUMBER"/>
           <xsd:element name="DNAME" xdb:SQLName="DNAME" xdb:SQLType="VARCHAR2">
             <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                 <xsd:maxLength value="30"/>
               </xsd:restriction>
             </xsd:simpleType>
           </xsd:element>
           <xsd:element name="LOC" type="xsd:double" xdb:SQLName="LOC"
                xdb:SQLType="NUMBER"/>
         </xsd:sequence>
       </xsd:complexType>
     </xsd:schema>',
    TRUE,
    FALSE,
    FALSE);
END;
/

CREATE OR REPLACE VIEW dept_xml of XMLType
  XMLSchema "http://www.oracle.com/dept_t.xsd" element "Department"
  WITH OBJECT ID (object_value.extract('/Department/DEPTNO').getnumberval()) AS
  SELECT dept_t(d.department_id, d.department_name, d.location_id)
    FROM departments d;

INSERT INTO dept_xml
  VALUES (
```

```
    XMLType.createXML(
      '<Department
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://www.oracle.com/dept_t.xsd" >
         <DEPTNO>300</DEPTNO>
         <DNAME>Processing</DNAME>
         <LOC>1700</LOC>
       </Department>'));

UPDATE dept_xml d
 SET d.object_value = updateXML(d.object_value, '/Department/DNAME/text()',
                                'Shipping')
   WHERE existsNode(d.object_value, '/Department[DEPTNO=300]') = 1;
```

XPath Rewrite on XMLType Views

XPath rewrites for `XMLType` views constructed using object types, object views, and `SYS_XMLGEN()` function or `XMLType` tables, are the same as that of regular `XMLType` table columns. Hence, `extract()`, `existsNode()`, or `extractValue()` operators on view columns get rewritten into underlying relational or object-relational accesses for better performance.

XPath rewrites for `XMLType` views constructed using the SQL/XML generation functions are also supported. Hence, `extract()`, `existsNode()`, or `extractValue()` operators on view columns get rewritten into underlying relational accesses for better performance.

XPath Rewrite on XMLType Views Constructed With SQL/XML Generation Functions

This section describes XML schema-based and non-schema-based XPath rewrites on `XMLType` views constructed with SQL/XML functions.

XPath Rewrite on Non-Schema-Based Views Constructed With SQL/XML

Example 16–10 illustrates XPath rewrites on non-schema-based XMLType views.

Example 16–10 Non-Schema-Based Views Constructed Using SQL/XML

```
CREATE OR REPLACE VIEW Emp_view OF XMLType  WITH OBJECT ID
     (EXTRACT(object_value,'/Emp/@empno').getnumberval())
       AS SELECT XMLELEMENT("Emp", XMLAttributes(employee_id),
            XMLForest(e.first_name ||' '|| e.last_name AS "name",
                        e.hire_date AS "hiredate")) AS "result"
          FROM employees e
          WHERE salary > 15000;
```

- Querying with `extractValue()` operator to select from `EMP_VIEW`

  ```
  SELECT EXTRACTVALUE(VALUE(e), '/Emp/name'),
         EXTRACTVALUE(VALUE(e), '/Emp/hiredate')FROM Emp_view e;
  ```

  becomes:

  ```
  SELECT e.first_name ||' '|| e.last_name, e.hire_date
      FROM employees e
    WHERE e.salary > 15000;
  ```

The rewritten query is a simple relational query. The `extractValue()` operator is rewritten down to the relational column access as defined in the `EMP_VIEW` view.

- Querying with `extractValue()` operator followed by `getNumberVal()` to select from `EMP_VIEW`

```
SELECT (EXTRACT(VALUE(e), '/Emp/@empno').getnumberval())
      FROM Emp_view e;
```

becomes:

```
SELECT e.employee_id
    FROM employees e
  WHERE e.salary > 15000;
```

The rewritten query is a simple relational query. The `extract()` operator followed by `getNumberVal()` is rewritten down to the relational column access as defined in the `EMP_VIEW` view

- Querying with `existsNode()` operator to select from `EMP_VIEW`:

```
SELECT EXTRACTVALUE(VALUE(e), '/Emp/name'),
       EXTRACTVALUE(VALUE(e), '/Emp/hiredate')
      FROM Emp_view e WHERE EXISTSNODE(VALUE(e), '/Emp[@empno=101]') = 1;
```

becomes:

```
SELECT e.first_name ||' '|| e.last_name, e.hire_date
    FROM employees e
  WHERE e.employee_id = 101 AND e.salary > 15000;
```

The rewritten query is a simple relational query. The XPATH predicate in `existsNode()` operator is rewritten down to the predicate over relational columns as defined in `EMP_VIEW` view.

If there is an index created on the `EMPLOYEES.EMPNO` column, then the query optimizer may use the index to speed up the query.

Querying with `existsNode()` operator to select from `EMP_VIEW`

```
SELECT EXTRACTVALUE(VALUE(e), '/Emp/name'),
       EXTRACTVALUE(VALUE(e), '/Emp/hiredate'),
       EXTRACTVALUE(VALUE(e), '/Emp/@empno')
       FROM Emp_view e
WHERE EXISTSNODE(VALUE(e),'/Emp[name="Steven King" or @empno = 101] ') = 1;
```

becomes:

```
SELECT e.first_name ||' '|| e.last_name, e.hire_date, e.employee_id
    FROM employees e
    WHERE (e.first_name ||' '|| e.last_name = 'Steven King' OR e.employee_id
 = 101)
         AND e.salary > 15000;
```

The rewritten query is a simple relational query. The XPath predicate in `existsNode()` operator is rewritten down to the predicate over relational columns as defined in `EMP_VIEW` view.

- Querying with `extract()` operator to select from `EMP_VIEW`

```
SELECT EXTRACT(VALUE(e), '/Emp/name'),
       EXTRACT(VALUE(e), '/Emp/hiredate')
      FROM Emp_view e;
```

becomes:

```
SELECT CASE WHEN e.first_name ||' '|| e.last_name IS NOT NULL THEN
       XMLELEMENT("name",e.first_name ||' '|| e.last_name) ELSE NULL END,
       CASE WHEN e.hire_date IS NOT NULL
            THEN XMLELEMENT("hiredate",e.hire_date)
       ELSE NULL END
  FROM employees e WHERE e.salary > 15000;
```

The rewritten query is a simple relational query. The `extract()` operator is rewritten to expressions over relational columns.

Note: Since the view uses `XMLForest()` to formulate `name` and `hiredate` elements, the rewritten query uses equivalent CASE expression to be consistent with `XMLForest()` semantics.

XPath Rewrite on View Constructed With SQL/XML Generation Functions

Example 16–11 illustrates an XPath rewrite on XML-schema-based `XMLType` view constructed with a SQL/XML function.

Example 16–11 XML-Schema-Based Views Constructed With SQL/XML

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/emp_simple.xsd', 4);
END;
/

BEGIN
 dbms_xmlschema.registerSchema('http://www.oracle.com/emp_simple.xsd',
   '<schema xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://www.oracle.com/emp_simple.xsd" version="1.0"
       xmlns:xdb="http://xmlns.oracle.com/xdb"
       elementFormDefault="qualified">
 <element name = "Employee">
  <complexType>
   <sequence>
    <element name = "EmployeeId" type = "positiveInteger"/>
    <element name = "Name" type = "string"/>
    <element name = "Job" type = "string"/>
    <element name = "Manager" type = "positiveInteger"/>
    <element name = "HireDate" type = "date"/>
    <element name = "Salary" type = "positiveInteger"/>
    <element name = "Commission" type = "positiveInteger"/>
    <element name = "Dept">
       <complexType>
          <sequence>
            <element name = "DeptNo" type = "positiveInteger" />
            <element name = "DeptName" type = "string"/>
            <element name = "Location" type = "positiveInteger"/>
          </sequence>
       </complexType>
    </element>
   </sequence>
  </complexType>
 </element>
</schema>', TRUE, TRUE, FALSE);
```

```
END;
/

CREATE OR REPLACE VIEW emp_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/emp_simple.xsd" ELEMENT "Employee"
   WITH OBJECT ID (extract(object_value,
                   '/Employee/EmployeeId/text()').getnumberval()) AS
   SELECT XMLElement("Employee",
     XMLAttributes( 'http://www.oracle.com/emp_simple.xsd' AS "xmlns" ,
              'http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
              'http://www.oracle.com/emp_simple.xsd
               http://www.oracle.com/emp_simple.xsd' AS "xsi:schemaLocation"),
      XMLForest(e.employee_id     AS "EmployeeId",
                e.last_name       AS "Name",
                e.job_id          AS "Job",
                e.manager_id      AS "Manager",
                e.hire_date       AS "HireDate",
                e.salary          AS "Salary",
                e.commission_pct  AS "Commission",
                XMLForest(d.department_id    AS "DeptNo",
                          d.department_name  AS "DeptName",
                          d.location_id      AS "Location") AS "Dept"))
   FROM employees e, departments d
   WHERE e.department_id = d.department_id;
```

A query using the `extractValue()` XML operator to select from `emp_xml`:

```
SELECT
 EXTRACTVALUE(VALUE(E), '/Employee/EmployeeId') as "a1",
 EXTRACTVALUE(VALUE(E), '/Employee/Name') as "b1",
 EXTRACTVALUE(VALUE(E), '/Employee/Job') as "c1",
 EXTRACTVALUE(VALUE(E), '/Employee/Manager') as "d1",
 EXTRACTVALUE(VALUE(E), '/Employee/HireDate') as "e1",
 EXTRACTVALUE(VALUE(E), '/Employee/Salary') as "f1",
 EXTRACTVALUE(VALUE(E), '/Employee/Commission') as "g1"
FROM emp_xml e
WHERE EXISTSNODE(VALUE(e), '/Employee/Dept[Location = 1700]') = 1;
```

becomes:

```
SELECT e.employee_id a1, e.last_name b1, e.job_id c1, e.manager_id d1,
       e.hire_date e1,
       e.salary f1, e.commission_pct g1
   FROM employees e, departments d
   WHERE e.department_id = d.department_id AND d.location_id = 1700;
```

The rewritten query is a simple relational query. The XPath predicate in `existsNode()` operator is rewritten down to the predicate over relational columns as defined in the `EMP_VIEW` view:

Querying with `existsNode()` operator to select from `emp_xml`

```
SELECT  EXTRACTVALUE(VALUE(e), '/Employee/EmployeeId') as "a1",
        EXTRACTVALUE(VALUE(e), '/Employee/Dept/DeptNo') as "b1",
        EXTRACTVALUE(VALUE(e), '/Employee/Dept/DeptName') as "c1",
        EXTRACTVALUE(VALUE(e), '/Employee/Dept/Location') as "d1"
      FROM emp_xml e
      WHERE EXISTSNODE(VALUE(e), '/Employee/Dept[Location = 1700 and
                                  DeptName="Finance"]') = 1;
```

becomes a simple relational query using the XPath rewrite mechanism. The XPath predicate in existsNode() operator is rewritten down to the predicate over relational columns as defined in the EMP_VIEW view:

```
SELECT e.employee_id a1, d.department_id b1, d.department_name c1,
         d.location_id d1
   FROM employees e, departments d
   WHERE (d.location_id = 1700 AND d.department_name = 'Finance')
   AND e.department_id = d.department_id;
```

XPath Rewrite on Views Using Object Types, Object Views, and SYS_XMLGEN()

The following sections describe XPath rewrite on XMLType views using object types, views, and SYS_XMLGEN().

XPath Rewrite on Non-Schema-Based Views Using Object Types or Views

Non-schema-based XMLType views can be created on existing relational and object-relational tables with object types and object views. This provides users with an XML view of the underlying data.

Existing relational data can be transformed into XMLType views by creating appropriate object types, and doing a SYS_XMLGEN at the top-level.

Example 16–12 Non-Schema-Based Views Constructed Using SYS_XMLGEN()

```
CREATE TYPE emp_t AS OBJECT (empno         NUMBER(6),
                             ename         VARCHAR2(25),
                             job           VARCHAR2(10),
                             mgr           NUMBER(6),
                             hiredate      DATE,
                             sal           NUMBER(8,2),
                             comm          NUMBER(2,2));
/

CREATE TYPE emplist_t AS TABLE OF emp_t;
/

CREATE TYPE dept_t AS OBJECT (deptno       NUMBER(4),
                              dname        VARCHAR2(30),
                              loc          NUMBER(4),
                              emps         emplist_t);
/

CREATE OR REPLACE VIEW dept_ov OF dept_t
  WITH OBJECT ID (deptno) as
  SELECT d.department_id, d.department_name, d.location_id,
         CAST(MULTISET(
           SELECT emp_t(e.employee_id, e.last_name, e.job_id, e.manager_id,
                  e.hire_date, e.salary, e.commission_pct)
             FROM employees e
             WHERE e.department_id = d.department_id)
             AS emplist_t)
    FROM departments d;

CREATE OR REPLACE VIEW dept_xml OF XMLType
  WITH OBJECT ID (extract(object_value, '/ROW/DEPTNO').getNumberVal()) AS
  SELECT sys_xmlgen(value(o)) FROM dept_ov o;
```

Querying department numbers that have at least one employee making a salary more than $15000

```
SELECT extractValue(value(x), '/ROW/DEPTNO')
  FROM dept_xml x
  WHERE existsNode(value(x), '/ROW/EMPS/EMP_T[sal > 15000]') = 1;
```

becomes:

```
SELECT d.department_id
  FROM departments d
  WHERE EXISTS (SELECT NULL FROM employees e
                WHERE e.department_id = d.department_id
                  AND e.salary > 15000);
```

Example 16–13 Non-Schema-Based Views Constructed Using SYS_XMLGEN() on an Object View

For example, the data in the `emp` table can be exposed as follows:

```
CREATE TYPE emp_t AS OBJECT
      (empno         NUMBER(6),
       ename         VARCHAR2(25),
       job           VARCHAR2(10),
       mgr           NUMBER(6),
       hiredate      DATE,
       sal           NUMBER(8,2),
       comm          NUMBER(2,2));
/

CREATE VIEW employee_xml OF XMLType
  WITH OBJECT  ID
        (object_value.EXTRACT('/ROW/EMPNO/text()').getnumberval()) AS
    SELECT SYS_XMLGEN(
        emp_t(e.employee_id, e.last_name, e.job_id, e.manager_id,
              e.hire_date, e.salary, e.commission_pct))
    FROM employees e;
```

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.employee_id, e.last_name, e.job_id, e.manager_id,
               e.hire_date, e.salary, e.commission_pct)
    FROM employees 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
    (object_value.EXTRACT('/ROW/EMPNO/text()').getnumberval()) AS
        SELECT SYS_XMLGEN(value(x)) from employee_ov x;
```

Queries on `SYS_XMLGEN` views are rewritten to access the object attributes directly if they meet certain conditions. Simple XPath traversals with `existsNode()`, `extractValue()`, and `extract()` are candidates for rewrite. See Chapter 6, "XML

Schema Storage and Query: Advanced Topics", for details on XPath rewrite. 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(e.employee_id)
  FROM employees e
  WHERE e.last_name = 'Smith';
```

XPath Rewrite on XML-Schema-Based Views Using Object Types or Object Views

Example 16–14 illustrates XPath rewrite on an XML-schema-based `XMLType` view using an object type.

Example 16–14 XML-Schema-Based Views Constructed Using Object Types

This example uses the same object types andXML Schema (`emp_complex.xsd`) as Example 16–5 on page 16-11.

```
CREATE VIEW xmlv_adts OF XMLType
   XMLSchema "http://www.oracle.com/emp_complex.xsd" ELEMENT "Employee"
   WITH OBJECT OID (
          object_value.extract(
            '/Employee/EmployeeId/text()').getNumberVal()) AS
          SELECT emp_t(e.employee_id, e.last_name, e.job_id, e.manager_id,
                    e.hire_date, e.salary, e.commission_pct,
                    dept_t(d.department_id, d.department_name, d.location_id))
            FROM employees e, departments d
            WHERE e.department_id = d.department_id;
```

A query using `extractValue()` operator:

```
SELECT extractValue(OBJECT_VALUE, '/Employee/EMPNO') "EmpID ",
       extractValue(OBJECT_VALUE, '/Employee/ENAME') "Ename ",
       extractValue(OBJECT_VALUE, '/Employee/JOB') "Job ",
       extractValue(OBJECT_VALUE, '/Employee/MGR') "Manager ",
       extractValue(OBJECT_VALUE, '/Employee/HIREDATE') "HireDate ",
       extractValue(OBJECT_VALUE, '/Employee/SAL') "Salary ",
       extractValue(OBJECT_VALUE, '/Employee/COMM') "Commission ",
       extractValue(OBJECT_VALUE, '/Employee/DEPT/DEPTNO') "Deptno ",
       extractValue(OBJECT_VALUE, '/Employee/DEPT/DNAME') "Deptname ",
       extractValue(OBJECT_VALUE, '/Employee/DEPT/LOC') "Location "
  FROM xmlv_adts
  WHERE existsNode(OBJECT_VALUE, '/Employee[SAL > 15000]') = 1;
```

becomes:

```
SELECT e.employee_id "EmpID ", e.last_name "Ename ", e.job_id "Job ",
       e.manager_id "Manager ", e.hire_date "HireDate ", e.salary "Salary ",
       e.commission_pct "Commission ", d.department_id "Deptno ",
       d.department_name "Deptname ", d.location_id "Location "
  FROM employees e, departments d
  WHERE e.department_id = d.department_id AND e.salary > 15000;
```

XPath Rewrite Event Trace

You can disable XPath rewrite for views constructed using a SQL/XML function by using the following event flag:

```
ALTER SESSION SET EVENTS '19027 trace name context forever, level 64';
```

You can disable XPath rewrite for view constructed using object types, object views, and `SYS_XMLGEN()` by using the following event flag:

```
ALTER SESSION SET EVENTS '19027 trace name context forever, level 1';
```

You can trace why XPath rewrite does not happen by using the following event flag. The trace message is printed in the tracefile.

```
ALTER SESSION SET EVENTS '19027 trace name context forever, level 8192';
```

Generating XML Schema-Based XML Without Creating Views

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 `CREATE VIEW` step, but still must 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 15, "Generating XML Data from the Database"

Example 16–15 Generating XML Schema-Based XML Without Creating Views

This example uses the same type and XML Schema definitions as Example 16–6 on page 16-14. With those definitions, `createXML` creates XML that is XML Schema-based.

```
SELECT (XMLTYPE.createXML(
          dept_t(d.department_id, d.department_name, d.location_id,
              CAST(MULTISET(SELECT emp_t(e.employee_id, e.last_name, e.job_id,
                   e.manager_id, e.hire_date, e.salary,
                   e.commission_pct)
                              FROM employees e
                              WHERE e.department_id = d.department_id)
                         AS emplist_t)),
          'http://www.oracle.com/dept_complex.xsd', 'Department'))
  FROM departments d;
```

As XMLType has an automatic constructor, `XMLTYPE.createXML` could in fact be replaced by just `XMLTYPE` here.

17

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 topics:

How Oracle Database Works with URLs and URIs

In developing Internet applications, and particularly Internet-based XML applications, you often must 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 Application Program Interface (API) name does case matter.

Oracle Database 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. Oracle Database 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 style sheets.

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 when 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.

> **See Also:** ,"XSL Transformation" on page 3-80

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.

Example 17–1 Using UriFactory

```
CREATE table uri_tab (url URIType);
INSERT INTO uri_tab VALUES (httpuritype.createuri('http://www.oracle.com'));
INSERT INTO uri_tab VALUES
       (dburitype.createuri('/SCOTT/EMP/ROW[ENAME="Jack"]'));
SELECT e.url.getclob() FROM uri_tab e;
```

See Also: "Registering New UriType Subtypes with the UriFactory Package" on page 17-19

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 17–1 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

Oracle *Database* supports datatypes in the database to store and retrieve objects that represent URIs. See "UriType Values Store Uri-References" on page 17-5. Each datatype uses a different protocol, such as HTTP, to retrieve data.

Oracle Database also provides 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 style sheets within database-generated Web pages. Oracle-supplied package `DBMS_METADATA` uses `DBUri` to reference XSL style sheets. `XDBUri` can also be used to reference XSLT style sheets 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, then you can replace it with a reference to the database by either:

  – Using a servlet

  – Using a `DBUri` or `XDBUri` to bring back the results

  Using `DBUri` or `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 are not required 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.

  Because 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`.

  See Also: *PL/SQL Packages and Types Reference*, "DBMS_METADATA package"

UriType Values Store Uri-References

URIs or Universal Resource Identifiers identify resources such as Web pages anywhere on the Web. Oracle Database provides the following `UriType subtypes` 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. Oracle Database provides a standard servlet than can be installed that interprets `DBUriType`.

Advantages of Using UriType Values

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.*** Because you can use UriType values 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 17-17

UriType Functions

The `UriType` abstract type supports a variety of functions that can be used over any subtype. Table 17–1 lists the `UriType` member functions.

Table 17–1 UriType Member Functions

UriType Member Functions	Description
getClob()	Returns the value pointed to by the URL as a character LOB value. The character encoding will be that of the database character set.

Table 17–1 (Cont.) UriType Member Functions

UriType Member Functions	Description
getUrl()	Returns the URL stored in the `UriType`. Do not use "url" directly. Use this function instead. This can be overridden by subtypes to give you the correct URL. For example, `HttpUriType` stores only the URL and not the http:// prefix. Hence `getUrl()` actually prepends the prefix and returns the value.
getExternalUrl()	Similar to the former (`getUrl`), 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 Oracle Database Works with URLs and URIs" on page 17-1.
getContentType()	Returns the MIME information for the URL. For `UriType`, this is an abstract function.
getXML()	Returns the `XMLType` object corresponding to the given URI. This is provided so that an application that must perform operations other than `getClob` or `getBlob` can use the `XMLType` methods to do those operations. This throws an exception if the URI does not point to a valid XML document.
getBlob()	Returns the Binary Large Object (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 17–2 lists the `HttpUriType` member functions.

Table 17–2 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 `getUrl`, 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.
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 `XMLType` object corresponding to this URI. Will throw an error if the target data is not XML. See also "getXML() Function" on page 17-7.

Table 17–2 (Cont.) HttpUriType Member Functions

HttpUriType Method	Description
getContentType()	Returns the MIME information for the URL. See also "getContentType() Function" on page 17-7.
createUri()	httpUriType constructor. Constructs the `httpUriType`.
httpUriType()	httpUriType constructor. Constructs the `httpUriType`.

Example 17–2 Using HTTPUriType

The following example creates a URI table to store the HTTP instances:

```
create table uri_tab ( url httpuritype);
```

Insert the HTTP instance:

```
insert into uri_tab values
       (httpuritype.createUri('http://www.oracle.com'));
```

Generate the HTML:

```
select e.url.getclob() from uri_tab e;
```

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 17–3 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), then an error is thrown.

DBUri, Intra-Database References

`DBUriType`, 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 17-25.

Formulating the DBUri

The URL syntax is obtained by specifying XPath-like syntax over a virtual XML visualization of the database. See Figure 17–1, "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. A child element is ann element that is wholly contained within another, referred to as its parent element. For example `<Parent><Child></Child></Parent>` illustrates a child element nested within its parent element.

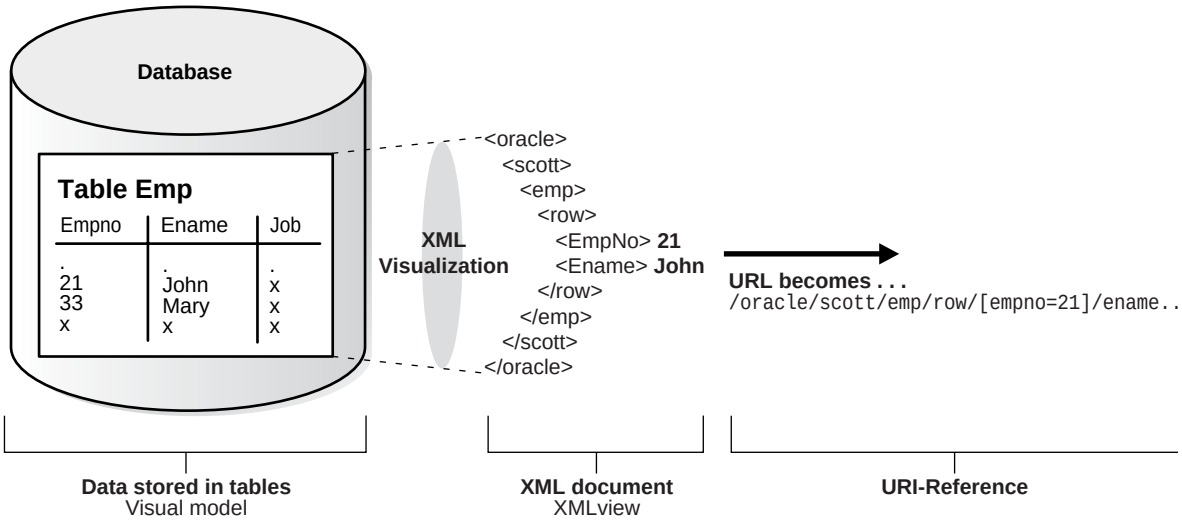
Example 17–4 The Virtual XML Document that Scott Sees

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 17–1 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 Example 17–4:

- 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 Oracle 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`.

You can create `DBUri` references 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. Oracle supports using an XPath to `XMLType` tables.

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 17–5 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
```

then you must include the pono predicate along with the ROW node as:

```
/oradb/scott/purchase_obj_tab/ROW[pono=100]/line_item_list
```

where purchase_obj_tab is a table in the SCOTT schema.

- A DBUri *must* identify exactly a single data value, either an object type or a collection. If the data value is an entire row, then you indicate that by including a ROW node. The DBUri can also point to an entire table. Note that only valid XML can be returned.

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 except within XMLType of XMLType tables. 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, then you can search on the attribute values as well.

Example 17–6 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 that has California or Oregon as state and either Portland as city name or 94404 as zipcode.

> **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:

```
/oradb/schemaname/tablename
```

Example 17–7 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 17–8 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 this 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 17–9 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 17–10 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 XSLT style sheets are stored in a CLOB column, you can retrieve the document text without having any enclosing column name tags. You can use the `text()` 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 17–11 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` because 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`, typically you will use `SCOTT/EMP`, because the actual table and column names are stored capitalized in the Oracle data dictionary.

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 Style Sheets in the Database

Applications can use XSLT style sheets to convert XML into other formats. The style sheets are represented as XML documents, stored as CLOBs. The application can use `DBUriType` objects:

- To access the XSLT style sheets stored in the database for use during parsing.
- To make references, such as import or include, to related XSLT style sheets. You can encode these references within the XSLT style sheet 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 17–3 lists the `DBUriType` methods and functions.

Table 17–3 DBUriType Methods and Functions

Method/Function	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 the URL that is stored in the `DBUriType`.
getExternalUrl()	Similar to `getUrl`, 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.
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 `XMLType` object corresponding to this URI.
getContentType()	Returns the MIME information for the URL.
createUri()	Constructs a `DBUriType` instance.
dbUriType()	Constructs a `DBUriType` instance.

Some of the functions that have a different or special action 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`, because 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 subtype of UriType and was introduced with Oracle9*i*. It provides a way to expose documents in 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 17–4 lists the XDBUriType methods. These methods do not take any arguments.

Table 17–4 XDBUriType Methods

Method	Description
getClob()	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.
get Blob()	Returns the value pointed to by the URL as a Binary Large Object (BLOB) value.
getUrl()	Returns the URL that is stored in the XDBUriType.
getExternalUrl()	Similar to getUrl, 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.
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 must perform operations other than getClob or 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 17–12 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 17–13 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, poName VARCHAR2(1000));
  -- We create an abstract type column so any type of URI can be used

-- Insert an absolute url into poUrl
-- The factory will create an XDBUriType because there's no prefix.
-- Here, po1.xml is an XML file that is stored in /public/orders/
INSERT INTO uri_tab VALUES
  (UriFactory.getUri('/public/orders/po1.xml'), 'SomePurchaseOrder');

-- 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:
CREATE FUNCTION returnclob()
RETURN CLOB
IS
  a UriType;
BEGIN
  -- Get absolute URL for purchase order named like 'Some%'
   SELECT poUrl INTO a FROM uri_tab WHERE poName LIKE 'Some%';
   RETURN a.getClob();
END;
/
```

Example 17–14 Retrieving Purchase Orders at a URL Using UriType, getXML() and extractValue()

Because `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 Oracle Database using Oracle Text. No special datastore is needed.

See: Chapter 4, "XMLType Operations","Indexing XMLType Columns" on page 4-26

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 17–4 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 17–15 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, poName VARCHAR2(200));
       -- We have created abstract type columns; if you know what kind of URIs
       -- you are going to store, you can create the appropriate types.

 -- 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.
-- 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');

-- Extract all the purchase orders
SELECT e.poUrl.getClob(), poName FROM uri_tab e;

-- In PL/SQL
CREATE FUNCTION returnclob()
RETURN CLOB
IS
  a SYS.UriType;
BEGIN
  SELECT poUrl INTO a FROM uri_Tab WHERE poName LIKE 'RelPo%';
  RETURN a.getClob();
END;
/
```

See: "Creating Instances of UriType Objects with the UriFactory Package" on page 17-19 for a description of how to use `UriFactory`

Using the Substitution Mechanism

You can create columns of the `UriType` directly and insert `HttpUriType`, `XDBUriType`, and `DBUriType` values into that column. You can also query the

column without knowing where the referenced document lies. For example, from Example 17–15, you inserted DBUri references into the uri_tab table as follows:

```
INSERT INTO uri_tab VALUES
  (UriFactory.getUri(
     '/oradb/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;
```

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 17–5.

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://, then getUri creates and returns an instance of a HttpUriType object.
- If the string starts with either /oradb/ or /dburi/, then getUri creates and returns an instance of a DBUriType object.
- If the string does not start with one of the prefixes noted in the preceding bullets, then getUri creates and returns an instance of a XDBUriType object.

> **Note:** The way UriFactory generates DBUriType instances has changed since Oracle9*i* release 1 (9.0.1):
>
> In Oracle9*i* 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 must have a /oradb o r /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 17–5 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.
`registerUrlHandler()` `PROCEDURE registerUrlHandler(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.
`unRegisterUrlHandler()` `PROCEDURE unregisterUrlHandler(prefix in varchar2)`	Unregisters a URL handler.

Example 17–16 UriFactory: Registering the ecom Protocol

Assume that you are storing different kinds of URIs in a single table:

```
CREATE TABLE url_tab (urlcol varchar2(80));
-- Insert an HTTP URL reference
INSERT INTO url_tab VALUES ('http://www.oracle.com/');
-- Insert a DBUri-ref reference
INSERT INTO url_tab VALUES ('/oradb/SCOTT/EMP/ROW[ENAME="Jack"]');
-- Create a new type to handle a new protocol called ecom://
-- This is just an example template. For it to execute, the implementations
-- of these functions need to be specified.
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:// prefixes
BEGIN
  -- 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;
/
-- Insert 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/EMP/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, then 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()`.

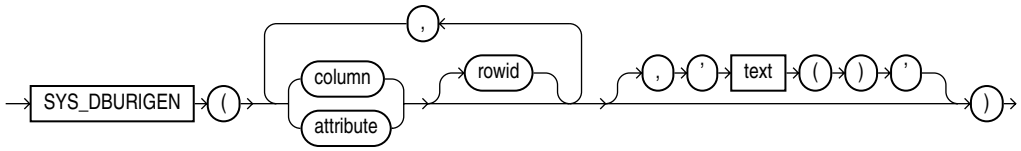
Example 17–17 SYS_DBURIGEN(): Generating a URI of type DBUriType that points to a Column

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 17–2.

Figure 17–2 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, then 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:** *Oracle Database SQL Reference* for SYS_DBURIGEN syntax details

If you do not specify an XML schema, then 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 is not required to be declared as a unique or primary key, as long as the columns uniquely identify a particular row in the result.
- 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, then the column is used both as the key column to identify the row and as the referenced column.

Example 17–18 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 17–19 Inserting Database References Using SYS_DBURIGEN()

```
CREATE TABLE doc_list_tab(docno NUMBER PRIMARY KEY, doc_ref SYS.DBUriType);
-- Insert /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, then 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 17–20 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 the story of my time in Egypt....');
```

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 the story of 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 17–21 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 run the SYS_DBURIGEN function inside PL/SQL as follows:

```
DECLARE
  ret SYS.dburitype;
BEGIN
  -- execute 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 is being desupported. Consequently the `oracle.xml.dburi.OraDBUriServlet` supported in Oracle9*i* 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 25, "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 produces 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 17–22 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 computer `machine.oracle.com` is running Oracle 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 17–6 describes the three optional arguments you can pass to `DBUri` servlet to customize the output.

Table 17–6 DBUri Servlet: Optional Arguments

Argument	Description
rowsettag	Changes the default root tag name for the XML document. For example: `http://machine.oracle.com:8080/oradb/SCOTT/EMP?rowsettag=Employee` This can also be used to put a tag around a URI that points to multiple rows. For example:
contenttype	Specifies the MIME type of the returned document. For example: `http://machine.oracle.com:8080/oradb/SCOTT/EMP?contenttype=text/plain`
transform	This argument passes a URL to UriFactory, which in turn retrieves the XSL stylehseet at that location. This style sheet is then applied to the XML document being returned by the servlet. For example: `http://machine.oracle.com:8080/oradb/SCOTT/EMP?transform=/oradb/SCOTT/XSLS/DOC/text()&contenttype=text/html`

Note: When using XPath notation in the URL for this servlet, you may have to precede certain characters with an escape character such as square brackets. You can use the `getExternalUrl()` functions in the `UriType` types 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 must 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 25, "Writing Oracle XML DB Applications in Java"
- Chapter 26, "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 17–23 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 25, "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 Oracle 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 17-19, 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:** *Oracle XML API Reference* for details of all functions in `DBUriFactory`

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:

18

Accessing Oracle XML DB Repository Data

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 Application Program Interfaces (APIs). It also introduces you to using `RESOURCE_VIEW` and `PATH_VIEW` as the SQL mechanism for accessing and manipulating repository data. It includes a table for comparing repository operations through the various resource APIs.

This chapter contains these topics:

Introducing Oracle XML DB Foldering

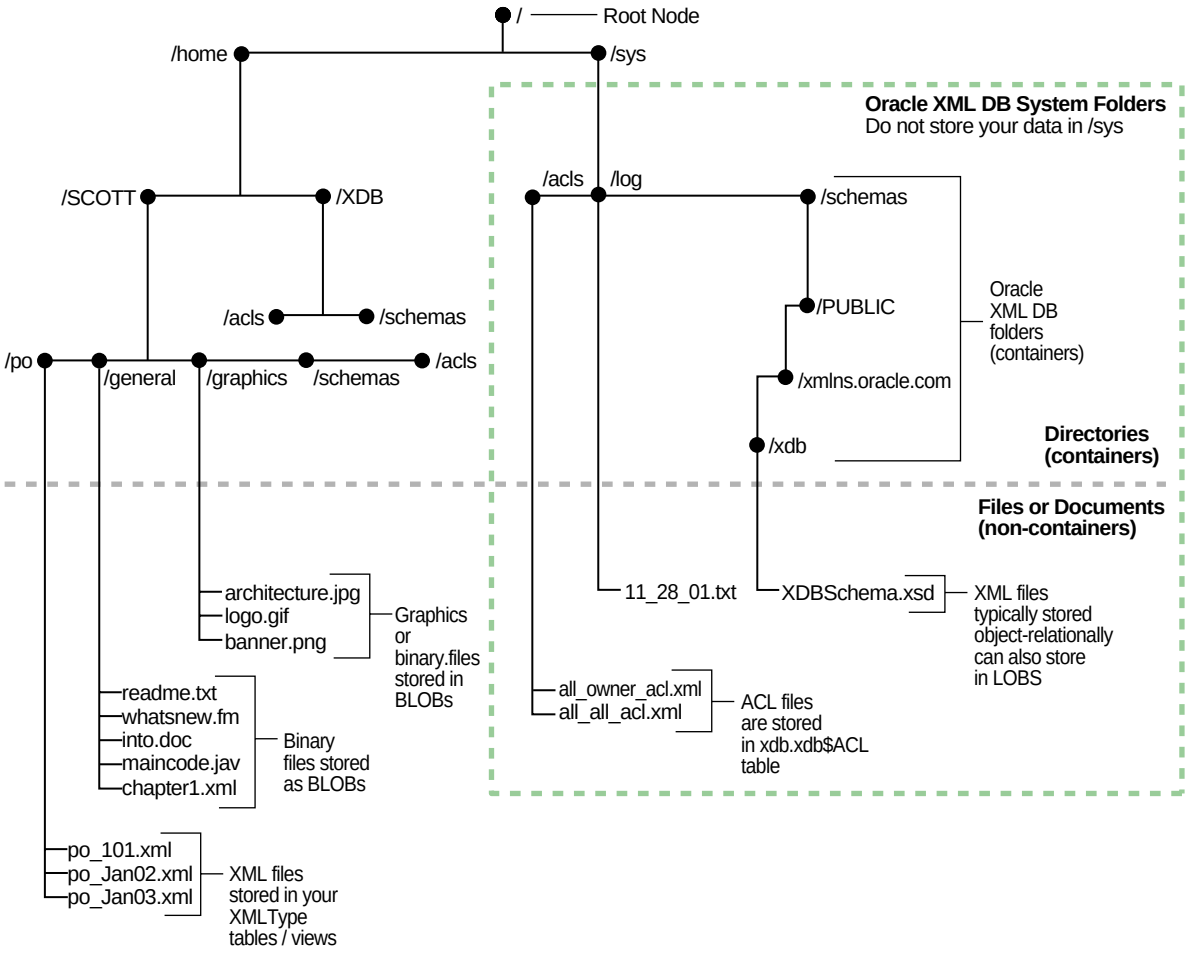
Using the foldering feature in Oracle XML DB you can store content in the database in hierarchical structures, as opposed to traditional relational database structures.

Figure 18–1 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 18–1 A Typical Folder Tree Showing Hierarchical Structures in Oracle XML Repository



Note: The directory `/sys` is used by Oracle XML DB to maintain system-defined XML schemas, Access Control Lists (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 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"
- Chapter 21, "PL/SQL Access and Management of Data Using DBMS_XDB"
- Chapter 22, "Java Access to Repository Data Using Resource API for Java"
- Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data"

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 list 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 4.
- Folder: A folder is a node (or directory) in the hierarchy that can contain a collection of resources. A folder is also a resource.
- Path Name: 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 namely any `XMLType` object that is mapped into the Oracle XML DB file system hierarchy.

See Also: Chapter 23, "Oracle XML DB Resource Security"

Many terms used by Oracle XML DB have common synonyms used in other contexts, as shown in Table 18–1.

Table 18–1 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

Table 18–1 (Cont.) Synonyms for Oracle XML DB Foldering Terms

Synonym	Oracle XML DB Foldering Term	Usage
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

Current Repository Folder List

The following is the current list of supplied Oracle XML DB repository files and folders. Note that besides the `sys` and `public` folders, you can create your own folders and files wherever you want:

```
/public
/sys
/sys/acls
/sys/acls/all_all_acl.xml
/sys/acls/all_owner_acl.xml
/sys/acls/bootstrap_acl.xml
/sys/acls/ro_all_acl.xml
/sys/apps
/sys/log
/sys/schemas
/sys/schemas/PUBLIC
/sys/schemas/PUBLIC/www.w3.org
/sys/schemas/PUBLIC/www.w3.org/2001
/sys/schemas/PUBLIC/www.w3.org/2001/xml.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBFolderListing.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResource.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBSchema.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBStandard.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/dav.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/ftplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/httplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/xdblog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/stats.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/xdbconfig.xsd
/xdbconfig.xml
```

Oracle XML DB Resources

Oracle XML DB resources conform to the `xdbresource.xsd` schema. This XML schema 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 that the tables be generated by Oracle XML DB, then 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 in 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:** "Creating Default Tables During XML Schema Registration" on page 5-12 in Chapter 5, "XML Schema Storage and Query: The Basics"

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, "XML Schema Storage and Query: The Basics"

Path-Name 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, then they receive 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. Because 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 20-10

Accessing Oracle XML DB Repository Resources

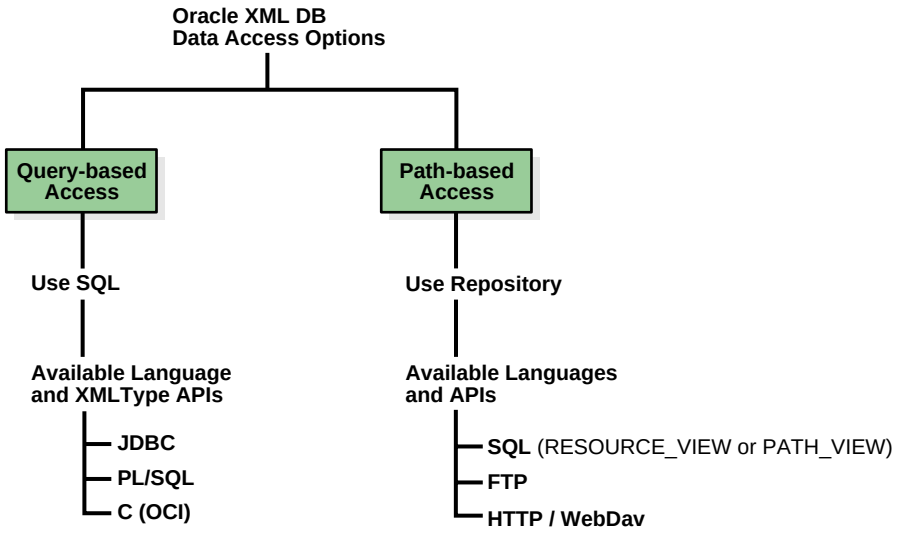
Oracle XML DB provides two techniques for accessing resources:

- "Navigational or Path Access" on page 18-7. Navigational/or 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 18-12. 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 18–2 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: a. How Structured Is Your Data?" on page 2-4.

> **See Also:** Table 18–3, " Accessing Oracle XML DB Repository: API Options"

Figure 18–2 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 host name, 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 18–3.
- Access repository data using HTTP/WebDAV from an Internet Explorer browser, such as when viewing Web Folders, as shown in Figure 18–4.

Figure 18–3 Oracle XML DB Folders in Windows Explorer

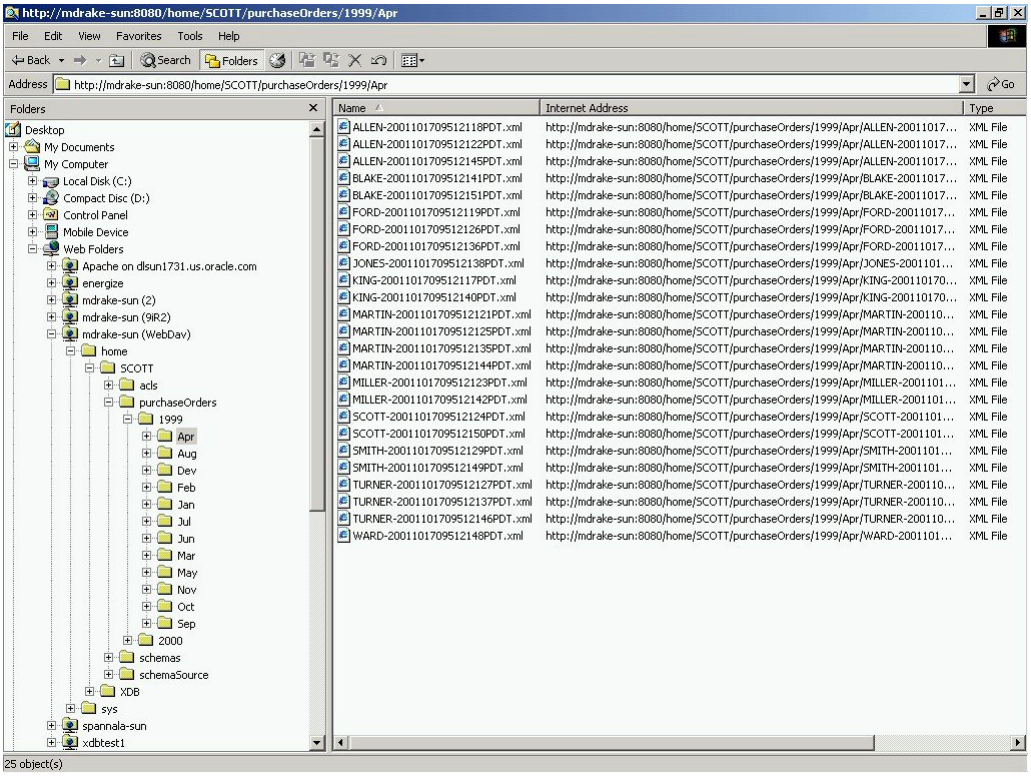
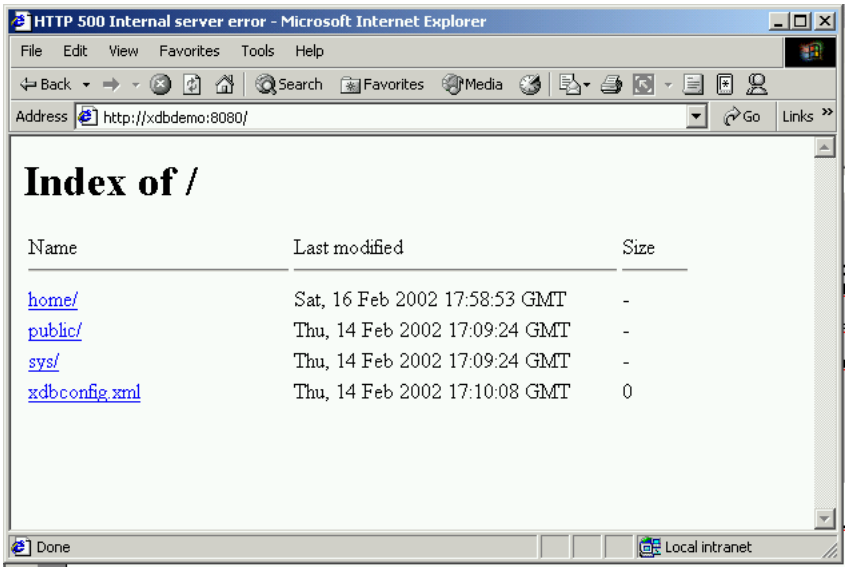


Figure 18–4 Accessing Repository Data Using HTTP/WebDAV and Navigational Access From IE Browser: Viewing Web Folders



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

Oracle XML DB protocol access uses the following steps:

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 re-authenticated (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`, then the session is run as the `ANONYMOUS` user. If the session has already been authenticated as the `ANONYMOUS` user, then there is no cost to reuse the existing session. If authentication has been provided, then the database re-authentication routines are used to authenticate the connection.
5. The request is parsed.
6. If the requested path name maps to a servlet (for HTTP only), then the servlet is invoked using Java Virtual Machine (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 file name extension for .xml, .xsl, .xsd, and so on. If the document is XML, then 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, then the default table specified for that element is used to store the contents of that resource.

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 24, "FTP, HTTP, and WebDAV Access to Repository Data"

Configuring Default Namespace to Schema Location Mappings

In general, XML DB identifies schema-based XMLType instances by pre-parsing the input XML document. If the appropriate `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attribute is found, the specified schema location URL is used to lookup the registered schema. If the appropriate `xsi:` attribute is not found, the XML document is considered to be non-schema-based.

XML DB provides a mechanism to configure default schema location mappings. If the appropriate `xsi:` attribute is not specified in the XML document, the default schema location mappings will be used. The XDB Configuration Schema has an element "schemaLocation-mappings" that can be used to specify the mapping between (namespace, element) pairs and its default schema location. If the "element" value is empty, the mapping applies to all global elements in the specified namespace. If the "namespace" value is empty, it corresponds to the null namespace.

The definition of the "schemaLocation-mappings" element in the XDB configuration file `<xdbconfig>` is as follows:

```
<element name="schemaLocation-mappings"
         type="xdbc:schemaLocation-mapping-type" minOccurs="0"/>

<complexType name="schemaLocation-mapping-type"><sequence>
     <element name="schemaLocation-mapping"
              minOccurs="0" maxOccurs="unbounded">
     <complexType><sequence>
       <element name="namespace" type="string"/>
       <element name="element" type="string"/>
       <element name="schemaURL" type="string"/>
     </sequence></complexType>
     </element></sequence>
</complexType>
```

The schema location used depends on mappings in the XDB configuration file for the namespace used and the root document element. For example, assume that the document does not have the appropriate `xsi:` attribute to indicate the schema location. Consider a document root element R in namespace N. The algorithm for identifying the default schema location is as follows:

1. If the XDB configuration file has a mapping for N and R, the corresponding schema location is used.

2. If the configuration file has a mapping for N, but not R, the schema location for N is used.
3. If the document root R does not have any namespace, the schema location for R is used.

For example, if your XDB configuration file includes the following mapping:

```
<schemaLocation-mappings>
   <schemaLocation-mapping>
      <namespace>http://www.oracle.com/example</namespace>
      <element>root</element>
      <schemaURL>http://www.oracle.com/example/sch.xsd</schemaURL>
   </schemaLocation-mapping>
   <schemaLocation-mapping>
      <namespace>http://www.oracle.com/example2</namespace>
      <element></element>
      <schemaURL>http://www.oracle.com/example2/sch.xsd</schemaURL>
   </schemaLocation-mapping>
   <schemaLocation-mapping>
     <namespace></namespace>
     <element>specialRoot</element>
     <schemaURL>http://www.oracle.com/example3/sch.xsd</schemaURL>
   </schemaLocation-mapping>
</schemaLocation-mappings>
```

The following schema locations are used:

- Namespace = `http://www.oracle.com/example`
  Root Element = `root`
  Schema URL = `http://www.oracle.com/example/sch.xsd`

  This mapping is used when the instance document specifies:

  ```
  <root xmlns="http://www.oracle.com/example">
  ```

- Namespace = `http://www.oracle.com/example2`
  Root Element = `null` (any global element in the namespace)
  Schema URL = `http://www.oracle.com/example2/sch.xsd`

  This mapping is used when the instance document specifies:

  ```
  <root xmlns="http://www.oracle.example2">
  ```

- Namespace = `null` (i.e null namespace)
  Root Element = `specialRoot`
  Schema URL = `http://www.oracle.com/example3/sch.xsd`

  This mapping is used when the instance document specifies:

  ```
  <specialRoot>
  ```

Note: This functionality is available only on the server side—when the XML is parsed on the server. If the XML is parsed on the client side, the appropriate `xsi:` attribute is still required.

Configuring XML File Extensions

XML DB repository treats certain files as XML documents based on their file extensions. When such files are inserted into the repository, XML DB pre-parses them to identify the schema location (or uses the default mapping if present) and inserts the document into the appropriate default table.

By default, the following extensions are considered as XML file extensions: xml, xsd, xsl, xlt. In addition, XML DB provides a mechanism for applications to specify other file extensions as XML file extensions. The "xml-extensions" element is defined in the XDB Configuration schema as follows:

```
<element name="xml-extensions"
         type="xdbc:xml-extension-type" minOccurs="0"/>


<complexType name="xml-extension-type"><sequence>
     <element name="extension" type="xdbc:exttype"
              minOccurs="0" maxOccurs="unbounded">
     </element></sequence>
 </complexType>
```

For example, the following fragment from the XDB configuration file *xdbconfig.xml* specifies that files with extensions vsd, vml and svgl should be treated as XML files:

```
<xml-extensions>
  <extension>vsd</extension>
  <extension>vml</extension>
  <extension>svgl</extension>
</xml-extensions>
```

Query-Based Access

Oracle XML DB provides two repository views to enable SQL access to repository data:

- `PATH_VIEW`
- `RESOURCE_VIEW`

Table 18–2 summarizes the differences between `PATH_VIEW` and `RESOURCE_VIEW`.

Table 18–2 Differences Between PATH_VIEW and RESOURCE_VIEW

PATH_VIEW	RESOURCE_VIEW
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 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW" for details on SQL repository access
- Chapter 23, "Oracle XML DB Resource Security"
- *Oracle XML API Reference*

Accessing Repository Data Using Servlets

Oracle XML DB implements Java Servlet API, version 2.2, with the following exceptions:

- All servlets must be distributable. They must expect to run in different VMs.
- WAR and `web.xml` files are not supported. Oracle XML DB supports a subset of the XML configurations in this file. An XSL style sheet can be applied to the `web.xml` to generate servlet definitions. An external tool must be used to create database roles for those defined in the `web.xml` file.
- JSP (Java Server Pages) support can be installed as a servlet and configured manually.
- `HTTPSession` and related classes are not supported.
- Only one servlet context (that is, one Web application) is supported.

See Also: Chapter 25, "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
- 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 18–3 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 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"
- Chapter 21, "PL/SQL Access and Management of Data Using DBMS_XDB"
- Chapter 22, "Java Access to Repository Data Using Resource API for Java"
- Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data"
- *Oracle XML API Reference*

Table 18–3 Accessing Oracle XML DB Repository: API Options

Data Access Operation	Query-Based Access: RESOURCE_VIEW API Path-Based Access: Resource API for PL/SQL	Path-Based Access: Protocols
Creating a resource	`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	`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	`UPDATE RESOURCE_VIEW SET resource = updateXML(res, '/Resource/propname', newval, '/Resource/propname2', newval2, …) WHERE EQUALS_PATH(res, :path) > 0;`	`WebDAV PROPPATCH;` FTP N/A
Updating the ACL of a resource	`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	`DELETE FROM RESOURCE_VIEW WHERE EQUALS_PATH(res, :path) > 0;`	`HTTP DELETE;` `FTP DELETE`
Forcibly removing all links to a resource	`DELETE FROM PATH_VIEW WHERE extractValue(res,'display_name') = 'My resource';`	N/A
Moving a resource or folder	`UPDATE PATH_VIEW SET path = newpath WHERE EQUALS_PATH(res, :path) > 0`	`WebDAV MOVE;` `FTP RENAME`
Copying a resource or folder	`INSERT INTO PATH_VIEW SELECT ::newpath, res, link FROM PATH_VIEW WHERE EQUALS_PATH(res,:oldpath)>0;`	`WebDAV COPY;` FTP N/A
Creating a link to an existing resource	`Call dbms_xdb.Link(srcpath IN VARCHAR2, linkfolder IN VARCHAR2, linkname IN VARCHAR2);`	N/A

Table 18–3 (Cont.) Accessing Oracle XML DB Repository: API Options

Data Access Operation	Query-Based Access: RESOURCE_VIEW API Path-Based Access: Resource API for PL/SQL	Path-Based Access: Protocols
Getting binary or text representation of resource contents by path name	`SELECT p.res.extract('/Resource/Contents') FROM RESOURCE_VIEW p WHERE EQUALS_PATH(res, :path) > 0 SELECT XDBUriType(:path).getBlob() FROM dual;`	`HTTP GET;` `FTP GET`
Getting XMLType representation of resource contents by path name	`SELECT extract(res, '/Resource/Contents/*') FROM RESOURCE_VIEW p WHERE EQUALS_PATH(Res, :path) > 0;`	N/A
Getting resource properties by path name	`SELECT extractValue(res, '/Resource/XXX') FROM RESOURCE_VIEW WHERE EQUALS_PATH(res,:path) > 0;`	`WebDAVPROPFIND (depth = 0);` `FTP N/A`
Listing a directory	`SELECT * FROM PATH_VIEW WHERE UNDER_PATH(res, :path, 1) > 0;`	`WebDAVPROPFIND (depth = 0);` `FTP N/A`
Creating a folder	`Call dbms_ xdb.createFolder(VARCHAR2);`	`WebDAV MKCOL;` `FTP MKDIR`
Unlinking a folder	`DELETE FROM PATH_VIEW WHERE EQUALS_PATH(res, :path) > 0;`	`HTTP DELETE;` `FTP RMDIR`
Forcibly deleting a folder and all links to it	`Call dbms_ xdb.deleteFolder(VARCHAR2);`	N/A
Getting a resource with a row lock	`SELECT … FROM RESOURCE_VIEW FOR UPDATE …;`	N/A
Putting a WebDAV lock on the resource	`DBMS_ XDB.LockResource(path,true,true);`	`WebDAV LOCK;` `FTP: QUOTE LOCK`
Removing a WebDAV lock	`DBMS_ XDB.GetLockToken(:path,deltoken);` `DBMS_ XDB.UnlockToken(path,deltoken);`	`WebDAV UNLOCK;` `QUOTE UNLOCK`
Committing changes	`COMMIT;`	Automatically commits at the end of each request
Rollback changes	`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 23, "Oracle XML DB Resource Security" for more detail on using access control on Oracle XML DB folders
- Chapter 26, "Managing Oracle XML DB Using Oracle Enterprise Manager"
- *Oracle XML API Reference* the chapters on DBMS_XDB

Setting and Accessing Custom Namespace Properties

Oracle XML DB resources declare a fixed set of metadata properties such as the `Owner` and `CreationDate`. You can specify values for these metadata attributes while creating or updating resources.

You can also store proprietary (custom) tags as extra metadata with resources, that is metadata properties that are not defined by the `Resource` XML schema. To do so, you must store the extra metadata as a CLOB in the `ResExtra` element.

Note that the default schema has a top-level `any` element (declared with `maxOccurs="unbounded"`). This allows any valid XML data as part of the resource document and gets stored in the RESEXTRA CLOB column as shown in Example 18–1.

Example 18–1 Storing Extra Metadata

```
<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>
```

You cannot extend the resource schema itself. However, you can set and access custom properties belonging to other namespaces (other than `XDBResource.xsd`) using DOM operations on the `<RESOURCE>` *document*. The following example shows some basic operations.

Example 18–2 Custom Properties on a Resource Document

```
-- Utility function to append a (direct) child to given document
create or replace function appendChild(doc xmltype, child xmltype)
return xmltype as
  d dbms_xmldom.DOMDocument;
  c dbms_xmldom.DOMDocument;
  dn dbms_xmldom.DOMNode;
  cn dbms_xmldom.DOMNode;
begin
  d := dbms_xmldom.newDOMDocument(doc);
  dn := dbms_xmldom.makeNode(d);
  dn := dbms_xmldom.getFirstChild(dn);
```

```
  c := dbms_xmldom.newDOMDocument(child);
  cn := dbms_xmldom.makeNode(c);
  cn := dbms_xmldom.getFirstChild(cn);
  dn := dbms_xmldom.appendChild(dn, cn);

  return doc;
end;
/


-- create a NSB resource
declare
ret boolean;
begin
ret := dbms_xdb.createresource('/public/foo1.txt', 'abc def');
end;
/

commit;

-- update resource to set custom property <RANDOM> in foo namespace
-- Note : the custom properties should belong to a namespace other than
-- the XDBResource.xsd namespace
--
update resource_view
set res =
   appendChild(res, xmltype('<FOO:RANDOM xmlns:foo="foo">3456</FOO:RANDOM>'))
where any_path = '/public/foo1.txt';
commit;

-- select resource
select e.res.getclobval()
from resource_view e
where e.any_path = '/public/foo1.txt';

-- select custom property
select e.res.extract('/r:Resource/foo:random',
  'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
   xmlns:foo="foo"').getstringval()
from resource_view e
where any_path = '/public/foo1.txt';

-- select custom property value
select e.res.extract('/r:Resource/foo:random/text()',
  'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
   xmlns:foo="foo"').getstringval()
from resource_view e
where any_path = '/public/foo1.txt';
```

19

Managing Oracle XML DB Resource Versions

This chapter describes how to create and manage versions of Oracle XML DB resources.

This chapter contains these topics:

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. When you update a resource such as a table or column, Oracle XML DB stores the pre-update contents as a separate resource version.

Oracle XML DB provides a PL/SQL package, `DBMS_XDB_VERSION` to put a resource under version-control and retrieve different versions of the resource. Versioning is currently only available for XML non-schema based resources.

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 Path Name".

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 Oracle Database.

Oracle does not guarantee that the resource object ID of a version is preserved across checkin and checkout. Everything but the resource object ID of the last version is preserved.

Oracle XML DB does not support versioning of XML schema-based resources.

See Also: *Oracle XML API Reference*

Oracle XML DB Versioning Terms Used in This Chapter

Table 19–1 lists the Oracle XML DB versioning terms used in this chapter.

Table 19–1 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.
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 Path Name

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 19–1 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.GetResoureceByResId(resid);
END;
/
```

Creating a Version-Controlled Resource (VCR)

Oracle XML DB does not automatically keep a history of updates because 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)
- Access control list (ACL), the list of access control entries that determines which principals have access to a given resource or resources.

When a Version-Controlled Resource (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 19-3.

Example 19–2 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, that is discussed in "Resource ID of a New Version" on page 19-3.

`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 names for version resources. However, it does provide 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`, that are described in the following sections.

Resource ID of a New Version

When a VCR is checked out and updated for the first time a copy of the existing resource is created. The resource ID of the latest version of the resource is never changed. You can obtain the resource ID of the old version by getting the predecessor of the current resource.

Example 19–3 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 19–4 Oracle XML DB: Creating and Updating a Version-Controlled Resource (VCR)

```
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');

  -- Check out 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';

  -- Check in the change
  resid2 := DBMS_XDB_VERSION.Checkin('/home/SCOTT/versample.html');

  -- The latest version can be obtained by resid2 and its predecessor
  -- can be obtained by using getPredecessor() or getPredsbyResId() functions.
  -- resid1 is no longer valid.
END;
/
-- Delete the VCR
DELETE FROM resource_view WHERE any_path = '/home/SCOTT/versample.html';

-- Once the preceding delete is done, any reference to the resource
-- (that is, check-in, check-out, 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. To update a VCR follow these steps:

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. You can perform the update using SQL through RESOURCE_VIEW or PATH_VIEW, or through any protocol such as WebDAV.
3. ***Checkin or uncheckout a resource***. If the resource is unchecked out, then the older version of the resource, obtained through the predecessor, is copied onto the current version. The older version is deleted.

Checkout

In Oracle9*i* release 2 (9.2) and higher, the VCR checkout operation is executed by calling `DBMS_XDB_VERSION.CheckOut()`. If you want to commit an update of a resource, then it is a good idea to commit after checkout. If you do not commit right after checking out, then you may have to rollback your transaction at a later point, and the update is lost.

Example 19–5 VCR Checkout

For example:

```
BEGIN
  -- Resource '/home/SCOTT/versample.html' is checked out.
  DBMS_XDB_VERSION.CheckOut('/home/SCOTT/versample.html');
END;
/
```

Checkin

In Oracle9*i* release 2 (9.2) and higher, 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 19–6 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 Oracle9*i* release 2 (9.2) and higher, 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 19–7 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. In other words, you can use `RESOURCE_VIEW`, `PATH_VIEW`, or WebDAV, for example.

> **See Also:** Chapter 20, "SQL Access Using 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 23, "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.

ACL of Version-Controlled Resources are the Same as the First Versions

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 19–2 describes the subprograms in `DBMS_XDB_VERSION`.

Table 19–2 DBMS_XDB_VERSION Functions and Procedures

DBMS_XDB_VERSION Function/Procedure	Description
`FUNCTION MakeVersioned` `MakeVersioned(pathname VARCHAR2) RETURN dbms_xdb_version.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, then a copy of the resource will be created, and the given `path name` 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. `pathname` - the path name of the resource to be put under version control. `return` - 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 MakeVersioned for VCR, and neither exception nor warning is raised. It is not permitted to make versioned for folder, version resource, and ACL. An exception is raised if the resource does not exist.
`PROCEDURE Checkout` `Checkout(pathname VARCHAR2);`	Checks out a VCR before updating or deleting it. `pathname` - the path name of the VCR to be checked out. This is not an auto-commit SQL operation. Two users cannot checkout the same VCR at the same time. If this happens, then 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: ▪ the given resource is not a VCR, ▪ the VCR is already checked out ▪ the resource does not exist
`FUNCTION Checkin` `checkin (pathname VARCHAR2) RETURN DBMS_XDB_VERSION.resid_type;`	Checks in a checked-out VCR. `pathname` - the path name of the checked-out resource. `return` - the resource id of the newly-created version. This is not an auto-commit SQL operation. `Checkin` does not have to take the same path name that was passed to checkout operation. However, the checkin `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, then the new name must be used to checkin 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, then the new path name must be used to checkin the resource.
`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, then 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, then the new path name must be used to checkin the resource.

Table 19–2 (Cont.) DBMS_XDB_VERSION Functions and Procedures

DBMS_XDB_VERSION Function/Procedure	Description
`FUNCTION GetPredecessors` `GetPredecessors(pathname VARCHAR2) RETURN resid_list_type;`	Given a version resource or a VCR, gets the predecessors of the resource by `pathname`, the path name of the resource. `return` - list of predecessors. Getting predecessors by `resid` is more efficient than by `pathname`. An exception is raised if the `resid` or `pathname` is not permitted.
`GetPredsByResId(resid dbms_xdb.resid_type) RETURN resid_list_type;`	Given a version resource or a VCR, gets the predecessors of the resource by `resid` (resource id) Note: The list of predecessors only contains one element (immediate parent), because Oracle does not support branching in this release. The following function `GetSuccessors` also returns only one element.
`FUNCTION GetSuccessors` `GetSuccessors(pathname VARCHAR2) 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 `pathname` is not permitted.
`GetSuccsByResId(resid dbms_xdb.resid_type) RETURN resid_list_type;`	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`

Guidelines for Using Oracle XML DB Versioning

This section describes guidelines for using Oracle XML DB versioning.

- You cannot switch a VCR to a non-VCR.
- You can access an old copy of a VCR after updating it. The old copy is the version resource of the last one checked-in, hence:
  - If you have the version ID or path name, then you can load it using that ID.
  - If you do not have its ID, then you can call getPredecessors() to get the ID.
- Only data in Oracle XML DB resources can be put under version control in this release.
- When a resource is turned into a VCR a copy of the resource is created and placed into the Version History. A flag marks the resource as a VCR. Earlier in this chapter it states that a version-controlled resource is an Oracle XML DB resource that is put under version control where a VCR is a reference to a version Oracle XML DB resource. It is not physically stored in the database. In other words no extra copy of the resource is stored when the resource is versioned, that is, turned into a VCR.
- Versions are stored in the same object-relational tables as the resources. In this release, versioning only works for XML non-schema-based resources, hence no unique constraints are violated.
- The documentation states that a version resource is a system-generated resource and does not have a path name. However you can still access the resource using the navigational path.

- When the VCR resource is checked out, no copy of the resource is created. When it is updated the first time, a copy of the resource is created. You can make several changes to the resource without checking it in. You will get the latest copy of the resource. Even if you are a different user, you will get the latest copy.
- Updates cannot be made to a checked out version by more than one use. Once the checkout happens and the transaction is committed, any user can edit the resource.
- When a checked out resource is checked in the original version checked out is placed into version history.
- Properties for resources are maintained for each version. Versions are stored in the same tables. In this release, versioning only works for XML non-schema-based resources, hence no constraints are violated.

20

SQL Access Using 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` that query resources based on their path names and `PATH` and `DEPTH` operators that return resource path names and depths respectively.

This chapter contains these topics:

Oracle XML DB RESOURCE_VIEW and PATH_VIEW

Figure 20–1 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 such as FTP, WebDAV, or application program interfaces (APIs), can be accessed in SQL using `RESOURCE_VIEW` values and `PATH_VIEW` values, and vice versa.

`RESOURCE_VIEW` consists of a resource, itself an `XMLType`, that contains the name of the resource that can be queried, its ACLs, and its properties, static or extensible.

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

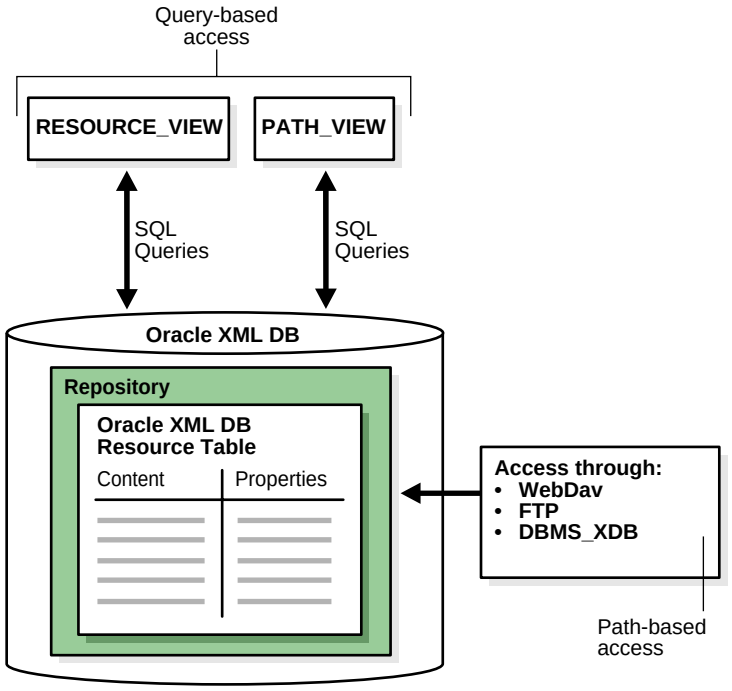
available to search the properties of a resource, and internal B*Tree indexes over Names and ACLs speed up access to these attributes of the Resource `XMLType`.

`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.

The base table for `RESOURCE_VIEW` is `XDB.XDB$RESOURCE` and should only be accessed through `RESOURCE_VIEW` or the `DBMS_XDB` API.

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

Figure 20–1 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
RESID       RAW          Resource OID which is a unique handle to the resource
```

See Also: Appendix H, "Oracle XML DB-Supplied XML Schemas and Additional Examples"

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
RESID       RAW          Resource OID
```

See Also: Appendix H, "Oracle XML DB-Supplied XML Schemas and Additional Examples"

Figure 20–2 illustrates the structure of Resource and PATH_VIEWs.

Note: Each resource may have multiple paths called links.

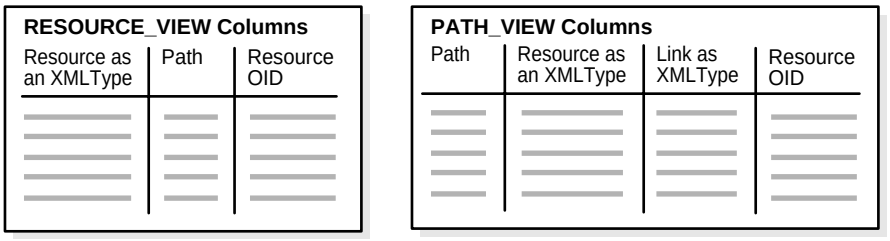
The path in the RESOURCE_VIEW is an arbitrary one and one of the accessible paths that can be used to access that resource. Oracle XML DB provides operator UNDER_PATH that enables applications to search for resources contained (recursively) within a particular folder, get the resource depth, and so on. Each row in the PATH_VIEW and RESOURCE_VIEW columns is of XMLType. DML on 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.

Path names in the ANY_PATH column of the RESOURCE_VIEW and the PATH column in the PATH_VIEW are absolute paths.

Path names from the PATH operator are relative paths under the path name specified by the UNDER_PATH operator. Suppose there are two resources pointed to by path names '/a/b/c' and '/a/d' respectively, a PATH operator that retrieves paths under the folder '/a' will return relative paths 'b/c' and 'd'.

When there are multiple links to the same resource, only paths under the path name specified by the UNDER_PATH operator are returned. Suppose '/a/b/c', '/a/b/d' and '/a/e' are links to the same resource, a query on the PATH_VIEW that retrieves all the paths under '/a/b' return only '/a/b/c' and '/a/b/d', not '/a/e'.

Figure 20–2 RESOURCE_VIEW and PATH_VIEW Structure



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 20–3 illustrates the difference between RESOURCE_VIEW and PATH_VIEW.

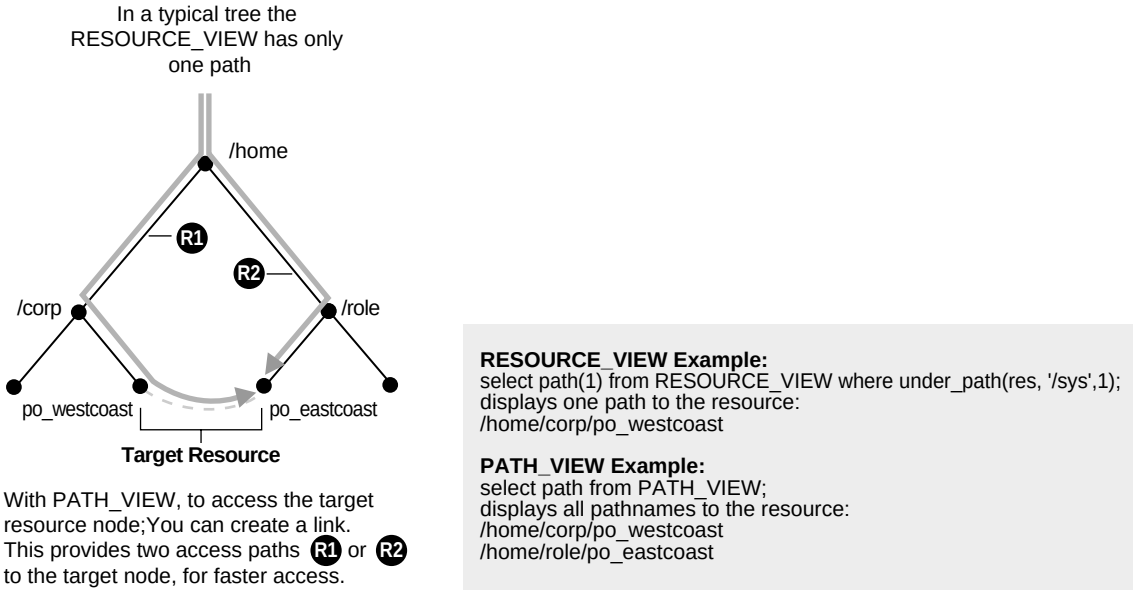
Because many Internet applications only need one URL to access a resource, RESOURCE_VIEW is widely applicable.

PATH_VIEW contains the *link* properties 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, then 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, then they will find the resource regardless of whether or not that path is the arbitrary one chosen to normally appear with that resource using RESOURCE_VIEW.

Figure 20–3 RESOURCE_VIEW and PATH_VIEW Explained



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 or its OID
  - 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 or their OID

See the "Using the Resource View and Path View API" and EQUALS_PATH.

Resource_View and Path_View APIs

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, then a functional implementation of UNDER_PATH can be chosen that walks back up the repository. This can be more efficient, because a much smaller number of links are required to be traversed. Figure 20–4 shows the UNDER_PATH syntax.

Figure 20–4 UNDER_PATH Syntax

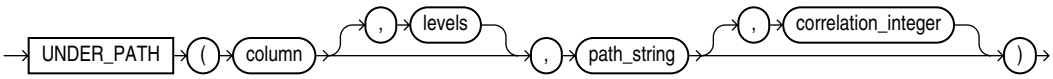


Table 20–1 describes the UNDER_PATH syntax.

Table 20–1 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: ▪ `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.
`INTEGER UNDER_ PATH(resource_column, depth, pathname);`	Determines if a resource is under a specified path, with a depth argument to restrict the number of levels to search. Parameters: ▪ `resource_column` - The column name or column alias of the `'resource'` column in the `path_view` or `resource_view.` ▪ `depth` - The maximum depth to search; a depth of less than `0` is treated as `0.` ▪ `pathname` - The path name to resolve.

Table 20–1 (Cont.) RESOURCE_VIEW and PATH_VIEW API Syntax: UNDER_PATH

Syntax	Description
`INTEGER UNDER_PATH(resource_column, pathname, correlation);`	Determines if a resource is under a specified path, with a correlation argument for ancillary operators. Parameters: ▪ `resource_column` - The column name or column alias of the `'resource'` column in the `path_view` or `resource_view.` ▪ `pathname` - The path name to resolve. ▪ `correlation` - An integer that can be used to correlate the `UNDER_PATH` operator (a primary operator) with ancillary operators (`PATH` and `DEPTH`).
`INTEGER UNDER_PATH(resource_column, depth, pathname, correlation);`	Determines if a resource is under a specified path with a depth argument to restrict the number of levels to search, and with a correlation argument for ancillary operators. Parameters: ▪ `resource_column` - The column name or column alias of the 'resource' column in the `path_view` or `resource_view.` ▪ `depth` - The maximum depth to search; a 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`). Note that only one of the accessible paths to the resource must be under the path argument for a resource to be returned.

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.

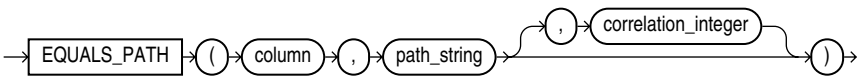
```
EQUALS_PATH INTEGER EQUALS_PATH( resource_column,pathname);
```

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.

Figure 20–5 illustrates the `EQUALS_PATH` syntax.

Figure 20–5 EQUALS_PATH Syntax



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);
```

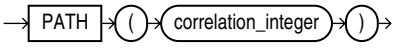
where:

- `correlation` is an integer that can be used to correlate the `UNDER_PATH` operator (a primary operator) with ancillary operators (`PATH` and `DEPTH`).

> **Note:** If a path is not under the specified `pathname` argument, a `NULL` value is returned as the output of the current path.

Figure 20–6 illustrates the `PATH` syntax.

Figure 20–6 PATH Syntax



Here are some examples of a `RESOURCE_VIEW` that include resources specified by paths:

```
'/a/b/c'
'/a/b/c/d'
'/a/e/c'
'/a/e/c/d'
```

Example 20–1 Determining Paths Under the Specified Pathname Argument

```
SELECT path(1) FROM resource_view
   WHERE UNDER_PATH(res, '/a/b', 1) = 1;
```

Returns the following:

```
PATH(1)
-------
c
c/d
2 rows returned
```

Example 20–2 Determining Paths Not Under the Specified Pathname Argument

```
SELECT path(1) FROM resource_view
   WHERE UNDER_PATH(res, '/a/b', 1)!=1
```

Returns the following:

```
PATH(1)
-------


2 rows returned
```

Note: For absolute paths use ANY_PATH as follows:

```
SELECT ANY_PATH
   FROM resource_view
   WHERE UNDER_PATH(res, '/a/b')=1;
```

This returns the following:

```
ANY_PATH
--------
/a/e/c
/a/e/c/d
2 rows returned
```

Example 20–3 Determining Paths Using Multiple Correlations

```
SELECT ANY_PATH, path(1), path(2)
   FROM resource_view
   WHERE UNDER_PATH(res, '/a/b', 1) = 1 or UNDER_PATH(res, '/a/e', 2) = 1;
```

This returns the following:

```
ANY_PATH           PATH(1)                  PATH(2)
--------------------------------------------------
/a/b/c             c
/a/b/c/d           c/d
/a/e/c                                      c
/a/e/c/d                                    c/d
4 rows returned
```

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 Repository Data Paths, Resources and Links: Examples

The following examples illustrate how you can access paths, resources, and link properties in Oracle XML DB repository:

Example 20–4 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 20–5 Using UNDER_PATH: Given a Path Name, Get a Resource From the RESOURCE_VIEW

```
SELECT any_path, extract(res, '/Resource') FROM resource_view
   WHERE under_path(res, '/sys') = 1;
```

Example 20–6 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 20–7 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 20–8 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 20–9 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 20–10 Using EQUALS_PATH to Locate a Path

```
SELECT any_path FROM resource_view
   WHERE equals_path(res, '/sys') > 0;
```

Example 20–11 Retrieve RESID of a Given Resource

```
select resid
from resource_view
where extract(res, '/Resource/Dispname') = 'example';
```

Example 20–12 Get the Path Name of a Resource Given Its OID

```
select any_path
from resource_view
where resid = :1;
```

Example 20–13 Select all Folders Under a Given Path

```
select any_path from resource_view
where under_path(res, 1, '/parent_folder') = 1
and existsNode(res, '/Resource[@Container="true"]') = 1;
```

Example 20–14 Join RESOURCE_VIEW with XMLType table MYPOs

```
SELECT Extract(value(p), '/PurchaseOrder/Item').getClobval()
FROM MYPOs p, RESOURCE_VIEW
WHERE ExistsNode(value(p), 'PurchaseOrder[PONum=1001 and Company = "Oracle
 Corp"]') = 1 and any_path like '/public/po1%';
```

Example 20–15 Creating Resources: Inserting Data Into a Resource

Note the insert syntax change here: a `NULL` value is added for the `resid` column.

```
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/example', NULL);
```

Inserting Data into a Repository Resource: Examples

The following example illustrates how you can insert data into a resource:

Example 20–16 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/example', NULL);
```

Deleting Repository Resources: Examples

The following examples illustrate how you can delete resources or paths:

Example 20–17 Deleting Resources

```
DELETE FROM resource_view WHERE any_path = '/home/SCOTT/example';
```

If only leaf resources are deleted, then you can perform a delete using `delete from resource_view where...`.

Example 20–18 Deleting Resources With Multiple Links

For multiple links to the same resource, deleting from `RESOURCE_VIEW` will delete all the links to that resource, while deleting from `PATH_VIEW` will delete only the specified path.

Suppose `'/home/file1'` is a link to `'/public/file'`.

```
DELETE FROM resource_view WHERE equals_path(res, '/home/file1')=1;
```

deletes both paths from the repository.

```
DELETE FROM path_view WHERE equals_path(res, '/home/file1')=1;
```

deletes `'/home/file1'` only.

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, then 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 20–19 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 20–20 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 20–21 Updating a Path in the PATH_VIEW

```
UPDATE path_view set path = '/home/XDB'
  WHERE path = '/home/SCOTT/example';
```

Example 20–22 Updating DisplayName in RESOURCE_VIEW

If you use the following code to update `DisplayName` in `RESOURCE_VIEW.res` after uploading a document, it is updated in the `RESOURCE_VIEW`, However, if you access using FTP, the old name and bytes are reset to `0`.

```
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">
    <DisplayName>bearprob_new.txt</DisplayName>
    <Language>en</Language>
    <CharacterSet>utf-8</CharacterSet>
    <ContentType>text/plain</ContentType>
   <RefCount>1</RefCount>
 </Resource>')
     WHERE any_path = '/bearprob.txt';
```

Here the `RESOURCE_VIEW UPDATE` statement replaces the entire resource, including its contents. Because the new Resource document specified does not have a `Contents` element, the contents are cleared. The following statement will update just the `DisplayName`:

```
UPDATE RESOURCE_VIEW r SET r.res = updatexml(r.res,
'/Resource/DisplayName/text()', 'bear_prob_new.txt') WHERE
   any_path='/bear_prob.txt';
```

The FTP protocol does not distinguish between `DisplayName` and the name of the resource in the path. If you have a resource at the path `/a/b/c`, the FTP client always displays it as `c`. However, Web Folders show the display name because the WebDAV protocol can distinguish between the path name and display name. Also note that updating the `DisplayName`, using the `RESOURCE_VIEW UPDATE` statement shown in the previous paragraph does not change the path of the resource. The resource will still be located at `/bearprob.txt`.

Note: If you must get all the resources under a directory, then you can use the `LIKE` operator, as shown in Example 20–4 on page 20-8.

If you must 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 20–5 on page 20-9.

The query plan for Example 20–4 will be more optimal than that of Example 20–5.

See Also: Chapter 18, "Accessing Oracle XML DB Repository Data",Table 18–3, " Accessing Oracle XML DB Repository: API Options" on page 18-14 for additional examples that use the `RESOURCE_VIEW` and `PATH_VIEW` operators

Working with Multiple Oracle XML DB Resources Simultaneously

Operations listed in Table 18–3 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 resource = updateXML(res, '/Resource/Contents', lob)
    WHERE extractValue(resource, '/Resource/DisplayName') = 'My stuff';
```

- Finding recursively in a folder

```
SELECT FROM RESOURCE_VIEW WHERE UNDER_PATH(resource, '/public') ...
```

- Copying a set of Oracle XML DB resources

```
INSERT INTO PATH_VIEW
  SELECT '/newlocation' || path, res, link, NULL FROM PATH_VIEW
    WHERE UNDER_PATH(resource, '/public', 1) = 1
    ORDER BY path;
```

Performance Tuning of XML DB

XML DB uses the `xdbconfig.xml` file for configuring the system and protocol environment. It includes an element `resource-view-cache-size` parameter that defines the in-memory size of the `RESOURCE_VIEW` cache. The default value is 1048576.

Some queries on `RESOURCE_VIEW` and `PATH_VIEW` can be sped up by tuning `resource-view-cache-size`. In general, the bigger the cache size, the faster the query. The default `resource-view-cache-size` is appropriate for most cases. However, you may want to enlarge your `resource-view-cache-size` element when querying a sizable `RESOURCE_VIEW`.

The extensible optimizer decides whether the `UNDER_PATH` or the `EQUALS_PATH` operator is evaluated by domain index scan or functional implementation. The optimizer needs statistics for XML DB to achieve the optimal query plan. Statistics can be collected by analyzing the XML DB tables and hierarchical index under XDB using the `ANALYZE` command or the `DBMS_STATS` package. The following is an example of using the `ANALYZE` command:

```
analyze table xdb$h_link compute statistics;
analyze table xdb$resource compute statistics;
analyze index xdbhi_idx delete statistics;
```

Two new performance elements are included in 10*g* release 1 (10.1). The default limits for these elements are soft limits. The system automatically adapts when these limits are exceeded. These elements are:

- `xdbcore-loadableunit-size` - This element indicates the maximum size to which a loadable unit (partition) can grow in Kilobytes. When a partition is read into memory or a partition is built while consuming a new document, the partition is built until it reaches the maximum size. The default value is 16 Kb.
- `xdbcore-xobmem-bound` - This element indicates the maximum memory in kilobytes that a document is allowed to occupy. The default value is 1024 Kb. Once the document exceeds this number, some loadable units (partitions) are swapped out.

> **See Also:** Appendix A, "Installing and Configuring Oracle XML DB"

Searching for Resources Using Oracle Text

The `XDB$RESOURCE` table in Oracle XML DB user schema stores in Oracle XML DB the metadata and data corresponding to resources, such as files and folders. You can search for resources containing a specific keyword by using the `CONTAINS` operator in `RESOURCE_VIEW` or `PATH_VIEW`.

Example 20–23 Find All Resources Containing Keywords "Oracle" and "Unix"

```
SELECT path
   FROM path_view
   WHERE contains(res, 'Oracle AND Unix') > 0;
```

Example 20–24 Find All Resources Containing Keyword "Oracle" that are Also Under a Specified Path.

```
SELECT any_path
   FROM resource_view
   WHERE contains(res, 'Oracle') > 0
   AND under_path(res, '/myDocuments') > 0;
```

To evaluate such queries, you must create a Context Index on the XDB$RESOURCE table. Depending on the type of documents stored in Oracle XML DB, choose one of the following options for creating your Context Index:

- ***If Oracle XML DB contains only XML documents***, that is, no binary data, a regular Context Index can be created on the XDB$RESOURCE table.

  See Also: Chapter 4, "XMLType Operations" and Chapter 9, "Full Text Search Over XML"

  ```
  CREATE INDEX xdb$resource_ctx_i
     ON xdb.xdb$resource x (value(x))
     INDEXTYPE IS ctxsys.context;
  ```

- ***If Oracle XML DB contains binary data***, for example Microsoft Word documents, a user filter is required to filter such documents prior to indexing. It is recommended that you use the DBMS_XDBT package (dbmsxdbt.sql) to create and configure the Context Index.

  See Also:

  - *Oracle XML API Reference*, the chapter on DBMS_XDBT for information on installing and using DBMS_XDBT.
  - Appendix F, "SQL and PL/SQL APIs: Quick Reference", DBMS_XDBT on page F-21

  ```
  REM Install the package - connected as SYS
  @dbmsxdbt
  REM Create the preferences
  exec dbms_xdbt.createPreferences;
  REM Create the index
  exec dbms_xdbt.createIndex;
  ```

DBMS_XDBT package also includes procedures to sync and optimize the index. You can use the configureAutoSync() procedure to configure automatic sync of the index by using job queues.

21

PL/SQL Access and Management of Data Using DBMS_XDB

This chapter describes the Oracle XML DB resource application program interface (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.

This chapter contains these topics:

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:

- *PL/SQL Packages and Types Reference*
- *Oracle XML API Reference*
- Appendix F, "SQL and PL/SQL APIs: Quick Reference"

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 access control list (ACL) based Security. ACL is a list of access control entries that determines which principals have access to a given resource or resources.

- Oracle XML DB Configuration
- Oracle XML DB Hierarchical Index Rebuild

DBMS_XDB: Oracle XML DB Resource Management

Table 21–1 lists the DBMS_XDB Oracle XML DB resource management methods.

Table 21–1 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.
`CreateResource`	`FUNCTION CreateResource (path IN VARCHAR2, data IN VARCHAR2) RETURN BOOLEAN;` Creates a new resource with the given string as its contents. `FUNCTION CreateResource (path IN VARCHAR2, data IN SYS.XMLTYPE) RETURN BOOLEAN;` Creates a new resource with the given XMLType data as its contents. `FUNCTION CreateResource (path IN VARCHAR2, datarow IN REF SYS.XMLTYPE) RETURN BOOLEAN;` Given a REF to an existing XMLType row, creates a resource whose contents point to that row. That row should not already exist inside another resource. `FUNCTION CreateResource (path IN VARCHAR2, data IN CLOB) RETURN BOOLEAN;` Creates a resource with the given CLOB as its contents. `FUNCTION CreateResource (abspath IN VARCHAR2, data IN BFILE, csid IN NUMBER := 0) RETURN BOOLEAN;` Creates an XML DB resource. For input data from `BFILE` or `BLOB`, an optional new parameter, `csid`, is added to the function to identify the character set of the input contents. `FUNCTION CreateResource (abspath IN VARCHAR2, data IN BLOB, csid IN NUMBER := 0) RETURN BOOLEAN;` For input data from `BFILE` or `BLOB`, an optional new parameter, `csid`, is added to the function to identify the character set of the input contents.
`CreateFolder`	Argument: `(path IN VARCHAR2)` Return value: `TRUE` if successful.
`DeleteResource`	Argument: `(path IN VARCHAR2)` Return value: N/A

The input character set id, if specified, must be a valid Oracle id; otherwise, an error is returned. This value if nonzero overrides any encoding specified in the source data. Otherwise, the character encoding of the resource is based on the MIME type which is derived from the abspath argument. If the MIME type is "`*/xml`" then the character encoding is determined by auto-detection as defined in Appendix F of the W3C XML Recommendation; otherwise, the character encoding of the input is defaulted to the database character set so that no conversion is applied. The character encoding of the data file is determined based on the following precedence:

- The `csid` value if nonzero.

- The MIME type based on the file extension specified in the `abspath` argument. If the MIME type is "`*/xml`" then the character encoding is determined by auto-detection, as defined in Appendix F of the W3C XML Recommendation. Otherwise, the character set of the BFILE or BLOB data is defaulted to the database character set.

> **Note:** the determination of the character set based on the MIME type is only done when creating a resource. Afterward, changing the MIME type of a resource, either through the `RESOURCE_VIEW` or `XDBResource` Java class, will not affect the character set property of the resource.

Use IANA character name for XML documents if possible. The character set id can be derived from an IANA name using: `UTL_GDK.CHARSET_MAP` and `NLS_CHARSET_ID` functions. For example:

Example 21–1 Deriving the Character Set ID From an IANA Name Using UTL_GDK.CHARSET_MAP

```
DECLARE
   oracs  VARCHAR2(255);
   csid   NUMBER;
BEGIN
   oracs := UTL_GDK.CHARSET_MAP('EUC-JP', utl_gdk.IANA_TO_ORACLE);
   csid := NLS_CHARSET_ID(oracs);
END;
```

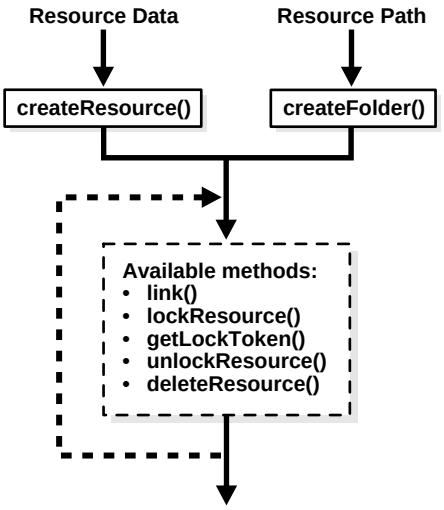
Using DBMS_XDB to Manage Resources, Calling Sequence

Figure 21–1 describes the calling sequence when using `DBMS_XDB` to manage repository resources:

1. When managing repository resources the calling sequence diagram assumes that the resources and folders already exist. If not, then you must 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.
2. If the resource or folder does not need additional processing or managing, then they are simply output.
3. If the resource or folder need additional processing or managing, then you can apply any or all of the following methods as listed in Table 21–1:
   - `Link()`
   - `LockResource()`
   - `GetLockToken()`
   - `UnlockResource()`
   - `DeleteResource()`

See Example 21–2 for an examples of using `DBMS_XDB` to manage repository resources.

Figure 21–1 Using DBMS_XDB to Manage Resources: Calling Sequence



Example 21–2 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 21–2 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 21–2 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

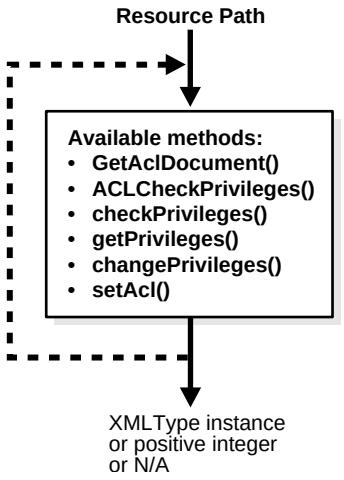
Using DBMS_XDB to Manage Security, Calling Sequence

Figure 21–2 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 21–2 to perform security management tasks:
   - getAclDocument()
   - ACLCheckPrivileges()
   - checkPrivileges()
   - getPrivileges()
   - changePrivileges()
   - setACL()

See Example 21–3 for an examples of using DBMS_XDB to manage repository resource security.

Figure 21–2 Using DBMS_XDB to Manage Security: Calling Sequence



Example 21–3 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 21–3 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 21–3 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

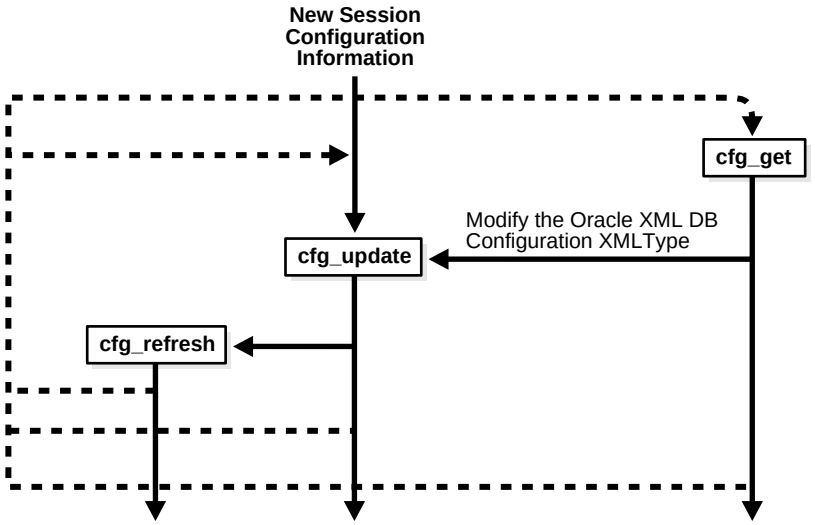
Figure 21–3 shows the calling sequence when using DBMS_XDB for configuration management.

The diagram shows the following sequence:

1. To manage the Oracle XML DB configuration, first retrieve the configuration instance using cfg_get.
2. You can then optionally modify the Oracle XML DB configuration XMLType instance to update it, or simply produce the Oracle XML DB configuration.
3. To update the Oracle XML DB configuration resource use cfg_update. You must 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 are not required to input a configuration XMLType instance.

See Example 21–4 for an example of using DBMS_XDB for configuration management of repository resources.

Figure 21–3 Using DBMS_XDB for Configuration Management: Calling Sequence



Example 21–4 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 ...
  dbms_xdb.cfg_update(config);
end;
/

-- To pick up the latest Oracle XML DB Configuration
--   In this example it is not needed as cfg_update()
--   automatically does a cfg_refresh().
call dbms_xdb.cfg_refresh();
```

22

Java Access to Repository Data Using Resource API for Java

This chapter describes the Oracle XML DB resource application program interface (API) for Java.

This chapter contains these topics:

- Introducing Oracle XML DB Resource API for Java
- Using Oracle XML DB Resource API for Java
- Oracle XML DB Resource API for Java Parameters
- Oracle XML DB Resource API for Java: Examples

Introducing Oracle XML DB Resource API for Java

See Also:

- *Oracle XML API Reference*
- Chapter 12, "Java API for XMLType"
- Appendix E, "Java APIs: Quick Reference"

Using Oracle XML DB Resource API for Java

With Oracle XML DB resource API for Java you use JDBC to access Oracle XML DB resource views for retrieving and modifying database resources.

Additionally, you can perform these operations:

- Use the Java API for `XMLType` to access and modify parts of `XMLType` objects.
- Use the `save()` method to write changes to the database.

Oracle XML DB resource API for Java includes a set of sub-interfaces that indicate resource type `Versioning` information WebDAV.

The API includes interfaces for objects such as workspaces, branches, and baselines (as defined by the WebDAV versioning specification).

Oracle XML DB Resource API for Java Parameters

Table 22–1 lists the parameters supported by Oracle XML DB resource API for Java.

Table 22–1 Oracle XML DB Resource API for Java: 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 DBMS_XDB.spi.XDBContextFactory.
XDB_RESOURCE_TYPE	Determines what data is returned to the application by default when a path name is resolved: ■ Resource: A resource class will be returned. ■ XMLType: The contents of the resource will be returned. In this release, Oracle XML DB has implemented only the javax.naming package. Oracle XML DB has an extension to this package, an extension to the lookup() 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).

Oracle XML DB Resource API for Java: Examples

Example 22–1 Resource JDBC: Using SQL To Determine Purchase Order Properties

This examples uses a SQL SELECT and criteria other than path name to retrieve 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();
```

23

Oracle XML DB Resource Security

This chapter describes the access control list (ACL) based security mechanism for Oracle XML DB resources, how to create ACLs, set and change ACLs on resources, and how ACL security interacts with other Oracle Database security mechanisms.

This chapter contains these topics:

Introducing Oracle XML DB Resource Security and ACLs

Oracle XML DB maintains object-level security for all resources in the Oracle XML DB repository.

> **Note:** XML objects that are 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 the Oracle XML DB repository. An ACL is a list of access control entries that determine which principals have access to a given resource or resources. ACLs are a standard security mechanism used in Java, Windows NT, and other systems.

See Also:

- Chapter 26, "Managing Oracle XML DB Using Oracle Enterprise Manager"
- Appendix F, "SQL and PL/SQL APIs: Quick Reference"
- *Oracle XML API Reference*

How the ACL-Based Security Mechanism Works

ACLs in Oracle XML DB are XML schema-based resources. They are stored and managed in Oracle XML DB. Every resource in the Oracle XML DB is protected by an ACL. Before a user performs an operation or method on a resource, a check of user privileges on the resource takes place. The set of privileges checked depends on the operation or method being performed. For example, to increase employee Scott's salary by 10 percent, `READ` and `WRITE` privileges are needed for the `/home/SCOTT/salary.xml` resource.

Some ACLs are supplied with Oracle XML DB. There is only one ACL, the **bootstrap ACL,** located at `/sys/acls/bootstrap_acl.xml` in Oracle XML DB repository, that is self-protected; that is, it is protected by its own contents. This ACL, supplied with Oracle XML DB, grants READ privilege to all users. The bootstrap ACL also grants FULL ACCESS to XDBADMIN (Oracle XML DB `ADMIN`) and `DBA` roles. The `XDBADMIN` role is particularly useful for users that must register global XML schemas.

Other ACLs supplied with Oracle XML DB are:

- `all_all_acl.xml` Grants all privileges to all users
- `all_owner_acl.xml` Grants all privileges to the owner of the resource
- `ro_all_acl.xml` Grants read privileges to all users

These ACLs are also protected by the bootstrap ACL.

Relationship Between ACLs and Resources

Every ACL conforms to the Oracle XML DB ACL schema which is an XML schema. This schema is located at: `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd`.

Every ACL is stored in a table named `XDB$ACL` that is owned by user XDB. This is an XML schema-based `XMLType` table. Hence every row in this table (and therefore each ACL) has a system-generated object identifier (OID) that can be accessed as a column named `OBJECT_ID`.

Every resource has a property named `ACLOID`. The `ACLOID` stores the OID of the ACL that protects the resource. Note that an ACL is also a resource in Oracle XML DB. Hence the `XMLRef` property of an ACL resource, for example, `/sys/acls/all_all_acl.xml`, is a REF to the row in table `XDB$ACL` that contains the actual content of the ACL. These two properties form the link between the `XDB$RESOURCE` table, which stores Oracle XML DB resources, and table `XDB$ACL`.

Access Control List Concepts

This section describes several access control list (ACL) terms and concepts:

- ***Principal.*** An entity that may be granted access control privileges to an Oracle XML DB resource. Oracle XML DB supports the following as principals:
  - Database users
  - Database roles. A database role can be understood as a group; for example, the DBA role represents the group of all database administrators.
  - LDAP users and groups. For details on using LDAP principals see "Integration with LDAP" on page 23-10.

  The special principal, dav:owner, corresponds to the owner of the resource being secured. The owner of the resource is one of the properties of the resource. Use of

the dav:owner principal allows greater ACL sharing between users, because the owner of the document often has special rights. See Also "Oracle XML DB Supported Privileges" on page 23-4.

- ***Privilege***: This is a particular right that can be granted or denied to a principal. Oracle XML DB has a set of system-defined rights (such as `READ` or `UPDATE`) that can be referenced in any ACL. Privileges can be granted or denied to the principal dav:owner, that represents the owner of the document, regardless of who the owner is. 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 inter operability between ACL clients. Please see "Oracle XML DB Supported Privileges" on page 23-4 for the list of atomic and aggregate privileges that can be used in ACLs.

  The set of privileges granted to a principal controls the ability of that principal 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, then the `read` privileges must be granted to `Scott` prior to the read operation. Therefore, privileges control how users can operate on resources.
- ***ACE (access control entry)***: ACE is an entry in the 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 because a user may be granted many roles.

  An Oracle XML DB ACE element has the following properties:
  - Operation: Either `grant` or `deny`
  - Principal: A valid principal, as described previously.
  - Privileges Set: Particular set of privileges to be granted or denied for a particular principal
  - Principal Format (optional): Specifies the format of the principal. It could be an LDAP distinguished name (DN), a short name (either a DB user/role or an LDAP nickname) or an LDAP GUID. The default is 'short name'. If the principal name matches both a DB user and an LDAP nickname, it is assumed to refer to the LDAP nickname.
  - Collection (optional): This is a boolean attribute that specifies whether the principal is a collection of users (LDAP group or DB role) or a single user (LDAP or DB user).
- ***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.

  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>dav:owner</principal>
    <grant>true</grant>
    <privilege>
      <dav:all/>
    </privilege>
  </ace>
</acl>
```

In this ACL there is only one ACE. The ACE grants all privileges to the owner of the resource.

- ***Default ACL:*** When a resource is inserted into the Oracle XML DB repository by default the ACL on its parent folder is used to protect the resource. After the resource is created a new ACL can be set on it.
- ***ACL file-naming conventions:*** Supplied ACLs use the following file-naming conventions: `<privilege>_<users>_acl.xml`

  where `<privilege>` represents the privilege granted and `<users>` represents the users that are granted access to the resource.
- ***ACL Evaluation Rules:*** As mentioned before, privileges are checked before a user is allowed to access a resource. This is done by evaluating the resource's ACL for the current user. To evaluate an ACL, the database collects the list of ACEs in the ACL that apply to the user logged into the current database session. The list of currently active roles for the user is maintained as a part of the session and is used to match ACEs (that specify roles as principals) with the current users. To resolve conflicts between ACEs, the following rule is used: if a privilege is denied by any ACE, then the privilege is denied for the entire ACL.

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.

Atomic Privileges

Table 23–1 lists the atomic privileges supported by Oracle XML DB.

Table 23–1 Oracle XML DB Supported 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`

Table 23–1 (Cont.) Oracle XML DB Supported Atomic Privileges

Privilege Name	Description	Database Counterpart
`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`
`link-to`	Allows resources to be linked	N/A
`unlink-from`	Allows resources to be unlinked	N/A
`read-acl`	Read the resource ACL	`SELECT`
`write-acl-ref`	Changes the resource ID	`UPDATE`
`update-acl`	Change the contents of the resource 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. Other privileges are part of the Oracle XML DB ACL namespace: `http://xmlns.oracle.com/xdb/acl.xsd`

Because 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 23–2 lists the aggregate privileges defined by Oracle XML DB, along with the atomic privileges of which they are composed.

Table 23–2 Aggregate Privileges

Aggregate Privilege Names	Atomic Privileges
`all`	All atomic privileges: `dav:read`, `dav:write`, `dav:read-acl`, `dav:write-acl`, `dav:lock`, `dav:unlock`
`dav:all`	All atomic privileges except `linkto`
`dav:read`	`read-properties`, `read-contents`, `resolve`
`dav:write`	`update`, `link`, `unlink`, `unlink-from`
`dav:read-acl`	`read-acl`
`dav:write-acl`	`write-acl-ref`, `update-acl`

Table 23–3 shows the privileges required for some common operations on resources in Oracle XML DB repository. The Privileges Required column assumes that you already have the `resolve` privilege on container C and all its parent containers, up to the root of the hierarchy.

Table 23–3 Privileges Needed for Operations on Oracle XML DB Resources

Operation	Description	Privileges Required
CREATE	Create a new resource in container C	update and link on C
DELETE	Delete resource R from container C	update and unlinkfrom on R, update and unlink on C
UPDATE	Update the contents or properties of resources R	update on R
GET	An FTP or 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.

Interaction with Database Table Security

Resources in the Oracle XML DB repository are either:

- LOB-based (content is stored in a LOB which is part of the resource). Access is determined only by the ACL that protects the resource. Or,
- REF-based (content is XML and is stored in a database table). Users must have the appropriate privilege in the underlying table (or view) where the XML content is stored, as well as permissions through the ACL for the resource.

Since the contents of a REF-based resource may actually be stored in a table, it is possible to access this data directly using SQL queries on the table. A uniform access control mechanism is one where the privileges needed are independent of the method of access (for example, FTP, HTTP, or SQL). To provide a uniform security mechanism using ACLs, the underlying table must first be hierarchy-enabled before resources that reference the rows in the table are inserted into Oracle XML DB. This is done using the DBMS_XDBZ.enable_hierarchy() procedure. This procedure adds two hidden columns to store the ACL OID and the OWNER of the resources that reference the rows in the table. It also adds a Row Level Security (RLS) policy to the table which checks the ACL whenever a SELECT, UPDATE, or DELETE statement is executed on the table. Note that the default tables produced by XML schema registration are already hierarchy-enabled.

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 have ACL checking done, although they will all have table-level security.

Note: You cannot hide data in XMLType tables from other users when using out-of-line storage. The out-of-line data is not protected by ACL Security.

Working with Oracle XML DB ACLs

As mentioned before, ACLs in Oracle XML DB are resources and hence all the methods that operate on resources also apply to ACLs. In addition, there are several APIs specific to ACLs. These are in the DBMS_XDB package. The procedures and functions in this package enable you to use PL/SQL to access Oracle XML DB security

mechanisms, check privileges given a particular ACL, and list the set of privileges the current user has for a particular ACL and a particular resource.

> **See Also:** Chapter 26, "Managing Oracle XML DB Using Oracle Enterprise Manager"

The following are examples of different ACL-related operations:

Creating an ACL Using DBMS_XDB.createResource()

Example 23–1 illustrates how to create an ACL:

Example 23–1 Creating an ACL Using DBMS_XDB.createResource()

```
declare
    b boolean;
begin
    b := DBMS_XDB.createResource('/home/SCOTT/acl1.xml',
       <acl description="myacl"
          xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
          xmlns:dav="DAV:"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                              http://xmlns.oracle.com/xdb/acl.xsd">
            <ace>
              <principal>dav:owner</principal>
                <grant>true</grant>
                  <privilege>
                   <dav:all/>
                  </privilege>
            </ace>
       </acl>');
end;
```

Setting the ACL of a Resource

Example 23–2 illustrates how to set the ACL of a resource using the `DBMS_XDB.setAcl()` procedure.

Example 23–2 Setting the ACL of a Resource

```
call DBMS_XDB.setAcl('/home/SCOTT/po1.xml', '/home/SCOTT/acl1.xml');
```

Deleting an ACL

Example 23–3 illustrates how to delete an ACL using the `DBMS_XDB.deleteResource()` procedure.

Example 23–3 Deleting an ACL

```
call DBMS_XDB.deleteResource('/home/SCOTT/acl1.xml');
```

If a resource is being protected by the ACL to be deleted, first change the ACL of that resource before deleting the ACL.

Updating an ACL

This can be done using standard methods for updating resources. In particular since an ACL is an XML document, `updateXML` and other operators can be used to manipulate ACLs.

Oracle XML DB ACLs are cached for fast evaluation. When a transaction that updates an ACL is committed, the modified ACL is picked up by existing database sessions after the timeout specified in the Oracle XML DB configuration file, `/xdbconfig.xml`, is up. The XPath for this configuration parameter is: `/xdbconfig/sysconfig/acl-max-age`. The unit of this timeout is second. Note that sessions initiated after the ACL is modified will use the new ACL without any delay.

If an ACL resource is updated with non-ACL content, the same rules that apply for deletion will apply, that is, if any resource is being protected by the ACL that is being updated, first change the ACL of that resource. There are 2 different cases for updating ACL:

- Updating the Entire ACL or Adding or Deleting an Entire ACE
- Updating Existing ACE(s)

Updating the Entire ACL or Adding or Deleting an Entire ACE

You can use FTP or WebDAV to update the ACL. For more details on how to use these protocols, see Chapter 24. You can also use `RESOURCE_VIEW` to do this.

Example 23–4 Updating the Entire ACL

```
UPDATE resource_view r SET
   r.res=updatexml('/r:Resource/r:Contents/a:acl', <new content of ACL>,
      'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
       xmlns:a="http://xmlns.oracle.com/xdb/acl.xsd"')
   WHERE r.any_path='/home/SCOTT/acl1.xml';
```

Updating Existing ACE(s)

This can be done using `RESOURCE_VIEW`. Example 23–5 illustrates changing the principal in an ACE from `SCOTT` to `JONES`:

Example 23–5 Updating Existing ACE(s) Using RESOURCE_VIEW

```
UPDATE resource_view r SET
   r.res=updatexml('/r:Resource/r:Contents/a:acl/a:ace/a:principal/text()',
                   'JONES',
                   'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                    xmlns:a="http://xmlns.oracle.com/xdb/acl.xsd"')
   WHERE r.any_path='/home/SCOTT/acl1.xml';
```

Retrieving the ACL Document for a Given Resource

Example 23–6 illustrates how to use the `DBMS_XDB.getAclDocument()` function to retrieve the ACL document for a given resource.

Example 23–6 Retrieving the ACL Document for a Resource

```
SELECT DBMS_XDB.getAclDocument('/home/SCOTT/po1.xml').getClobVal() FROM dual;
```

Retrieving Privileges Granted to the Current User for a Particular Resource

Example 23–7 illustrates how to retrieve privileges granted to the current user using the DBMS_XDB.getPrivileges() function.

Example 23–7 Retrieving Privileges Granted to the Current User for a Particular Resource

```
SELECT DBMS_XDB.getPrivileges('/home/SCOTT/po1.xml').getClobVal() FROM dual;
```

Checking if the Current User Has Privileges on a Resource

Example 23–8 illustrates how to use the DBMS_XDB.checkPrivileges() function to check if the current user has a given set of privileges on a resource. This function returns a nonzero value if the user has the privileges.

Example 23–8 Checking if the Current User has a Given Set of Privileges on a Resource

```
SELECT DBMS_XDB.checkPrivileges('/home/SCOTT/po1.xml',
       '<privilege
          xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
          xmlns:dav="DAV:"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                              http://xmlns.oracle.com/xdb/acl.xsd">
           <read-contents/>
           <read-properties/>
        </privilege>') FROM dual ;
```

Checking if the Current User Has Privileges With the ACL and Resource Owner

This is typically used by applications that must perform ACL evaluation on their own before allowing the user to perform an operation. Use the DBMS_XDB.aclCheckPrivileges() function.

Example 23–9 Checking if the Current User Has a Given Set of Privileges Given the ACL and the Resource Owner

```
SELECT DBMS_XDB.aclCheckPrivileges('/home/SCOTT/acl1.xml','SCOTT',
   '<privilege
     xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
     xmlns:dav="DAV:"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                         http://xmlns.oracle.com/xdb/acl.xsd">
       <read-contents/>
       <read-properties/>
    </privilege>') FROM dual;
```

Retrieving the Path of the ACL that Protects a Given Resource

Example 23–10 retrieves the path of the ACL that protects a given resource by using a RESOURCE_VIEW query. The query uses the fact that the XMLRef and ACLOID elements of the resource form the link between an ACL and a resource.

Example 23–10 Retrieving the Path of the ACL that Protects a Given Resource

```
SELECT a.any_path from resource_view a
   WHERE sys_op_r2o(extractValue(a.res, '/Resource/XMLRef')) =
     (select extractValue(r.res, '/Resource/ACLOID') FROM resource_view r
```

```
      WHERE r.any_path='<path_of_resource>');
```

The inner query retrieves the `ACLOID` of the given resource. The outer query retrieves the path to the resource based on the OID of the content. The latter is retrieved by applying `sys_op_r2o` to the `XMLRef`.

Retrieving the Paths of all Resources Protected by a Given ACL

Example 23–11 illustrates how to retrieve the paths of all resources protected by a given ACL. This is done with a query similar to that used in Example 23–10.

Example 23–11 Retrieving the Paths of All Resources Protected by a Given ACL

```
SELECT a.any_path FROM resource_view a
   WHERE extractValue(a.res, '/Resource/ACLOID') =
         (SELECT sys_op_r2o(extractValue(r.res, '/Resource/XMLRef'))
            FROM resource_view r
            WHERE r.any_path='<path_of_ACL>');
```

The inner query retrieves the `OID` of the specified ACL's content. The outer query selects the paths of all resources `ACLOIDs` match the `OID` of the given ACL

Integration with LDAP

Some setup steps outside of XML DB need to be performed before you can use LDAP users and groups as principals in ACLs. You need to set up the Oracle Internet Directory (OID), register the database with OID, and set up SSL authentication between the database and OID. For more details on these steps, see the *Oracle Advanced Security Administrator's Guide*.

This section describes the main steps involved in allowing LDAP users to use the features of XML DB. The scenario presented here deals with a single shared database schema (also termed schema-independent LDAP users). In this case, a single database user is created. The mapping of multiple LDAP users to the shared database schema is maintained in OID. Users can log in to the database (using SQL or protocols like FTP and WebDAV that are supported by XML DB) using the LDAP username and password. They are then automatically mapped to the corresponding shared schema.

1. Create a DB User Corresponding to the Shared Schema

   ```
   CREATE USER myapps IDENTIFIED GLOBALLY AS '' ;
   GRANT CREATE SESSION, CONNECT, RESOURCE TO myapps;
   ```

2. Create LDAP Users

   ```
   dn: cn=user1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US
   cn: user1
   sn: snuser1
   uid: uiduser1
   objectclass: top
   objectclass: inetorgperson
   objectclass: orclUser
   orclPassword: {x- orcldbpwd}1.0:46B35D8418C795B3
   ```

3. Map LDAP Users to Shared DB Schema

Example 23–12 Mapping a Single LDAP User to a Shared DB Schema

```
dn: cn=odelmMyapps1,cn=server1,cn=OracleContext,ou=Americas,o=Oracle,C=US
cn: odelmMyapps1
```

```
objectclass: top
objectclass: OrclDBEntrylevelMapping
OrclDBDistinguishedName:cn=user1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US
orclDBNativeuser:myapps
```

Example 23–13 Mapping a Subtree of Users to a Shared DB Schema

```
dn: cn=odelmMyapps2,cn=server1,cn=OracleContext,ou=Americas,o=Oracle,C=US
cn: odelmMyapps2
objectclass: top
objectclass: OrclDBSubtreelevelMapping
OrclDBDistinguishedName:ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US
orclDBNativeuser:myapps
```

4. Create LDAP groups and specify its members

Example 23–14 Creating LDAP Groups and Specifying Their Members

```
dn: cn=grp1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US
cn: grp1
objectclass: top
objectclass: orclgroup
objectclass: orclprivilegegroup
objectclass: groupOfUniqueNames
uniquemember: cn=user1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US
uniquemember: cn=user3,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US
```

5. Define ACLs in Oracle XML DB

Example 23–15 ACL Referencing an LDAP User

```
<acl description="/public/txmlacl1/acl1.xml"
      xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                          http://xmlns.oracle.com/xdb/acl.xsd">
   <ace principalFormat="DistinguishedName">
  <principal>cn=user1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US</principal>
    <grant>true</grant>
    <privilege>
      <dav:all/>
    </privilege>
  </ace>
</acl>
```

Example 23–16 ACL Referencing an LDAP Group

```
<acl  xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                          http://xmlns.oracle.com/xdb/acl.xsd">
  <ace principalFormat="DistinguishedName">
    <principal>cn=grp1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US</principal>
    <grant>true</grant>
    <privilege>
      <dav:read/>
    </privilege>
  </ace>
</acl>
```

6. Create Resources in Oracle XML DB and Protect Them with ACLs

All existing APIs and protocol methods can be used to create resources and protect them with ACLs.

7. Connect as LDAP users Using SQL/DAV/FTP and Access Oracle XML DB

   All Oracle XML DB functionality is available when you connect as the LDAP user. The implicit ACL resolution is based on the current LDAP user and the corresponding LDAP group membership information.

Performance Issues for Using ACLs

Since ACLs are checked for every repository access, the performance of the ACL check operation is critical to the performance of the repository. In XML DB the required performance for this operation is achieved by employing several caches. ACLs are cached in a shared (shared by all sessions in the instance) cache. The performance of this cache is better when there are fewer ACLs in your system. Hence it is recommended that you share ACLs (between resources) as much as possible. Also, the cache works best when the number of ACEs in an ACL is at most 16.

There is also a session-specific cache of privileges granted to a given user by a given ACL. The entries in this cache have a time out (in seconds) specified by the element `<acl-max-age>` in the XDB configuration file (`/xdbconfig.xml`). For maximum performance this timeout should be as large as possible. But note that there is a trade-off here: the greater the timeout, the longer it will take for current sessions to pick up an updated ACL.

XML DB also maintains caches to improve performance when using ACLs that have LDAP principals (LDAP groups or users). The goal of these caches is to minimize network communication with the LDAP server. One is a shared cache that maps LDAP GUIDs to the corresponding LDAP nicknames and Distinguished Names (DNs). This is used when an ACL document is being displayed (or converted to `CLOB` or `VARCHAR2` forms from `XMLType`). To purge this cache, use the `DBMS_XDBZ.PurgeLdapCache()` procedure. The other cache is session-specific and maps LDAP groups to their members (nested membership). Note that whenever XML DB encounters an LDAP group for the first time (in a session) it will get the nested membership of that group from the LDAP server. Hence it is best to use groups with as few members and levels of nesting as possible.

24

FTP, HTTP, and WebDAV Access to Repository Data

This chapter describes how to access Oracle XML DB repository data using FTP, HTTP/WebDAV protocols.

This chapter contains these topics:

Introducing Oracle XML DB Protocol Server

As described in Chapter 2, "Getting Started with Oracle XML DB" and Chapter 18, "Accessing Oracle XML DB Repository Data", 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 or file system. Because 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. The user names and passwords to be used with the protocols are the same as those for SQL*Plus. Enterprise users are also supported.

See Also: "Accessing Data Stored in Oracle XML DB Repository Resources" on page 18-13

Note: Oracle XML DB protocols are not supported on EBCDIC platforms.

Session Pooling

Oracle XML DB protocol server maintains a shared pool of sessions. Each protocol connection is associated with one session from this pool. After a connection is closed the session is put back into the shared pool and can be used to serve later connections.

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 `xdbconfig.xml` file, or disable it by setting pool size to zero.

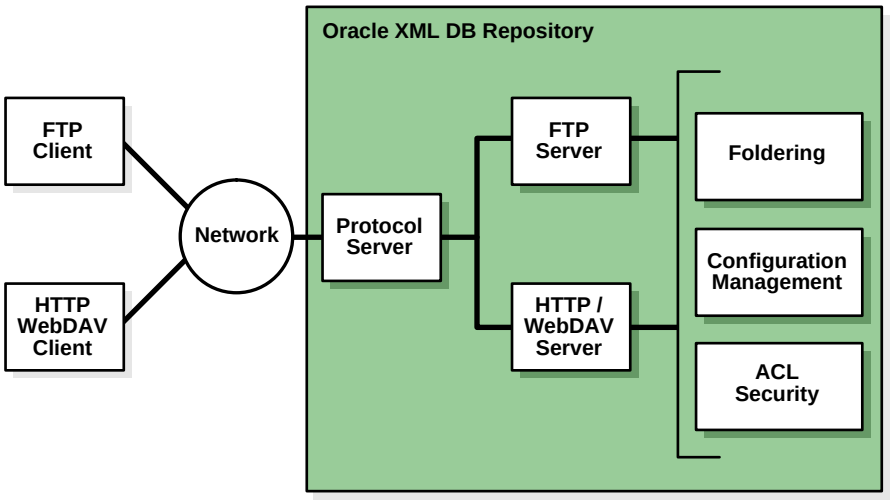
Java Servlets

Session pooling can affect users writing Java servlets, because 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. State for each user must be stored in the database or in a look-up table, rather than assuming that a session will only exist for a single user.

> **See Also:** Chapter 25, "Writing Oracle XML DB Applications in Java"

Figure 24–1 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 24–1 Oracle XML DB Architecture: Protocol Server



Oracle XML DB Protocol Server Configuration Management

Oracle XML DB protocol server uses configuration parameters stored in `/xdbconfig.xml` to initialize its startup state and manage session level configuration. The following section describes the protocol-specific configuration parameters that you can configure in the Oracle XML DB configuration file. The

session pool size and timeout parameters cannot be changed dynamically, that is, you will need to restart the database in order for these changes to take effect.

> **See Also:** Appendix A, "Installing and Configuring Oracle XML DB"

Configuring Protocol Server Parameters

Figure 24–1 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 24–2 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 24–3 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
  ```

> **See Also:**
>
> - Appendix A, "Installing and Configuring Oracle XML DB" for more information about the configuration file `/xdbconfig.xml`
> - Appendix H, "Oracle XML DB-Supplied XML Schemas and Additional Examples", the section, "xdbconfig.xsd: XML Schema for Configuring Oracle XML DB"
> - "Configuring Default Namespace to Schema Location Mappings" on page 18-10 for more information about the `schemaLocation-mappings` parameter
> - "Configuring XML File Extensions" on page 18-12 for more information about the `xml-extensions` parameter

For examples of the usage of these parameters, see the configuration file `/xdbconfig.xml`, listed in and .

Table 24–1 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.

Table 24–1 (Cont.) Common Protocol Configuration Parameters

Parameter	Description
`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.
`xml-extensions`	Specifies the list of filename extensions that are treated as XML content by Oracle XML DB.
`session-pool-size`	Maximum number of sessions that are kept in the protocol server session pool
`/xdbconfig/sysconfig/call-timeout`	If a connection is idle for this time (in hundredths of a second), then the shared server serving the connection is freed up to serve other connections.
`session-timeout`	Time (in hundredths of a second) after which a session (and consequently the corresponding connection) 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 session timeout is not present in the configuration
`schemaLocation-mappings`	Specifies the default schema location for a given namespace. This is used if the instance XML document does not contain an explicit `xsi:schemaLocation` attribute.
`/xdbconfig/sysconfig/default-lock-timeout`	Time 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 24–2 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 24–3 Configuration Parameters Specific to HTTP/WebDAV (Except Servlet Parameters)

Parameter	Description
`http-port`	Port on which HTTP/WebDAV server listens
`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.
`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 is a resource in the container with any of these names. If so, then the contents of that file are sent, instead of a list of resources in the container.
`default-url-charset`	The character set in which an HTTP protocol server assumes incoming URL is encoded when it is not encoded in UTF-8 or the request's Content-Type field Charset parameter.

Interaction with Oracle XML DB File System Resources

The protocol specifications, RFC 959 (FTP), RFC 2616 (HTTP), and RFC 2518 (WebDAV) implicitly assume an abstract, hierarchical file system on the server side. This is mapped to the Oracle XML DB hierarchical repository. Oracle XML DB repository provides features such as:

- Name resolution
- Access control list (ACL)-based security. ACL is a list of access control entries that determine which principals have access to a given resource or resources. See also Chapter 23, "Oracle XML DB Resource 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://www.ietf.org/rfc/rfc959.txt`
- `http://www.ietf.org/rfc/rfc2616.txt`
- `http://www.ietf.org/rfc/rfc2518.txt`

Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents

Oracle XML DB protocol server enhances the protocols by always checking if XML documents being inserted are based on XML schemas registered in the repository.

- If the incoming XML document specifies an XML schema, then the Oracle XML DB storage to use is decided by that XML schema. This functionality comes in handy when you must 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, then it is stored as a binary document.

Event-Based Logging

In certain cases, it may be useful to log the requests received and responses sent by a protocol server. This can be achieved by setting event number `31098` to level `2`. To set this event, add the following line to your `init.ora` file and restart the database:

```
event="31098 trace name context forever, level 2"
```

Using FTP and Oracle XML DB Protocol Server

The following sections describe FTP features supported by Oracle XML DB.

Oracle XML DB Protocol Server: FTP Features

File Transfer Protocol (FTP) is one of the oldest and most popular protocols on the net. FTP is specified in RFC959 and provides access to heterogeneous file systems in a uniform manner. FTP works by providing well defined commands 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 or directories are downloaded and browsed, and then the connection is closed.

> **See Also:** RFC 959: FTP Protocol Specification— `http://www.ietf.org/rfc/rfc959.txt`

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.

Controlling Character Sets for FTP

Oracle Database supports two FTP quote commands to control character sets for different purposes: `set_nls_locale` and `set_charset`.

- **set_nls_locale**

  ```
  quote set_nls_locale {<charset_name> | NULL}
  ```

  This command is used to control the encoding of the file and directory names specified by the users in the FTP commands. It also controls the encoding of the file and directory names in the response returned to the users. Only IANA character set names can be specified for this parameter. If `nls_locale` is set to `NULL` or not set then it is defaulted to the database character set.

- **set_charset**

  ```
  quote set_charset  {<charset_name> | NULL}
  ```

  This command is used to specify the character set of the data to be sent to the server. This parameter, if defined, overrides the Byte Order Mark (BOM) and the encoding declaration inside the document. The keyword `NULL` is used to unset the `charset` parameter. The BOM is a signature to indicate the order of the following stream of bytes defined in the Unicode Standard.

  The algorithm used to determine the character encoding of incoming data is as follows:

  - The `charset` parameter value, if it is not `NULL`, determines the character set.
  - If the `charset` parameter value is `NULL`, the MIME type of the data is evaluated.

    If the MIME type is `*/xml` then the character set is determined by the presence of the BOM and the encoding declaration inside the XML document. Otherwise, the database character set is used.
  - Text documents are assumed to be in the database character set if the `set_charset` command is not set. This parameter does not apply to binary files that are not text.

Handling Error 421. Modifying the FTP Session's Default Timeout Value

If you are frequently disconnected from the server and have to reconnect and traverse the entire directory before doing the next operation, you may need to modify the default timeout value for FTP sessions. If the session is idle for more than this period, it gets disconnected. You can increase the timeout value (default = 6000 centiseconds) by modifying the configuration document as follows and then restart the database:

Example 24–1 Modifying the FTP Session's Default Timeout Value

```
declare
newconfig XMLType;
begin
  select updatexml(
    dbms_xdb.cfg_get(),
'/xdbconfig/sysconfig/protocolconfig/ftpconfig/session-timeout/text()',
    123456789) into newconfig from dual;
  dbms_xdb.cfg_update(newconfig);
end;
/

commit;
```

Using HTTP and Oracle XML DB Protocol Server

Oracle XML DB implements HyperText Transfer Protocol (HTTP), HTTP 1.1 as defined in RFC2616 specification.

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 exception of the following optional features:

- gzip and compress transfer encodings
- byte-range headers
- The TRACE method (used for proxy error debugging)
- Cache-Control directives (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—http://www.ietf.org/rfc/rfc2616.txt

Non-Supported HTTP Features

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 action. It should be compiled, and loaded into the database. Finally, the servlet name must be associated with a pattern, which can be an extension such as `*.jsp` or a path name such as `/a/b/c` or `/sys/*`, as described in Java servlet application program interface (API) version 2.2.

While processing an HTTP request, the path name for the request is matched with the registered patterns. If there is a match, then the protocol server invokes the corresponding servlet with the appropriate initialization parameters. For Java servlets, the existing Java Virtual Machine (JVM) infrastructure is used. This starts the JVM if need be, which in turn 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 25, "Writing Oracle XML DB Applications in Java"

Sending Multibyte Data From a Client

When a client sends multibyte data in a URL, RFC 2718 specifies that the client should send the URL using the %HH format where HH is the hexadecimal notation of the byte value in UTF-8 encoding. The following are URL examples that can be sent to XML DB in an HTTP or WebDAV context:

```
http://urltest/xyz%E3%81%82%E3%82%A2
http://%E3%81%82%E3%82%A2
http://%E3%81%82%E3%82%A2/abc%E3%81%86%E3%83%8F.xml
```

XML DB processes the requested URL, any URLs within an IF header, any URLs within the DESTINATION header, and any URLs in the REFERRED header that contains multibyte data.

The `default-url-charset` configuration parameter can be used to accept requests from some clients that use other non-conformant forms of URL of non-ASCII characters. If a request with non-ASCII characters fails, try setting this value to the native character set of the client environment. The character set used in such URL fields must be specified with an IANA charset name.

`default-url-charset` controls the encoding for non-conforming URLs. It is not required to be set unless a non-conforming client that does not send the `Content-Type` charset is used.

> **See Also:** RFC 2616: HTTP 1.1 Protocol Specification—`http://www.ietf.org/rfc/rfc2616.txt`

Non-Ascii Characters in URLs

Non-ascii characters appearing in URLs passed to an HTTP Server should be converted to `UTF-8` and escaped in the %HH format, where HH is the hexadecimal notation of the byte value. For flexibility, XML DB protocol server interprets the incoming URLs by testing whether it is encoded in one of the following character sets in the order presented here:

- UTF-8
- Charset parameter of the Content-Type field of the request if specified
- Character set if specified in the default-url-charset configuration parameter
- Character set of the database server

Controlling Character Sets for HTTP

The following sections describe how character sets are controlled for data transferred using HTTP.

Request Character Set The character set of the HTTP request body is determined with the following algorithm:

- The Content-Type header is evaluated. If the Content-Type header specifies a charset value, the specified charset is used.
- The MIME type of the document is evaluated as follows:
  - If the MIME type is "`*/xml`", the character set is determined as follows:

    - If a BOM is present `UTF-16` is used.

    - If an encoding declaration is present, the specified encoding is used.

    - If neither a BOM or encoding declaration is present, `UTF-8` is used.
  - If the MIME type is text, `ISO8859-1` is used.
  - If the MIME type is neither "`*/xml`" or `text`, the database character set is used.

Note that there is a difference between HTTP and SQL or FTP. For text documents the default is ISO8859-1 as specified by the IETF.org *RFC 2616: HTTP 1.1 Protocol Specification*.

Response Character Set

The response generated by XML DB HTTP/WebDAV Server is in the character set specified in the `Accept-Charset` field of the request. Accept-Charset can have a list of character sets. Based on the q-value Oracle XML DB chooses one that does not require conversion. This might not necessarily be the charset with the highest q-value. If Oracle XML DB cannot find one, then the conversion is based on the highest q-value.

Using WebDAV and Oracle XML DB

Web Distributed Authoring and Versioning (WebDAV) is 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—http://www.ietf.org/rfc/rfc2518.txt

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.
- The `COPY`, `MOVE` and `DELETE` methods comply with section 2 of the Internet Draft titled 'Binding Extensions to WebDAV'.
- 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 24–2.
4. Click **Next**.
5. Enter any name to identify this WebFolder
6. Click **Finish**.

You can now access Oracle XML DB repository just like you access any Windows folder.

Figure 24–2 Creating a WebFolder in Windows 2000

25

Writing Oracle XML DB Applications in Java

This chapter describes how to write Oracle XML DB applications in Java. It includes design guidelines for writing Java applications including servlets, and how to configure the Oracle XML DB servlets.

This chapter contains these topics:

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. An application server is a server designed to host applications and their environments, permitting server applications to run. A typical example is Oracle Application Server, 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. 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.

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 Oracle Database.

Servlets work better as the top level entry point into Oracle Database, and require using HTTP as the protocol to access Oracle Database.

Which Oracle XML DB APIs Are Available Inside and Outside the Database?

All Oracle XML DB application program interfaces (APIs) are available to applications running both in the server and outside the database, including:

- JDBC support for XMLType
- XMLType class
- Java DOM implementation

Design Guidelines: Java Inside or Outside the Database?

When choosing an architecture for writing Java Oracle XML DB applications, consider the following guidelines:

HTTP: Accessing Java Servlets or Directly Accessing XMLType Resources

If the downstream client wants to deal with XML in its textual representation, then 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 requirement 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, then 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 Oracle Application Server application server and access data in the database either using JDBC, or by using the `java.net.*` or similar APIs to get XML data through HTTP.

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.
- `RequestDispatcher` and associated methods are not supported.
- `HTTPServletRequest.getCookies()` method is not supported.
- Only one `ServletContext` (and one web-app) is currently supported.
- Stateful servlets (and thus the `HttpSession` class methods) are not supported. Servlets must maintain state in the database itself.

Configuring Oracle XML DB Servlets

Oracle XML DB servlets are configured using the `/xdbconfig.xml` file in 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 25–1 lists the XML elements defined for the servlet deployment descriptor by the Java Servlet specification, along with extension elements supported by Oracle XML DB.

Table 25–1 XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
`auth-method`	Java	no	Specifies an HTTP authentication method required for access	--
`charset`	Oracle	yes	Specifies a IANA character set name	For example: ISO8859, UTF8
`charset-mapping`	Oracle	yes	Specifies a mapping between a filename extension and a charset	--
`context-param`	Java	no	Specifies a parameter for a web application	Not yet supported
`description`	Java	yes	A string for describing a servlet or Web application	Supported for servlets
`display-name`	Java	yes	A string to display with a servlet or web app	Supported for servlets
`distributable`	Java	no	Indicates whether or not this servlet can function if all instances are not running in the same Java virtual machine	All servlets running in Oracle Database MUST be distributable.
`errnum`	Oracle	yes	Oracle error number	See *Oracle Database Error Messages*
`error-code`	Java	yes	HTTP 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	--

Table 25–1 (Cont.) XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
`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	--
`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

Table 25–1 (Cont.) XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
`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	--
`servlet-class`	Java	Yes	Specifies the classname for the Java servlet	--
`servlet-language`	Oracle	Yes	Specifies the programming language in which the servlet is written.	Either Java, C, or PL/SQL. Currently, only Java is supported for customer-defined servlets.
`servlet-mapping`	Java	Yes	Specifies a filename pattern with which to associate the servlet	All of the mappings defined by Java are supported
`servlet-name`	Java	Yes	String name for a servlet	Used by Servlet APIs
`servlet-schema`	Oracle	Yes	The Oracle Schema in which the Java class is loaded. If not specified, then the schema is searched using the default resolver specification.	If this is not specified, then the servlet must be loaded into the SYS schema to ensure that everyone can access it, or the default Java class resolver must be altered. Note that the servlet-schema is capitalized unless the value is quoted with double-quotes.
`session-config`	Java	No	Configuration information for an `HTTPSession`	`HTTPSession` is not supported
`session-timeout`	Java	No	Timeout for an HTTP session	`HTTPSession` is not supported
`small-icon`	Java	Yes	Small icon to associate with a servlet	--
`taglib`	Java	No	JSP tag library	JSPs currently not supported
`taglib-uri`	Java	No	URI for JSP tag library description file relative to the web.xml file	JSPs currently not supported
`taglib-location`	Java	No	Path name relative to the root of the web application where the tag library is stored	JSPs currently not supported
`url-pattern`	Java	Yes	URL pattern to associate with a servlet	See Section 10 of Java Servlet 2.2 spec

Table 25–1 (Cont.) XML Elements Defined for Servlet Deployment Descriptors

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`

Notes:

- **Note 1**: The following parameters defined for the `web.xml` file by Java are usable only by J2EE-compliant Enterprise Java Bean containers, and are not required for Java Servlet Containers that do not support a full J2EE environment: *env-entry, env-entry-name, env-entry-value, env-entry-type, ejb-ref, ejb-ref-type, home, remote, ejb-link, resource-ref, res-ref-name, res-type, res-auth*
- **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). An ACL is a list of access control entries that determines which principals have access to a given resource or resources. A future release will support using a `web.xml` file to generate ACLs.

See Also: 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, then Oracle listener hands the connection to a shared server dispatcher.
2. When a new HTTP request arrives, the dispatcher wakes up a shared server.
3. The HTTP headers are parsed into appropriate structures.
4. The shared server attempts to allocate a database session from the 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, then 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, then Oracle XML DB attempts to authenticate the session as the ANONYMOUS user. If that database user account is locked, then no unauthenticated access is allowed.
7. The URL in the HTTP request is matched against the servlets in the `xdbconfig.xml` file, as specified by the Java Servlet 2.2 specification.

8. The XML DB Servlet Container is invoked in the Java VM inside Oracle. If the specified servlet has not been initialized yet, then the servlet is initialized.
9. The Servlet reads input from the `ServletInputStream`, and writes output to the `ServletOutputStream`, and returns from the `service()` method.
10. If no uncaught Oracle error occurred, then the session is put back into the session pool.

See Also: Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data"

The Session Pool and XML DB Servlets

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()`, that 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`, then 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 25–2 lists non-implemented Java Servlet 2.2 methods. In this release they result in runtime exceptions.

Table 25–2 Non-Implemented Java 2.2 Methods

Interface	Methods
`HttpServletRequest`	`getSession(), isRequestedSessionIdValid()`
`HttpSession`	ALL
`HttpSessionBindingListener`	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 25–1 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 Oracle 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 and the Import-Export Utility for loading 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:

26

Managing Oracle XML DB Using Oracle Enterprise Manager

This chapter describes how to use Oracle Enterprise Manager for managing Oracle XML DB, including configuring, creating, and managing repository resources. The repository source include database objects such as XML schemas and `XMLType` tables.

This chapter contains these topics:

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 Oracle Database, both Enterprise and Standard Editions. To run the Enterprise Manager version that supports Oracle XML DB functionality, use Oracle9*i* 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://xmlns.oracle.com/xdb/xdbconfig.xsd`
- Oracle inserts a default configuration document. It creates resource `/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, then 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, then these objects will have to be created and mapped manually to the database schema.

See Also:

- Chapter 3, "Using Oracle XML DB"
- Chapter 5, "XML Schema Storage and Query: The Basics"

For most cases, it is assumed that you have XML schema annotated with information to automatically generate object types and object tables. Hence Oracle Database, as part of XML schema registration, automatically generates these objects.

See Also: "Managing XML Schema and Related Database Objects" on page 26-19

Oracle XML DB is now ready for you to insert conforming XML documents.

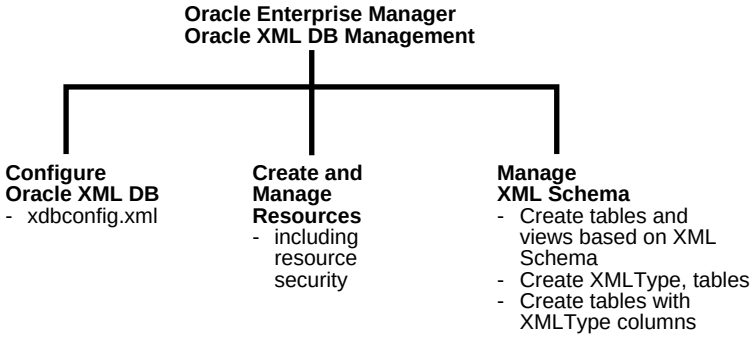
Oracle Enterprise Manager Oracle XML DB Features

With Oracle 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 26–1 and Figure 26–2.

Figure 26–1 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 parameters for this file. 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 26-4.

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 26-8.

Manage XML Schema and Related Database Objects

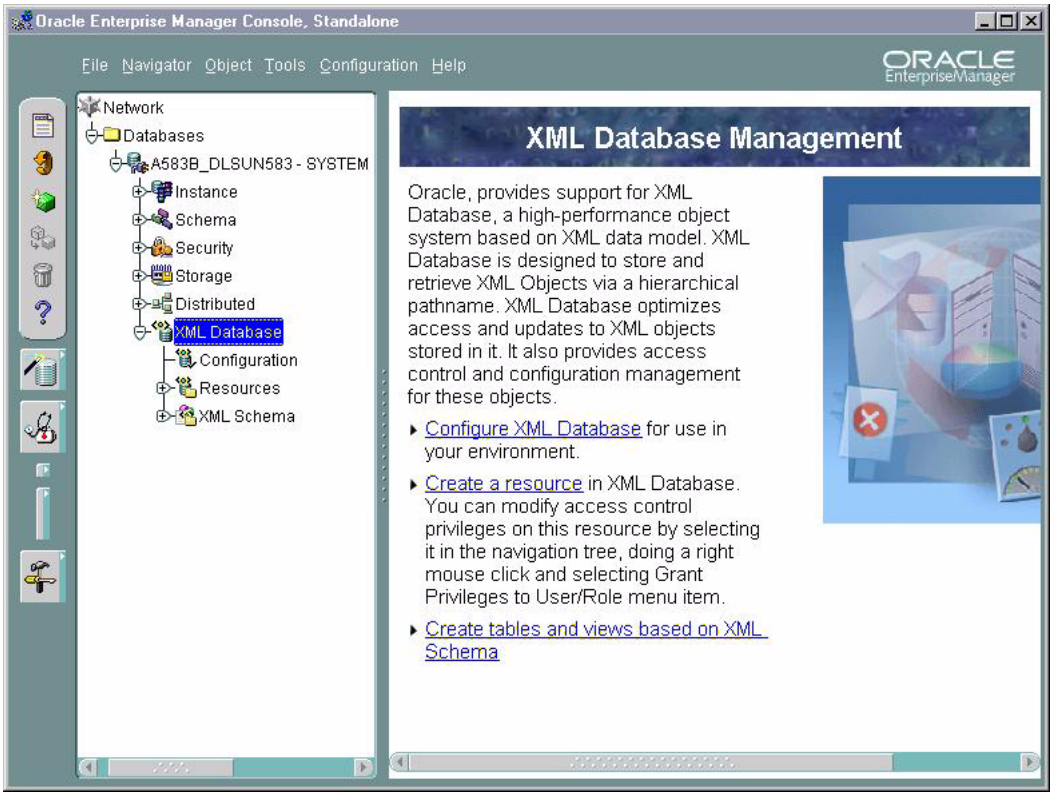
To access the XML schema management options, select the XML Database object in the Navigator and click **Create Table and Views Based on XML Schema** in the detail view. See "Managing XML Schema and Related Database Objects" on page 26-19.

You can register, delete, view, generate (that is, reverse engineer) an XML schema and see its dependencies. You can also view an XML schema 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-dependent objects, such as tables, views, indexes, object types, array types, and table types.

Figure 26–2 Enterprise Manager Console: XML Database Management Window



The Enterprise Manager Console for Oracle XML DB

See Figure 26–2. 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 must perform.

Hierarchical Navigation Tree: Navigator

Use the left-hand Navigator, to select the Oracle XML DB resources and database objects you must 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. ACL is a list of access control entries that determines which principals have access to a given resource or resources.

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://xmlns.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 26–3 and See Figure 26–4.

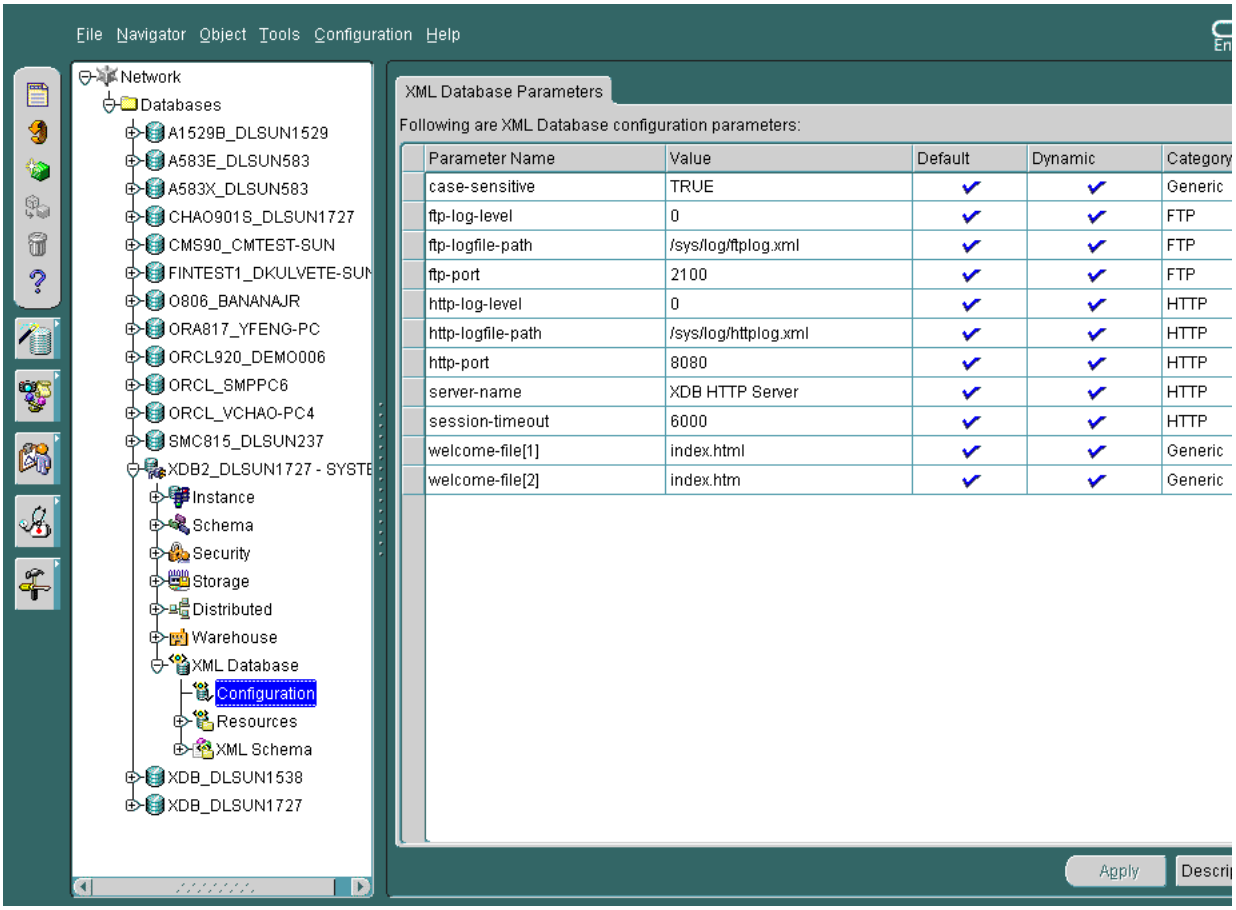
The **XML Database Parameters** page displays a list of configuration parameters for the current XML database. You can also access this page from the XML Database Management main window > Configure XML Database.

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 **Apply** 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 **Description** at the bottom of the page. A text **Description** text box appears that describes in greater detail the parameter you selected. You can close the Description box by clicking Description again.

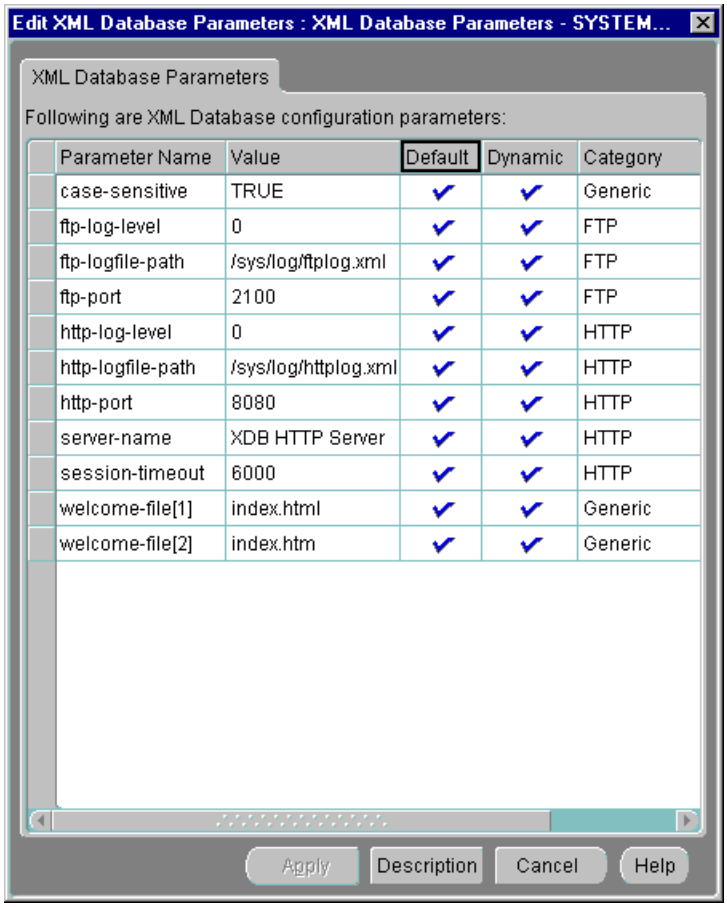
Figure 26–3 Enterprise Manager Console: Configuring Oracle XML DB



Because 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 26–4 Enterprise Manager Console: Edit XML Database 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 application program interface (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 26–5 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 26–6.

Figure 26–5 Enterprise Manager: Oracle XML DB Resources Tree Showing Resources Folder Selected

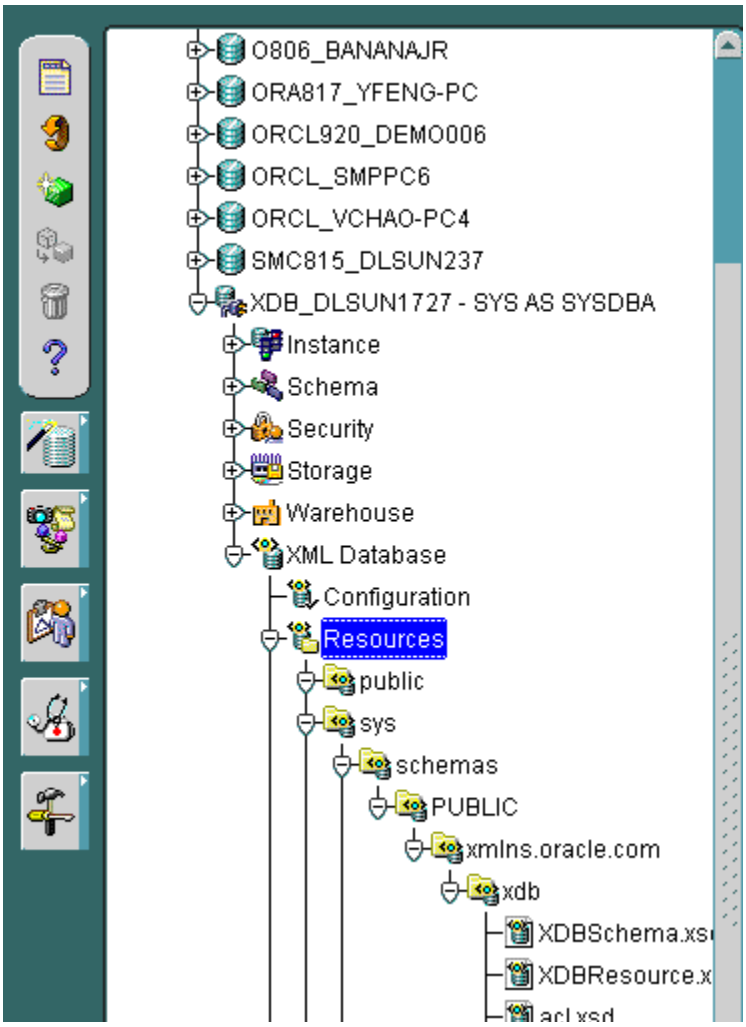
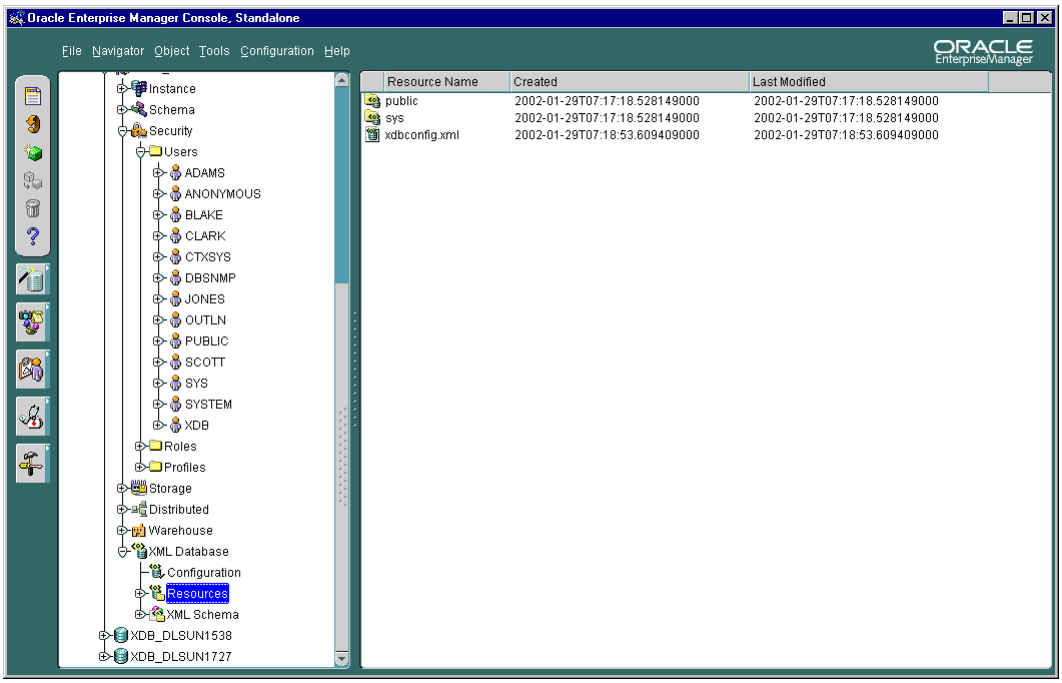


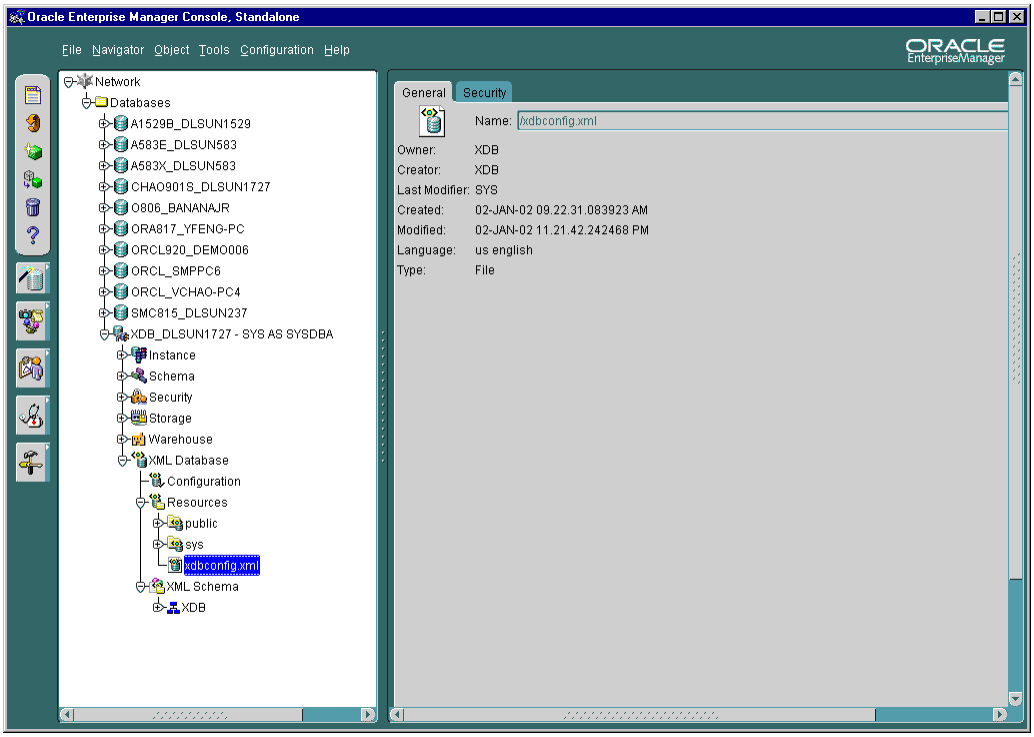
Figure 26–6 Enterprise Manager: Top Level Resources Under Root



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 26–7.

Figure 26–7 Enterprise Manager: Individual Oracle XML DB 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

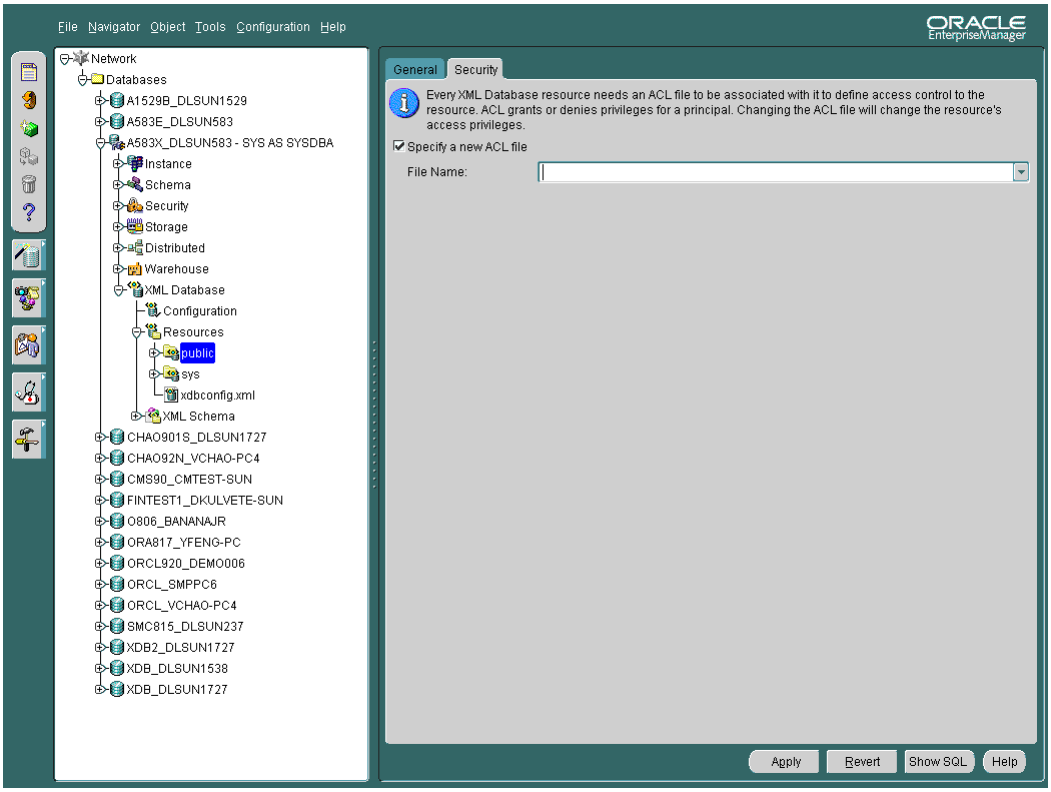
Oracle XML DB Resources Security page changes the ACL associated with the XML DB resource container or file. Use the ACL files to restrict access to all XML DB resources. When you change the ACL file for the resource, you change the access privileges for the resource file. See Figure 26–8.

To specify a new ACL file:

1. Click **Specify a new ACL File** and then choose a new ACL file from the drop down list in the **File Name** field.
2. Click **Apply** to apply the change.

3. Click **Revert** to abandon any changes you have made to the **File Name** field.

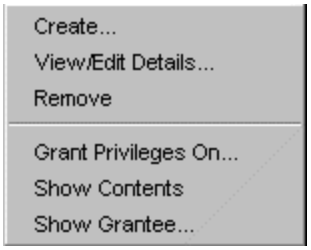
Figure 26–8 Enterprise Manager: Individual Oracle XML DB Resource - Security Page



Individual Resource Content Menu

Figure 26–9 shows the context-sensitive menu displayed when you select and right-click an individual Oracle XML DB resource object in the Navigator.

Figure 26–9 Enterprise Manager: When You Right-Click an Individual Resource...



You can perform the following Oracle XML DB tasks from the Enterprise Manager Content Menu:

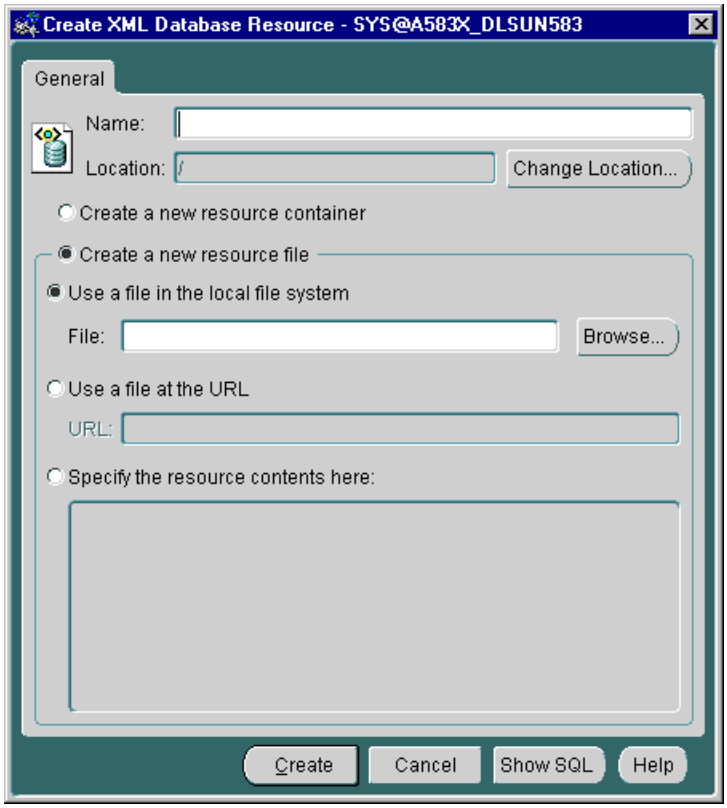
- Create Resource
- Grant Privileges On...
- Show Contents
- Show Grantee

Create Resource

Figure 26–10 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 26–10 Enterprise Manager: Create Oracle XML DB Resource Dialog Box



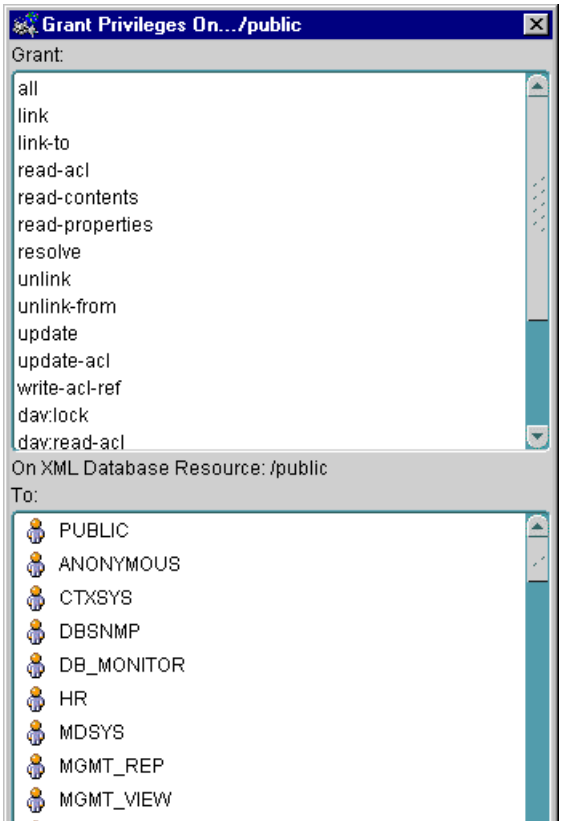
Grant Privileges On...

Figure 26–11 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 or group in the To: box at the bottom of the dialog page. Use the same procedure to select multiple users or roles to which to grant privileges.

See Also: "Enterprise Manager and Oracle XML DB: ACL Security" on page 26-15.

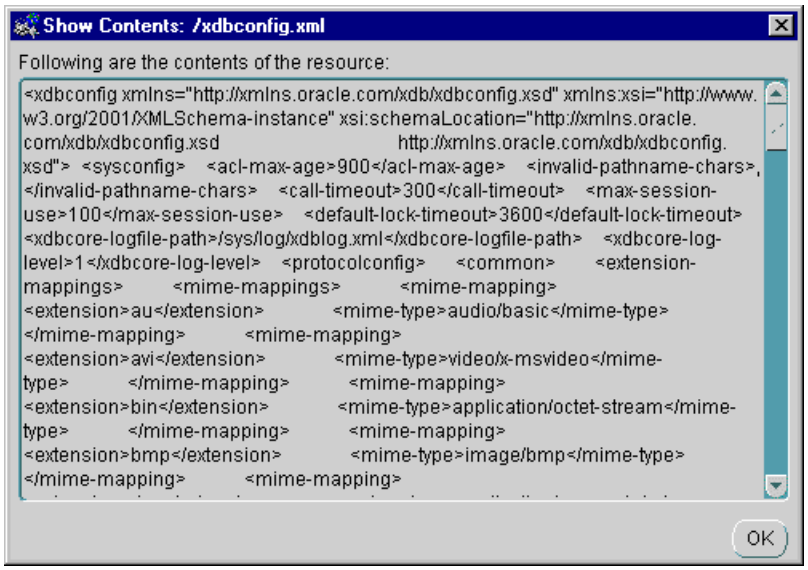
Figure 26–11 Enterprise Manager: Granting Privileges to Oracle XML DB Resources



Show Contents

Figure 26–12 is an example of a **Show Contents** dialog box. It displays the contents of the selected resource file.

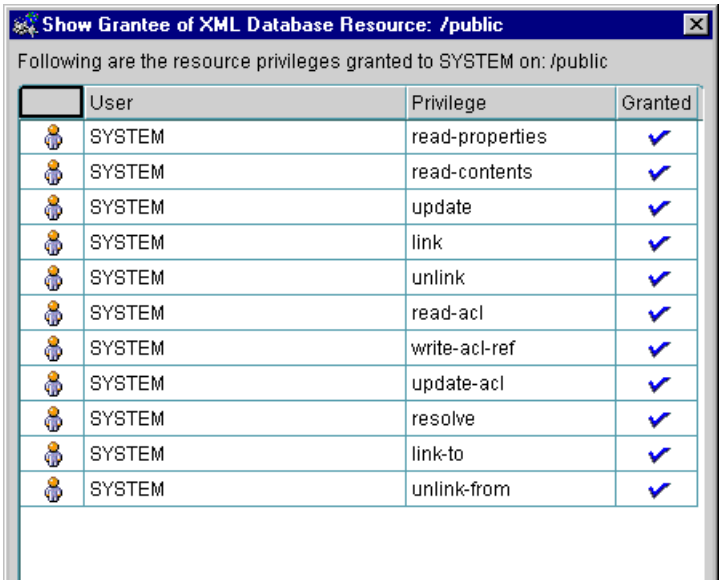
Figure 26–12 Enterprise Manager: Show Contents Menus of Individual Oracle XML DB Resource



Show Grantee

Figure 26–11 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 26–13 Enterprise Manager: Show Grantee of Oracle XML DB Resources



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 26–14. This user security option is described in more detail in "Granting and Revoking User Privileges with User > XML Tab" on page 26-16.
- ***To view or modify a resource 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 26–15.

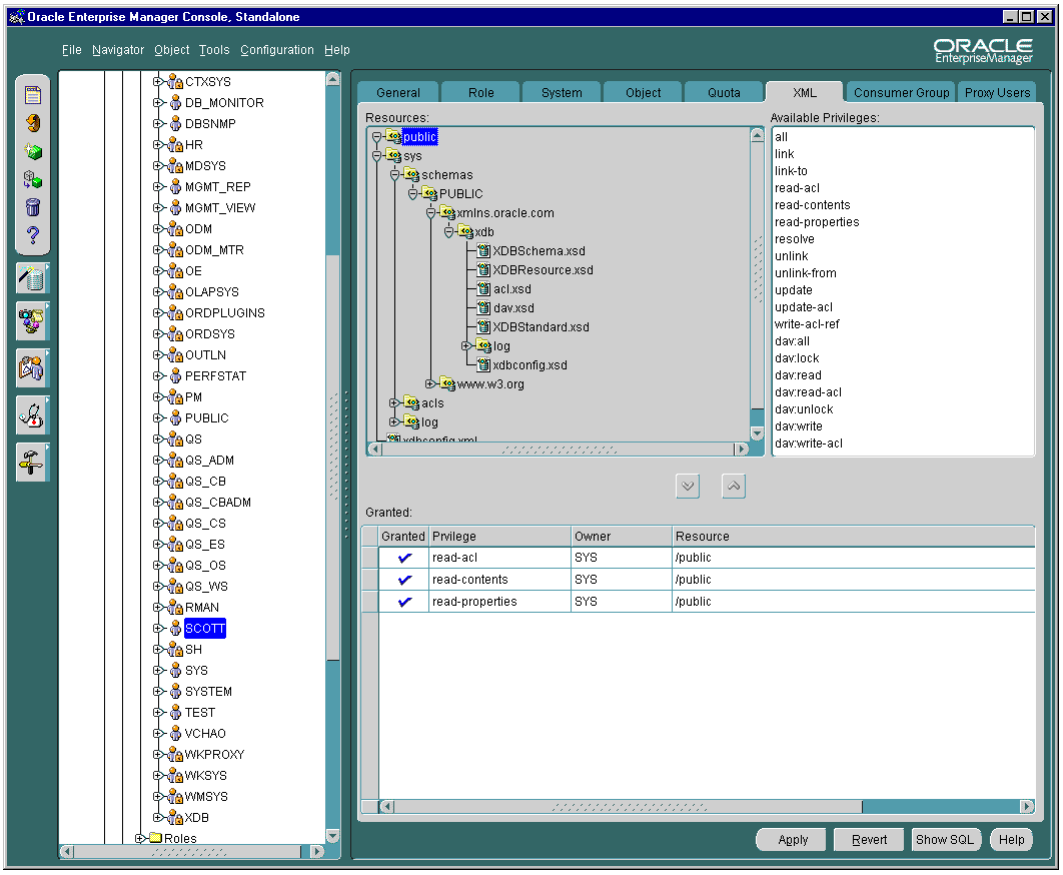
Granting and Revoking User Privileges with User > XML Tab

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 26–14 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 a privilege and then press the down arrow 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

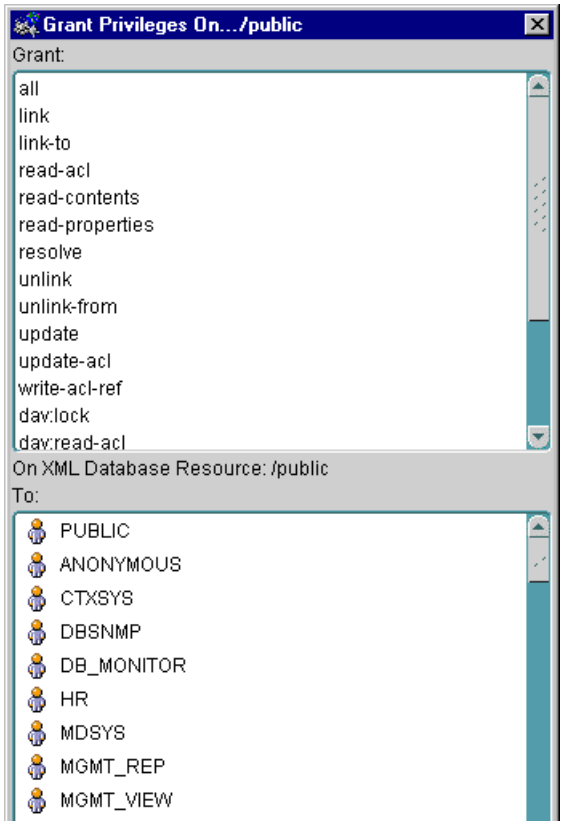
The **Granted** list displays all privileges granted to the user or role for the resource selected in the **Resources** list. You can revoke a privilege by highlighting it and clicking the up arrow to remove it.

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
  - link-to
  - unlink-from
  - resolve
  - dav:lock
  - dav:unlock
- **Aggregate Privileges**
  - dav:read (read-properties, read-contents, resolve)
  - dav:write (update, link, unlink, unlink-from)
  - dav:read-acl (read-acl)
  - dav:write-acl (write-acl-ref, update-acl)
  - dav:all (dav:read, dav:write, dav:read-acl, dav:write-acl, dav:look, dav:unlock)

Figure 26–15 Granting Privileges on a Resource



See Also: Chapter 23, "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 as shown in Figure 26–16 and Figure 26–17. With this page you can:

- Register an XML schema
- Create a structured storage based on an 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 26–16:

- ***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/xdb/XDBResource.xsd`

```
http://xmlns.oracle.com/xdb/XDBSchema.xsd
http://xmlns.oracle.com/xdb/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 26–16 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**.
- ***Dependent object owners***. Under each dependent object type, the tree lists the owner.
  - **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 26–16 Enterprise Manager Console: Navigating XML Schema

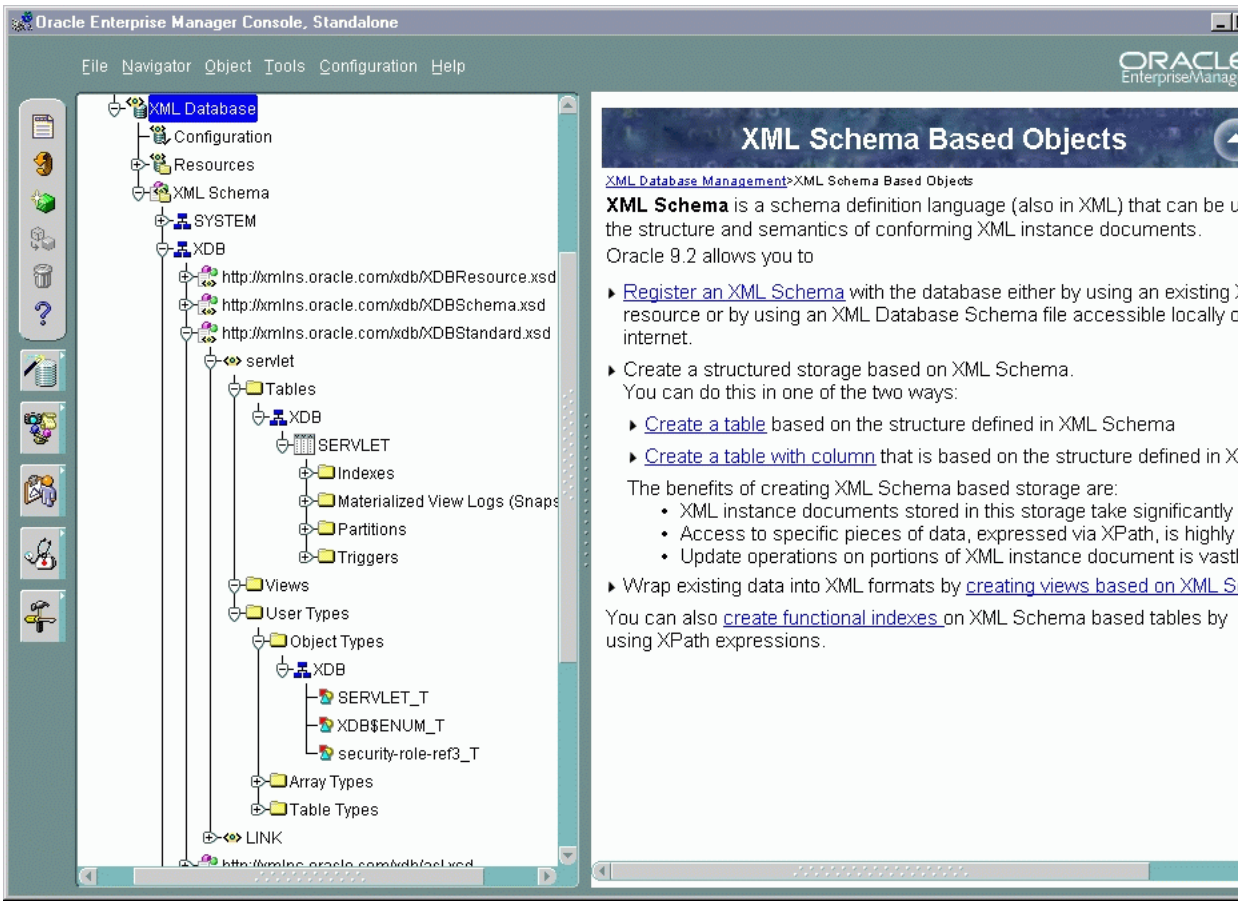
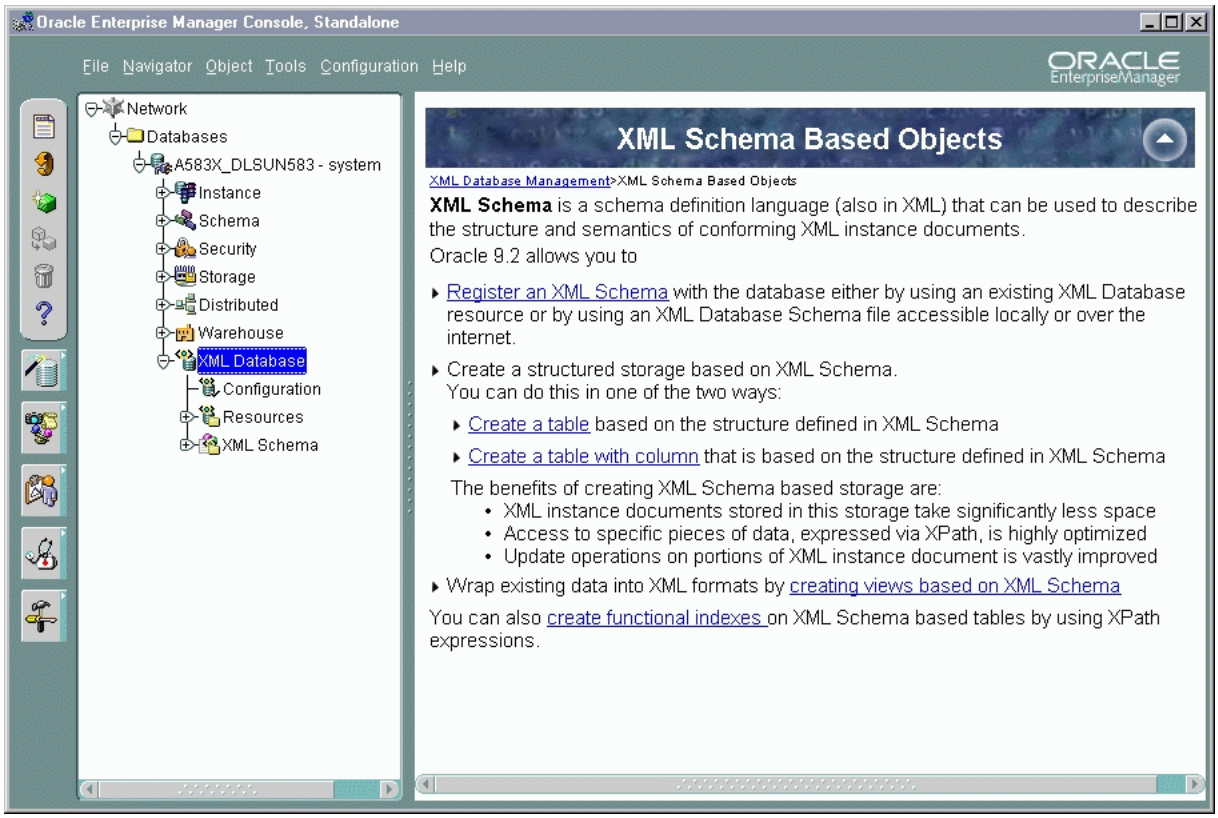


Figure 26–17 Enterprise Manager: Creating XML Schema-Based Objects



Registering an XML Schema

Registering an XML schema is one of the central, and often first, tasks before you use Oracle XML DB. XML schema are registered using `DBMS_XMLSCHEMA.registerSchema()`.

> **See Also:** Chapter 5, "XML Schema Storage and Query: The Basics"

Figure 26–18 shows you how to register an XML schema.

General Page

From the GENERAL page, enter 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 form 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 26–19. Press Create from either the General Tab or Options Tab to register this XML schema.

Figure 26–18 Enterprise Manager: Registering an XML Schema - General Page

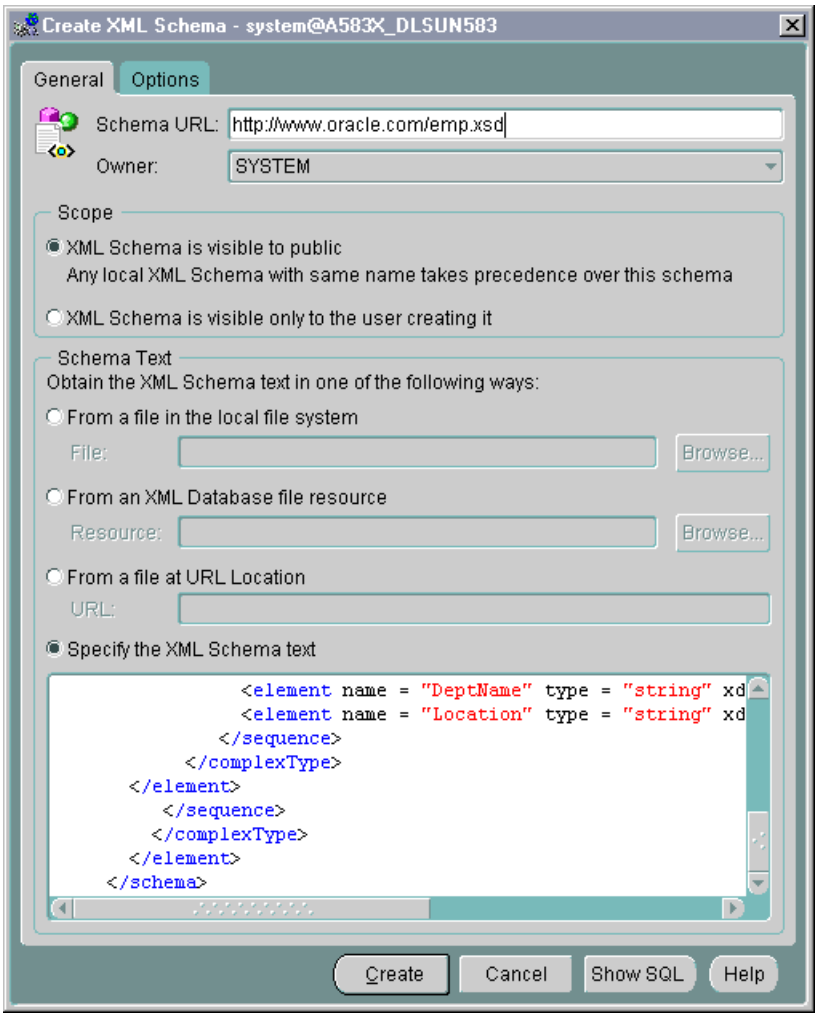
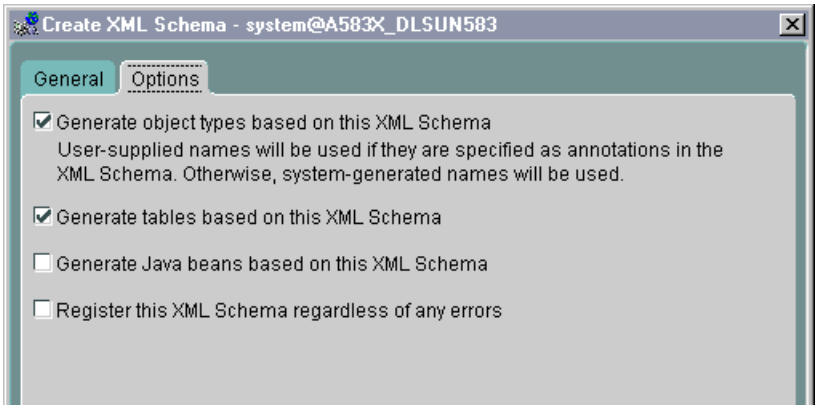


Figure 26–19 Enterprise Manager: Creating XML Schema - Options Page



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 26-24
- Creating Tables with XMLType Columns on page 26-26

Creating Views

To create an `XMLType` view, see "Creating a View Based on XML Schema" on page 26-28.

Creating Function-Based Indexes

To create function-based indexes see "Creating a Function-Based Index Based on XPath Expressions" on page 26-30.

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 26–21.

Figure 26–20 Enterprise Manager: Creating XMLType Tables-General Page

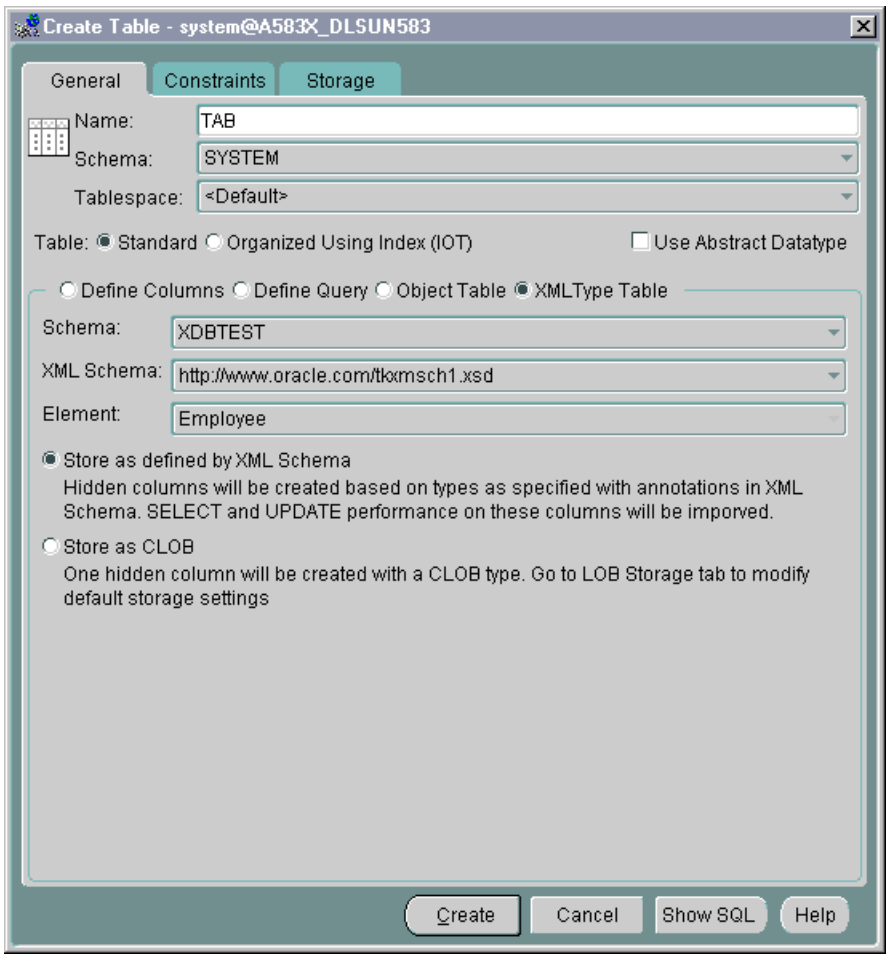
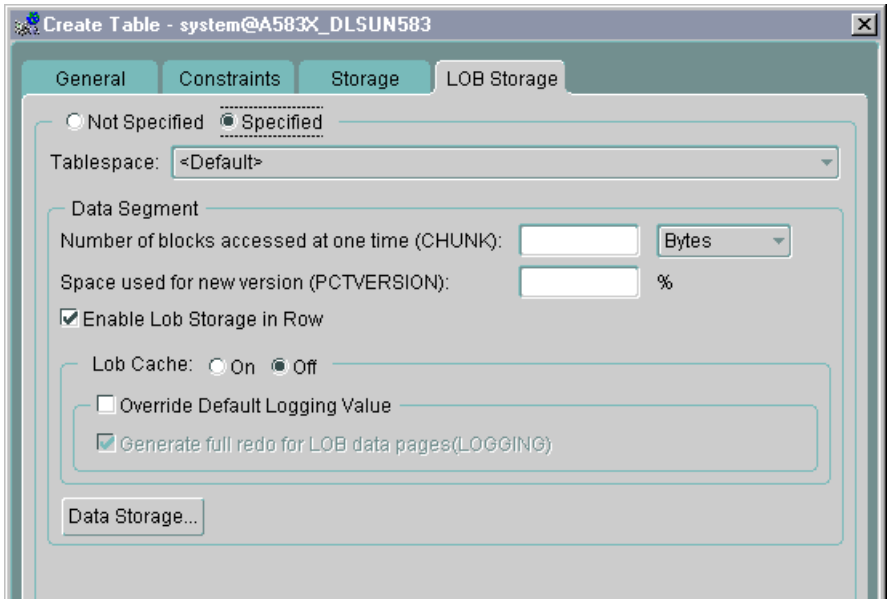


Figure 26–21 Enterprise Manager: Creating XMLType Tables - Specifying LOB Storage

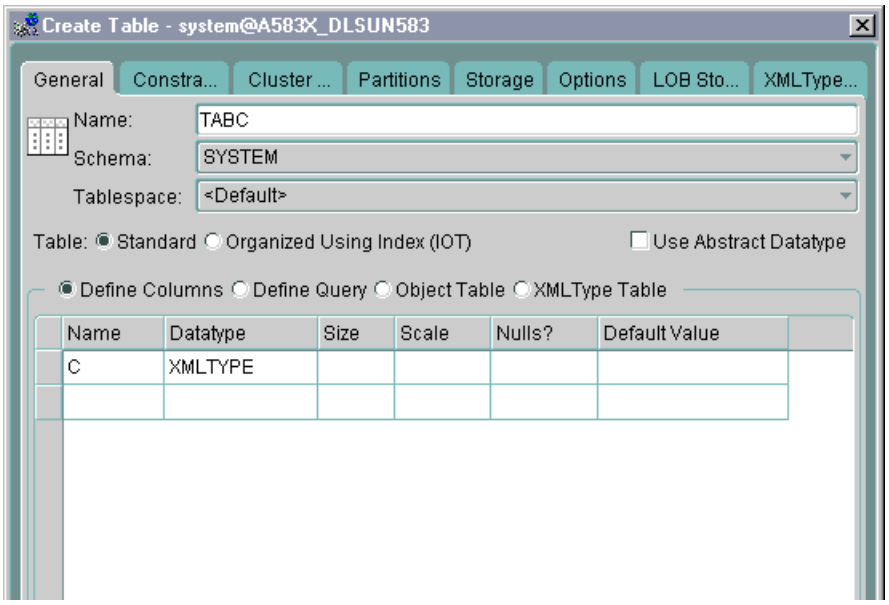


Creating Tables with XMLType Columns

See Figure 26–22 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 26–23.

Figure 26–22 Enterprise Manager: Creating Tables With XMLType Column - General Page



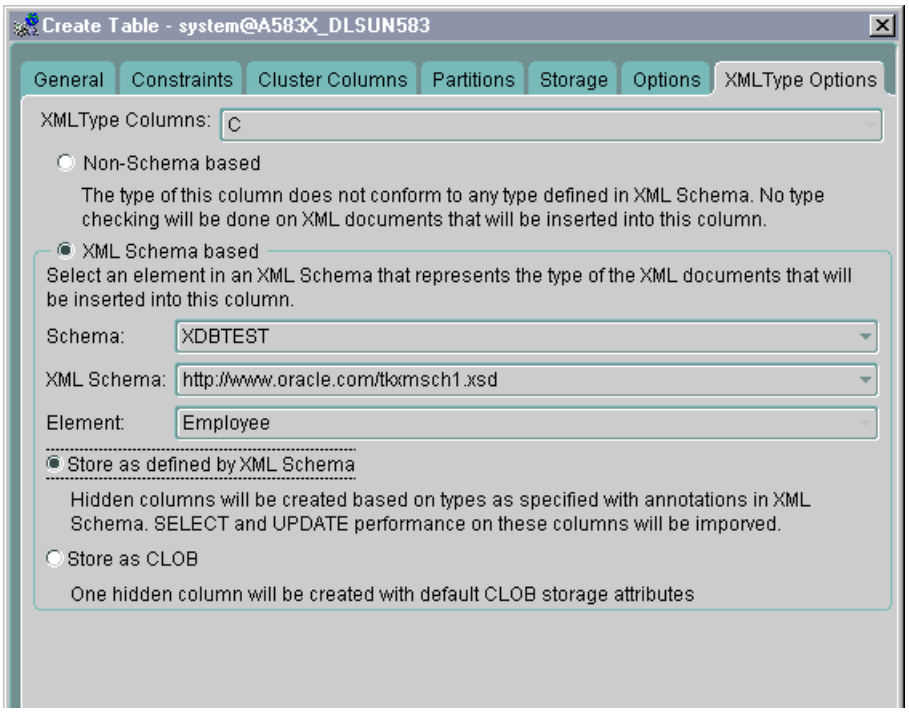
From this screen you can specify for a particular `XMLType` column, whether it is XML schema-based or non-schema-based.

If it is XML schema-based:

1. Under the "Schema" option, select the XML schema owner, from the drop-down list.
2. Under "XML Schema", select the actual XML schema from the drop-down list.
3. Under "Element", select the required element to form the `XMLType` column, from the drop-down list.
4. Specify the storage:
5. Store as defined by the XML schema.
6. 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 then you are not required to change the default settings.

Figure 26–23 Enterprise Manager: Creating Tables With XMLType Column - XMLType Options



Creating a View Based on XML Schema

Figure 26–24 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 26–25.

   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 16, "XMLType Views".

Figure 26–24 Enterprise Manager: Creating an XMLType View - General Page

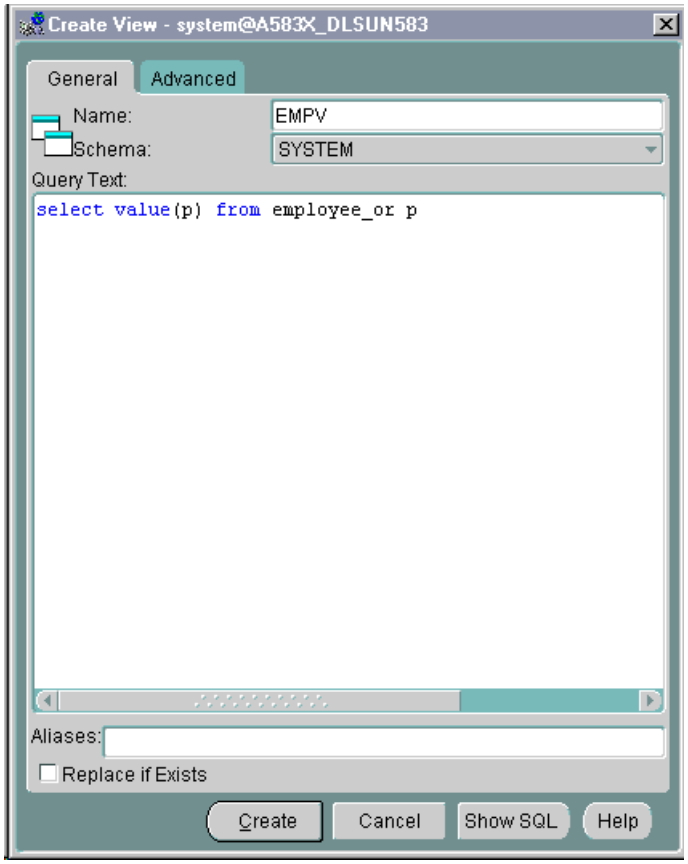
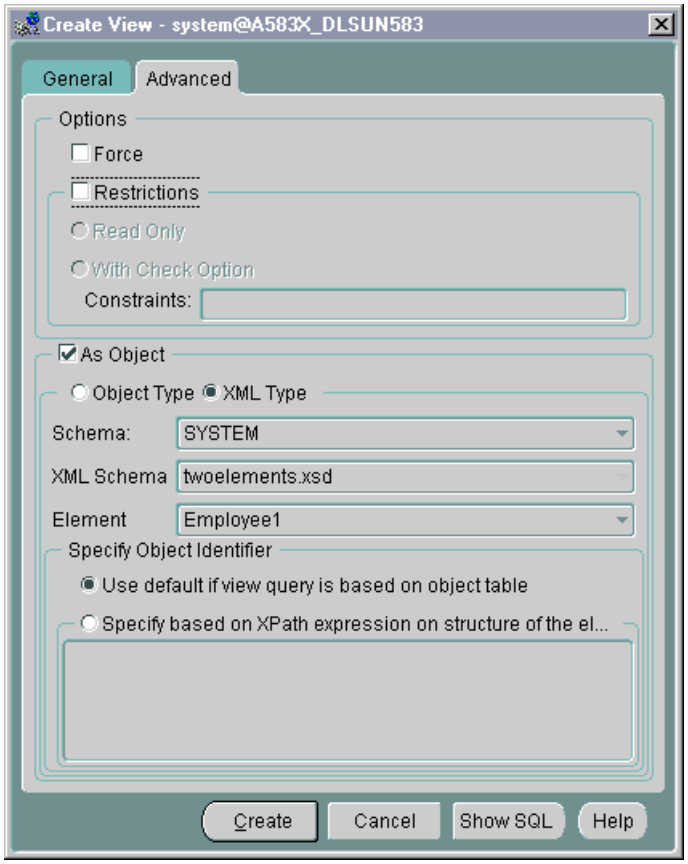


Figure 26–25 Enterprise Manager: Creating XMLType Views - Advanced Page



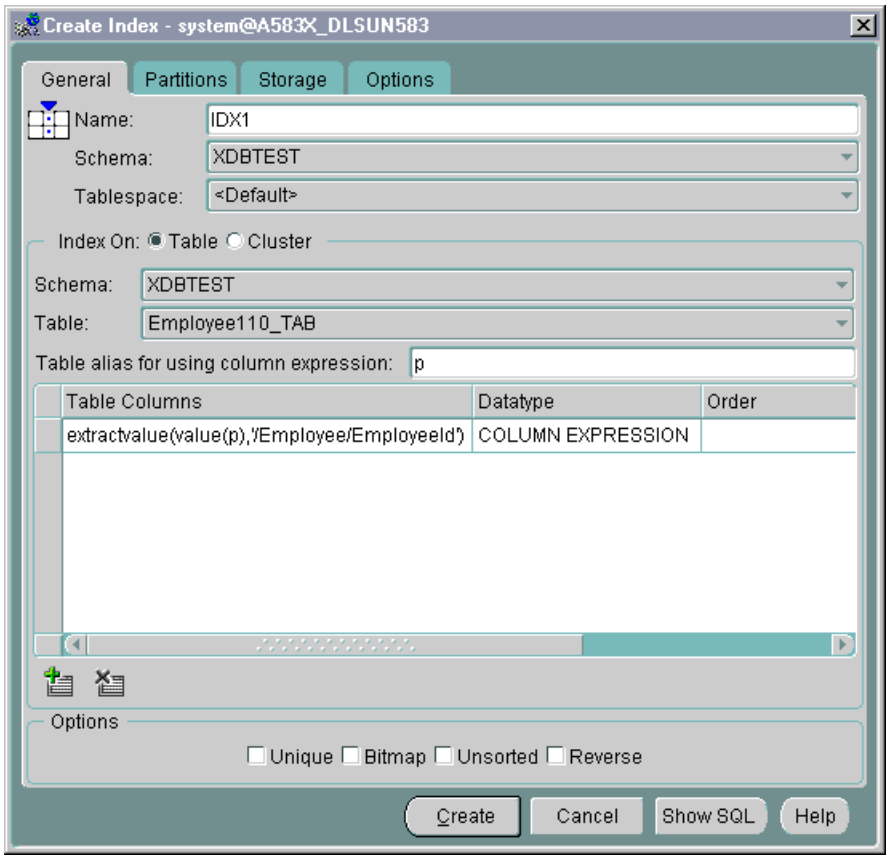
Creating a Function-Based Index Based on XPath Expressions

See Figure 26–26 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 are not required to change this.

Figure 26–26 Enterprise Manager: Creating a Function-Based Index Based on XPath Expressions



27

Loading XML Data into Oracle XML DB Using SQL*Loader

This chapter describes how to load XML data into Oracle XML DB with a focus on SQL*Loader.

This chapter contains these topics:

- Loading XMLType Data into Oracle Database
- Using SQL*Loader to Load XMLType Data
- Loading Very Large XML Documents into Oracle Database

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

Loading XMLType Data into Oracle Database

In Oracle9*i* release 1 (9.0.1) and higher, the Export-Import utility and SQL*Loader support `XMLType` as a column type. In Oracle Database 10*g*, SQL*Loader also supports loading `XMLType` *tables* and the loading is independent of the underlying storage. In other words, you can load XMLType data whether it is stored in LOBs or object-relationally. The `XMLType` data can be loaded by SQL*Loader using both the conventional and direct path methods.

> **Note:** One limitation is that SQL*Loader does not support direct path loading if the data involves inheritance.

> **See Also:** Chapter 28, "Importing and Exporting XMLType Tables" and *Oracle Database Utilities*

Restoration

In the current release, Oracle XML DB repository information is not exported when user data is exported. This means that both the resources and all information are not exported.

Using SQL*Loader to Load XMLType Data

XML columns are columns declared to be of type `XMLType`.

SQL*Loader treats `XMLType` columns and tables like any other object relational columns and tables. All methods described in the following sections for loading LOB

data from the primary datafile or from LOBFILE, also apply to loading XMLType columns and tables when the XMLType data is stored as a LOB.

See Also: *Oracle Database Utilities*

Note: You cannot specify a SQL string for LOB fields. This is true even if you specify `LOBFILE_spec`.

XMLType data can be present in a control file or in a LOB file. In this case, the LOB file name is present in the control file.

Because XMLType data can be quite large, SQL*Loader can load LOB data from either a primary datafile (in line with the rest of the data) or from LOBFILEs independent of how the data is stored. That is, the underlying storage can still be object-relational. This section addresses the following topics:

- Loading XMLType Data from a Primary Datafile
- Loading XMLType Data from an External LOBFILE (BFILE)
- Loading XMLType Data from LOBFILEs
- Loading XMLType Data from a Primary Datafile

Using SQL*Loader to Load XMLType Data in LOBs

To load internal LOBs, Binary Large Objects (BLOBs), Character Large Objects (CLOBs), and National Character Large Object (NCLOBs), or `XMLType` columns and tables from a primary datafile, use the following standard SQL*Loader formats:

- Predetermined size fields
- Delimited fields
- Length-value pair fields

These formats are described in the following sections and in more detail in *Oracle Database Utilities.*

Loading LOB Data in Predetermined Size Fields

This is a very fast and conceptually simple format to load LOBs.

Note: Because the LOBs you are loading 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.

Loading LOB Data in Delimited Fields

This format handles LOBs of different sizes within the same column (datafile field) without problem. However, this added flexibility can affect performance, because SQL*Loader must scan through the data, looking for the delimiter string.

As with single-character delimiters, when you specify string delimiters, you should consider the character set of the datafile. When the character set of the datafile is different than that of the control file, you can specify the delimiters in hexadecimal (that is, hexadecimal string). If the delimiters are specified in hexadecimal notation, then the specification must consist of characters that are valid in the character set of the input datafile. In contrast, if hexadecimal specification is not used, then the delimiter specification is considered to be in the client (that is, the control file)

character set. In this case, the delimiter is converted into the datafile character set before SQL*Loader searches for the delimiter in the datafile.

Loading 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)--`VARRAY`, `VARCHAR`, or `VARCHAR2` 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 additional attempts to read data from that file produce results equivalent to reading data from an empty field.

You should not specify the same LOBFILE as the source of two different fields. If you do so, then typically, the two fields will read the data independently.

Using SQL*Loader to Load XMLType Data Directly From the Control File

`XMLType` data can be loaded directly from the control file itself. In this release, SQL*Loader treats XMLType data like any other scalar type. For example, consider a table containing a NUMBER column followed by an XMLType column stored object-relationally. The control file used for this table can contain the value of the NUMBER column followed by the value of the XMLType instance.

SQL*Loader also accommodates XMLType instances that are very large. In this case you also have the option to load the data from a LOB file.

Loading Very Large XML Documents into Oracle Database

You can use SQL*Loader to load large amounts of XML data into Oracle Database.

> **See Also:** Chapter 3, "Using Oracle XML DB", "Loading Large XML Files into Oracle Database Using SQL*Loader" on page 3-12

Example 27–1 illustrates how to load XMLType data into Oracle Database.

Example 27–1 Loading Very Large XML Documents Into Oracle Database Using SQL*Loader

This example uses the control file, `load_data.ctl` to load `XMLType` data into table `foo`.

This code registers the XML schema, `person.xsd`, in Oracle XML DB and then creates table `foo`. You can also create the table within the XML schema registration process.

```
CREATE TYPE person_t AS OBJECT(name VARCHAR2(100), city VARCHAR2(100));
/
BEGIN
  -- delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/person.xsd', 4);
END;
/
BEGIN
  DBMS_XMLSCHEMA.registerschema('http://www.oracle.com/person.xsd',
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"' ||
           ' xmlns:per="http://www.oracle.com/person.xsd"' ||
           ' xmlns:xdb="http://xmlns.oracle.com/xdb"' ||
           ' elementFormDefault="qualified"' ||
           ' targetNamespace="http://www.oracle.com/person.xsd">' ||
     ' <element name="person" type="per:persontype"' ||
              ' xdb:SQLType="PERSON_T"/>' ||
       ' <complexType name="persontype" xdb:SQLType="PERSON_T">' ||
         ' <sequence>' ||
           ' <element name="name" type="string" xdb:SQLName="NAME"' ||
                   ' xdb:SQLType="VARCHAR2"/>' ||
           ' <element name="city" type="string" xdb:SQLName="CITY"' ||
                   ' xdb:SQLType="VARCHAR2"/>' ||
         ' </sequence>' ||
       ' </complexType>' ||
      '</schema>',
     TRUE,
     FALSE,
     FALSE);
END;
/
CREATE TABLE foo OF XMLType
  XMLSCHEMA "http://www.oracle.com/person.xsd" ELEMENT "person";
```

load_data.ctl

Here is the control file for loading `XMLType` data using the registered XML schema, person.xsd:

```
LOAD DATA
INFILE *
INTO TABLE foo TRUNCATE
XMLType(xmldata)
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(
xmldata
)
BEGINDATA
<person xmlns="http://www.oracle.com/person.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oracle.com/person.xsd
http://www.oracle.com/person.xsd"> <name> xyz name 2</name> </person>
```

Here is the SQL*Loader command for loading the XML data into Oracle Database:

```
sqlldr [username]/[password] load_data.ctl (optional: direct=y)
```

In the previous control file, the data was present in the control file itself, and a record spanned only one line. In the following example, the data is present in a separate file (`person.dat`) from the control file (`lod2.ctl`). Not only does the data file contain more than one row, but each row also spans multiple lines.

```
LOAD DATA
INFILE *
INTO TABLE foo TRUNCATE
xmltype(xmldata)
FIELDS
(
fill filler char(1),
xmldata LOBFILE (CONSTANT person.dat) TERMINATED BY '[XML]'
)
BEGINDATA
0
0
0
```

The three zeroes after `BEGINDATA` indicate that three records are present in the file `person.dat` (and they are terminated by `[XML]`). The contents of `person.dat` are as follows:

```
<person xmlns="http://www.oracle.com/person.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.oracle.com/person.xsd
                            http://www.oracle.com/person.xsd">
  <name>xyz name 2</name>
</person>
[XML]
<person xmlns="http://www.oracle.com/person.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.oracle.com/person.xsd
                            http://www.oracle.com/person.xsd">
  <name> xyz name 2</name>
</person>
[XML]
<person xmlns="http://www.oracle.com/person.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.oracle.com/person.xsd
                            http://www.oracle.com/person.xsd">
  <name>xyz name 2</name>
</person>
[XML]

sqlldr [username]/[password] lod2.ctl (optional: direct=y)
```

28

Importing and Exporting XMLType Tables

This chapter describes how you can import and export `XMLType` tables for use with Oracle XML DB.

It contains the following sections:

Overview of IMPORT/EXPORT Support in Oracle XML DB

Oracle XML DB supports `XMLType` tables and columns that can store XML data and be based on a registered XML schema. Tables storing XML schema-based or non-schema-based data can be imported and exported.

Resources and Foldering Do Not Fully Support IMPORT/EXPORT

Oracle XML DB also supports a foldering mechanism in the database that provides a file-system like paradigm to database data. This model uses path names and URIs to refer to data (referred to as *resources*) rather than table names, column names, and so on. This release however does not support this paradigm using IMPORT/EXPORT.

However for resources based on a registered XML schema, the actual `XMLType` tables storing the data can be exported and imported. This implies that only the XML data is exported while the relationship in the Oracle XML DB foldering hierarchy would be lost.

Non-XML Schema-Based XMLType Tables and Columns

`XMLType` tables and columns can be created without any XML schema specification in which case the XML data is stored in a CLOB.

Data from these tables can be exported and imported in a manner similar to LOB columns. The export dump file stores the actual XML text.

XML Schema-Based XMLType Tables

Oracle supports the export and import of XML schema-based `XMLType` tables. An `XMLType` table depends on the XML schema used to define it. Similarly the XML schema has dependencies on the SQL object types created or specified for it. Thus, exporting a user with XML schema-based `XMLType` tables, consists of the following steps:

1. ***Exporting SQL Types During XML Schema Registration.*** As a part of the XML schema registration process, SQL types can be created. These SQL types are exported as a part of `CREATE TYPE` statement along with their OIDs.
2. ***Exporting XML Schemas.*** After all the types are exported, XML schemas are exported as XML text as part of the `DBMS_XMLSCHEMA.REGISTERSCHEMA` statement. In this statement:
3. ***Exporting XML Tables.*** The next step is to export the tables. Export of each table consists of two steps:
    1. The table definition is exported as a part of the `CREATE TABLE` statement along with the table OID.
    2. The data in the table is exported as XML text. Note that data for out-of-line tables is not explicitly exported. It is exported as a part of the data for the parent table.

    Note: OCTs and nested tables are not exported separately. They are exported as parts of the parent table.

Guidelines for Exporting Hierarchy-Enabled Tables

The following describes guidelines for exporting hierarchy-enabled tables:

- The RLS policies and path-index triggers are not exported for hierarchy-enabled tables. This implies that when these tables are imported, they are not hierarchy-enabled.
- Hidden columns `ACLOID` and `OWNERID` are not exported for these tables. This is because in an imported database, the values of these columns could be different and hence should be re-initialized.

IMPORT/EXPORT Syntax and Examples

The `IMPORT/EXPORT` syntax and description are described in *Oracle Database Utilities*. This chapter includes additional guidelines and examples for using IMPORT/EXPORT with `XMLType` data.

IMPORT/EXPORT Example

Assumptions: The examples here assume that you are using a database with the following features:

- Two users, U1 and U2
- U1 has a registered local XML schema, SL1. This also created a default table TL1.
- U1 has a registered global XML schema, SG1. This also created a default table TG1.
- U2 has created table TG2 based on schema SG1.

User Level Import/Export

Example 28–1 Exporting XMLType Data

```
exp sytem/manager file=file1 owner=U1
```

This exports the following:

- Any types that were generated during schema registration of schemas SL1 and SG1.
- Schemas SL1 and SG1
- Tables TL1 and TG1 and any other tables that were generated during schema registration of schemas SL1 and SG1.
- Any data in any of the preceding tables.

Example 28–2 Exporting XMLType Tables

```
exp sytem/manager file=file2 owner=U2
```

This exports the following:

- Table TG2 and any other tables that were generated during creation of TG2.
- Any data in any of the preceding tables.

> **Note:** This does not export Schema SG1 or any types that were created during the registration of schema SG1.

Example 28–3 Importing Data from a File

```
imp system/manager file=file1 fromuser=U1 touser=newuser
```

This imports all the data in file1.dmp to schema newuser.

Import fails if the FROMUSER object types and object tables already exist on the target system. See "Considerations When Importing Database Objects" in *Database Utilities*.

Table Mode Export

An `XMLType` table has a dependency on the XML schema that was used to define it. Similarly the XML schema has dependencies on the SQL object types created or specified for it. Importing an `XMLType` table requires the existence of the XML schema and the SQL object types. When a TABLE mode export is used, only the table related metadata and data are exported. To be able to import this data successfully, the user needs to ensure that both the XML schema and object types have been created.

Example 28–4 Exporting XML Data in TABLE Mode

```
exp SYSTEM/MANAGER file=expdat.dmp tables=U1.TG1
```

This exports:

- Table TG1 and any nested tables that were generated during creation of TG1.
- Any data in any of the preceding tables.

Note: This does not export schema SG1 or any types that were created during the registration of schema SG1.

Example 28–5 Importing XML Data in TABLE Mode

```
imp SYSTEM/MANAGER file=expdat.dmp fromuser=U1 touser=U2 tables=TG1
```

This creates table TG1 for user U2 because U2 already has access to the globals schema SG1 and the types that it depends on.

Import fails if the FROMUSER object types and object tables already exist on the target system. See "Considerations When Importing Database Objects" in *Database Utilities*.

Metadata in Repository is Not Exported During a Full Database Export

Oracle XML DB stores the metadata (and the non-schema data) for the repository in the XML DB database user schema. Because Oracle does not support the export of the repository structure, these metadata tables and structures are not exported during a full database export.

The entire XML DB ("XDB") user schema is skipped during a full database export and any database objects owned by XML DB ("XDB") are not exported.

Importing and Exporting with Different Character Sets

As with other database objects, XML data is exported in the character set of the exporting server. During import, the data gets converted to the character set of the importing server.

Part VII

XML Data Exchange Using Oracle Streams Advanced Queuing

Part VII of this manual describes XML data exchange using Oracle Advanced Queuing (AQ) and the AQ `XMLType` support. Part VII contains the following chapter and Appendixes:

29

Exchanging XML Data With Oracle Streams AQ

This chapter describes how XML data can be exchanged using Oracle Streams Advanced Queuing (AQ).

This chapter contains these topics:

What Is Oracle Streams Advanced Queuing?

Oracle Streams Advanced Queuing (AQ) provides database integrated message queuing functionality:

- 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 Oracle Database brings the integrity, reliability, recoverability, scalability, performance, and security features of Oracle Database to message queuing. Integration with Oracle Database 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 Oracle Database, 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 29–1 shows an Oracle 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.

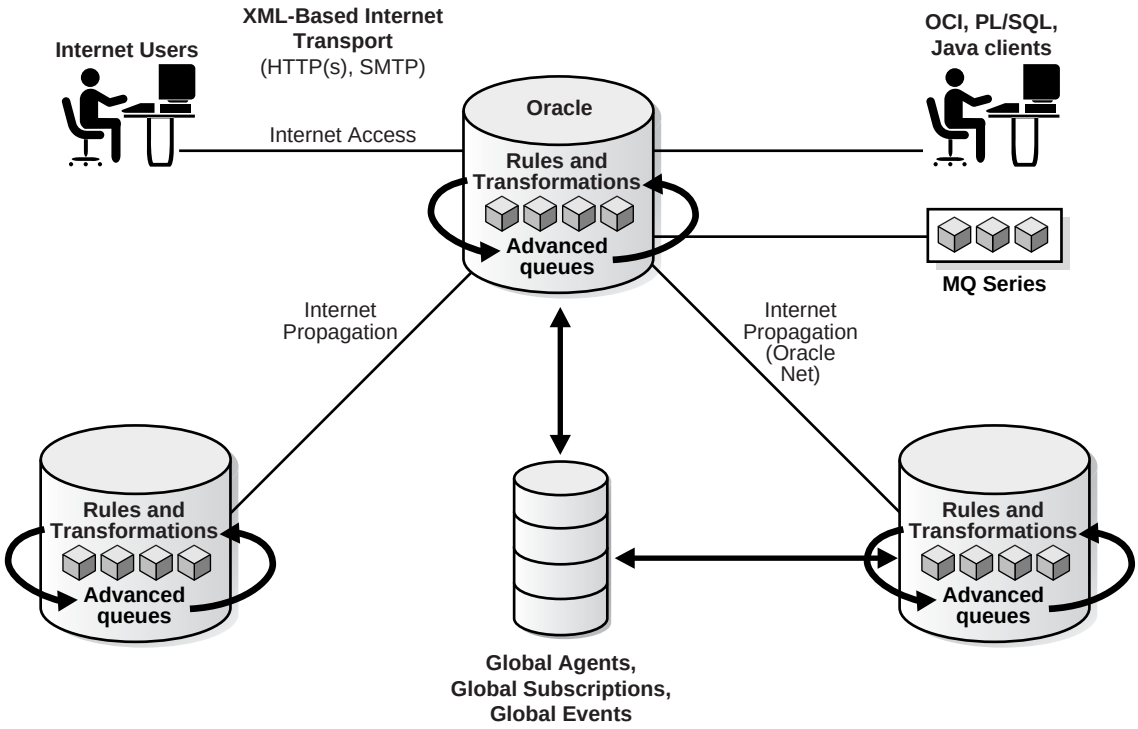
  Oracle uses this in its enterprise application integration products. XML messages are sent from applications to an Oracle AQ hub. This serves as a message server for any application that wants the message. Through this hub-and-spoke architecture, XML messages can be communicated asynchronously to multiple loosely coupled receiving applications.

Figure 29–1 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 or 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 29–1 Oracle Streams 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 carry out 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.

Oracle Streams and AQ

Oracle Streams (Streams) enables you to share data and events in a stream. The stream can propagate this information within a database or from one database to another. The stream routes specified information to specified destinations. This provides greater functionality and flexibility than traditional solutions for capturing and managing events, and sharing the events with other databases and applications.

Streams enables you to break the cycle of trading off one solution for another. It enable you to build and operate distributed enterprises and applications, data warehouses, and high availability solutions. You can use all the capabilities of Oracle Streams at the same time.

You can use Streams to:

- ***Capture changes at a database.*** You can configure a background capture process to capture changes made to tables, database schemas, or the entire database. A capture process captures changes from the redo log and formats each captured change into a logical change record (LCR). The database where changes are generated in the redo log is called the source database.
- ***Enqueue events into a queue.*** Two types of events may be staged in a Streams queue: LCRs and user messages. A capture process enqueues LCR events into a queue that you specify. The queue can then share the LCR events within the same database or with other databases. You can also enqueue user events explicitly with a user application. These explicitly enqueued events can be LCRs or user messages.
- ***Propagate events from one queue to another.*** These queues may be in the same database or in different databases.
- ***Dequeue events.*** A background apply process can dequeue events. You can also dequeue events explicitly with a user application.
- ***Apply events at a database.*** You can configure an apply process to apply all of the events in a queue or only the events that you specify. You can also configure an apply process to call your own PL/SQL subprograms to process events.

  The database where LCR events are applied and other types of events are processed is called the destination database. In some configurations, the source database and the destination database may be the same.

Streams Message Queuing

Streams allows user applications to:

- Enqueue messages of different types
- Propagate messages are ready for consumption
- Dequeue messages at the destination database

Streams introduces a new type of queue that stages messages of `SYS.AnyData` type. Messages of almost any type can be wrapped in a `SYS.AnyData` wrapper and staged in `SYS.AnyData` queues. Streams interoperates with Advanced Queuing (AQ), which supports all the standard features of message queuing systems, including multiconsumer queues, publishing and subscribing, content-based routing, internet propagation, transformations, and gateways to other messaging subsystems.

> **See Also:** *Oracle Streams Concepts and Administration,* and its Appendix, Appendix A, "XML Schema for LCRs".

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`

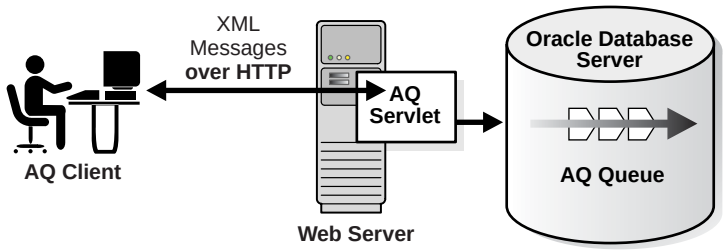
> **See Also:** *Oracle Streams Advanced Queuing User's Guide and Reference*

iDAP Architecture

Figure 29–2 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.
- Oracle Server/Database. Oracle Streams AQ servlet connects to Oracle Database to perform operations on your queues.

Figure 29–2 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 29–1 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.XMLTtype',
    compatible         => '8.1');
END;
BEGIN
  DBMS_AQADM.create_queue (
    queue_name   => 'OS_bookedorders_que',
    queue_table  => 'OS_orders_pr_mqtab');
END;
```

Because the representation of orders at the Overseas Shipping site is different from the representation of orders at the Order Entry site, 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 *Oracle Streams Advanced Queuing User's Guide and Reference* the section on Creating Transformations in Chapter 8.

Example 29–2 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.XMLTtype;
  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;
      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;
CREATE TYPE mypayload_type as OBJECT (xmlDataStream CLOB, dtd CLOB, pdf BLOB);
```

Guidelines for Using XML and Oracle Streams Advanced Queuing

This section describes guidelines for using XML and Oracle Streams Advanced Queuing.

Storing Oracle Streams AQ XML Messages with Many PDFs as One Record?

You can exchange XML documents between businesses using Oracle Streams Advanced Queuing, where each message received or sent includes an XML header, XML attachment (XML data stream), DTDs, and PDF files, and store the data in a database table, such as a `queuetable`. You can enqueue the messages into Oracle queue tables as one record or piece. Or you can enqueue the messages as multiple records, such as one record for XML data streams as CLOB type, one record for PDF files as RAW type, and so on. You can also then dequeue the messages.

You can achieve this in the following ways:

- By defining an object type with (CLOB, RAW, ...) attributes, and storing it as a single message.
- By using the AQ message grouping feature and storing it in multiple messages. Here the message properties are associated with a group. To use the message grouping feature, all messages must be the same payload type.

To specify the payload, first create an object type, for example:

```
CREATE TYPE mypayload_type as OBJECT (xmlDataStream CLOB, dtd CLOB, pdf BLOB);
```

then store it as a single message.

Adding New Recipients After Messages Are Enqueued

You can use the queue table to support message assignments. For example, when other businesses send messages to a specific company, they do not know who should be assigned to process the messages, but they know the messages are for Human Resources (HR) for example. Hence all messages will go to the HR supervisor. At this point, the message is enqueued in the queue table. The HR supervisor is the only recipient of this message, and the entire HR staff have been pre-defined as subscribers for this queue.

You cannot change the recipient list after the message is enqueued. If you do not specify a recipient list then subscribers can subscribe to the queue and dequeue the message. Here, new recipients must be subscribers to the queue. Otherwise, you have to dequeue the message and enqueue it again with new recipients.

Enqueuing and Dequeuing XML Messages?

Oracle Streams AQ supports enqueuing and dequeuing objects. These objects can have an attribute of type `XMLType` containing an XML document, as well as other interested "factored out" metadata attributes that may be useful to send along with the message. Refer to the latest AQ document, *Oracle Streams Advanced Queuing User's Guide and Reference* to get specific details and see more examples.

Parsing Messages with XML Content from Oracle Streams AQ Queues

You may want to parse messages with XML content, from an Oracle Streams AQ queue and then update tables and fields in an ODS (Operational Data Store), in other words you may want to retrieve and parse XML documents, then map specific fields to database tables and columns. To get metadata such as AQ enqueue or dequeue times, JMS header information, and so on, based on queries on certain XML tag values, the easiest way is by using Oracle XML Parser for Java and Java Stored Procedures in tandem with Oracle Streams AQ (inside Oracle Database).

- If you store XML as CLOBs then you can definitely search the XML using Oracle Text but this only helps you find a particular message that matches a criteria.
- To do aggregation operations over the metadata, view the metadata from existing relational tools, or use normal SQL predicates on the metadata, then having the data only stored as XML in a CLOB will not be good enough.

You can combine Oracle Text XML searching with some redundant metadata storage as factored out columns and use SQL that combines normal SQL predicates with the Oracle Text `CONTAINS()` clause to have the best of both of these options.

See Also: Chapter 9, "Full Text Search Over XML".

Preventing the Listener from Stopping Until the XML Document Is Processed

When receiving XML messages from clients as messages you may be required to process them as soon as they come in. Each XML document could take say about 15 seconds to process. For PL/SQL, one procedure starts the listener and dequeues the message and calls another procedure to process the XML document and the listener could be held up until the XML document is processed. Meanwhile messages accumulate in the queue.

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.

Oracle Database added PL/SQL callbacks in the Oracle Streams AQ notification framework. This allows you register a PL/SQL callback that is invoked asynchronously when a message shows up in a queue.

Using HTTPS with AQ

To send XML messages to suppliers using HTTPS and get a response back, you can use Oracle Streams AQ internet access functionality. You can enqueue and dequeue messages over HTTP(S) securely and transactionally using XML. For more details see:

`http://otn.oracle.com/products/aq/`

Storing XML in Oracle Streams AQ Message Payloads

You can store XML in Oracle Streams AQ message payloads natively other than having an ADT as the payload with `sys.xmltype` as part of the ADT. In Oracle9*i* release 2 (9.2) and higher you can create queues with payloads and attributes as `XMLType`.

Comparing iDAP and SOAP

iDAP is the SOAP specification for AQ operations. iDAP is the XML specification for Oracle Streams AQ operations. SOAP defines a generic mechanism to invoke a service. iDAP defines these mechanisms to perform AQ operations.

iDAP in addition has the following key properties not defined by SOAP:

- Transactional behavior. You can perform AQ operations in a transactional manner. You transaction can span multiple iDAP requests.
- Security. All the iDAP operations can be done only by authorized and authenticated users.

A

Installing and Configuring Oracle XML DB

This appendix describes how you install, reinstall, upgrade, manage, and configure Oracle XML DB.

This appendix contains these topics:

Installing Oracle XML DB

You are required to install Oracle XML DB under the following conditions:

- "Installing or Reinstalling Oracle XML DB From Scratch" on page A-1
- "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). If Oracle XML DB is already installed, complete the steps in "Reinstalling Oracle XML DB" on page A-3.

Installing a New Oracle XML DB With Database Configuration Assistant

Oracle XML DB is part of the seed database and installed by Database Configuration Assistant (DBCA) by default. No additional steps are required to install Oracle XML DB. However, if you select the Advanced database configuration, then you can configure Oracle XML DB tablespace and FTP, HTTP, and WebDAV port numbers.

By default DBCA performs the following tasks during installation:

- 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 stored in Oracle XML DB repository, including data stored using:

- SQL, for example using `RESOURCE_VIEW` and `PATH_VIEW`
- Protocols such as FTP, HTTP, and WebDAV

You can store data in tables outside this tablespace and access the data through the repository by having REFs to that data stored in the tables in this tablespace.

> **Caution:** The Oracle XML DB tablespace should not be dropped. If dropped, then it renders all repository data inaccessible.

Dynamic Protocol Registration Registers FTP and HTTP Services with Local Listener

Oracle XML DB installation, includes a dynamic protocol registration that registers FTP and HTTP services with the local Listener. You can perform start, stop, and query with `lsnrctl`. For example:

- start: `lsnrctl start`
- stop: `lsnrctl stop`
- query: `lsnrctl status`

Changing FTP or HTTP Port Numbers To change FTP or HTTP port numbers, update the tags `<ftp-port>` and `<http-port>` in file, `/xdbconfig.xml` in Oracle XML DB repository.

After updating the port numbers dynamic protocol registration automatically stops FTP/HTTP service on old port numbers and starts them on new port numbers if the local Listener is up. If local Listener is not up, restart the Listener after updating the port numbers.

> **See Also:** Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data" for a description of how to update `/xdbconfig.xml`

Postinstallation As explained in the previous section, Oracle XML DB uses dynamic protocol registration to setup FTP and HTTP Listener services with the local Listener. So, make certain that the Listener is up when accessing 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 Database Configuration Assistant

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_password> <XDB_TS_NAME> <TEMP_TS_NAME>
```

```
#Create the tables and views needed to run XML DB
```

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 Oracle9*i* release 2 (9.2.0) compatibility or higher, and Java Virtual Machine (JVM) is installed.

Postinstallation

After the manual installation, carry out these tasks:

1. Add the following dispatcher entry to the `init.ora` file:

   ```
   dispatchers="(PROTOCOL=TCP) (SERVICE=<sid>XDB)"
   ```

2. Restart the database and 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

> **Note:** All user data stored in Oracle XML DB repository is also lost when you drop user XDB!

To reinstall Oracle XML DB follow these steps:

1. Remove the dispatcher by removing the XML DB dispatcher entry from the `init.ora` file as follows:

   ```
   dispatchers="(PROTOCOL=TCP) (SERVICE=<sid>XDB)"
   ```

   If the server parameter file is used, run the following command when the instance is up and while logged in as SYS:

   ```
   ALTER SYSTEM RESET dispatchers scope=spfile sid='*';
   ```

2. Drop user XDB and tablespace XDB by connecting to SYS and running the following SQL script:

   ```
   @?/rdbms/admin/catnoqm.sql
   ALTER TABLESPACE <XDB_TS_NAME> offline;
   DROP TABLESPACE <XDB_TS_NAME> including contents;
   ```

3. Re-create tablespace XDB.
4. Execute `catnoqm.sql`.
5. Execute `catqm.sql`.
6. Execute `catxdbj.sql`.

7. Install Oracle XML DB manually as described in "Installing a New Oracle XML DB Manually Without Database Configuration Assistant" on page A-2.

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, then you must install Oracle XML DB manually as described in "Installing a New Oracle XML DB Manually Without Database Configuration Assistant" on page A-2.

Upgrading Oracle XML DB From Release 9.2 to 10*g* Release 1 (10.1)

All Oracle XML DB upgrade tasks are handled automatically when you use Database Upgrade Assistant to upgrade your database from any version of Oracle9*i* release 2 to Oracle Database 10*g* release 1 (10.1).

Note: In 10*g* release 1 (10.1), Oracle XML DB configuration is validated against the schema. Therefore, existing, invalid configuration documents must be validated before upgrading to the new release.

See Also: Oracle Database Upgrade Guide for details about using Database Upgrade Assistant

Privileges for Nested XMLType Tables When Upgrading to Oracle Database 10*g*

In Oracle9*i* release 2 (9.2), when you granted privileges on an `XMLType` table, they were not propagated to nested tables deeper than one level. In Oracle Database 10*g*, these privileges are propagated to all levels of nested tables.

When you upgrade from Oracle9*i* release 2 (9.2) to Oracle Database 10*g* with these nested tables, the corresponding nested tables (in Oracle Database 10*g*) will not have the right privileges propagated and users will not be able to access data from these tables. A typical error encountered is, `ORA-00942:table or view not found`. The workaround is to reexecute the original GRANT statement in Oracle Database 10*g*. This ensures that all privileges are propagated correctly.

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.

Oracle XML DB is managed through a configuration resource stored in Oracle XML DB repository, `/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 make an explicit call to refresh to the latest configuration.

See Also: Chapter 26, "Managing Oracle XML DB Using Oracle Enterprise Manager"

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 of an access control list (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>` Oracle Database 10*g* 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 24, "FTP, HTTP, and WebDAV Access to Repository Data", Table 24–1, Table 24–2, and Table 24–3, for a list of protocol configuration parameters
- Chapter 25, "Writing Oracle XML DB Applications in Java", "Configuring Oracle XML DB Servlets" on page 25-3

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>
      <acl-cache-size>32</acl-cache-size>
      <invalid-pathname-chars>,</invalid-pathname-chars>
      <case-sensitive>true</case-sensitive>
      <call-timeout>300</call-timeout>
      <max-link-queue>65536</max-link-queue>
      <max-session-use>100</max-session-use>
      <persistent-sessions>false</persistent-sessions>
      <default-lock-timeout>3600</default-lock-timeout>
      <xdbcore-logfile-path>/sys/log/xdblog.xml</xdbcore-logfile-path>
      <xdbcore-log-level>0</xdbcore-log-level>
      <resource-view-cache-size>1048576</resource-view-cache-size>

      <protocolconfig>
          <common>
             <extension-mappings>
                <mime-mappings>
                  <mime-mapping>
                    <extension>au</extension>
                    <mime-type>audio/basic</mime-type>
                  </mime-mapping>
                  <mime-mapping>
                    <extension>avi</extension>
                    <mime-type>video/x-msvideo</mime-type>
                  </mime-mapping>
                  <mime-mapping>
                    <extension>bin</extension>
                    <mime-type>application/octet-stream</mime-type>
                  </mime-mapping>
                </mime-mappings>

                <lang-mappings>
                  <lang-mapping>
```

```
          <extension>en</extension>
          <lang>english</lang>
        </lang-mapping>
      </lang-mappings>

      <charset-mappings>
      </charset-mappings>

      <encoding-mappings>
        <encoding-mapping>
          <extension>gzip</extension>
          <encoding>zip file</encoding>
        </encoding-mapping>
        <encoding-mapping>
          <extension>tar</extension>
          <encoding>tar file</encoding>
        </encoding-mapping>
      </encoding-mappings>
   </extension-mappings>

   <session-pool-size>50</session-pool-size>
   <session-timeout>6000</session-timeout>
</common>

<ftpconfig>
  <ftp-port>2100</ftp-port>
  <ftp-listener>local_listener</ftp-listener>
  <ftp-protocol>tcp</ftp-protocol>
  <logfile-path>/sys/log/ftplog.xml</logfile-path>
  <log-level>0</log-level>
  <session-timeout>6000</session-timeout>
  <buffer-size>8192</buffer-size>
</ftpconfig>

<httpconfig>
  <http-port>8080</http-port>
  <http-listener>local_listener</http-listener>
  <http-protocol>tcp</http-protocol>
  <max-http-headers>64</max-http-headers>
  <session-timeout>6000</session-timeout>
  <server-name>XDB HTTP Server</server-name>
  <max-header-size>16384</max-header-size>
  <max-request-body>2000000000</max-request-body>
  <logfile-path>/sys/log/httplog.xml</logfile-path>
  <log-level>0</log-level>
  <servlet-realm>Basic realm="XDB"</servlet-realm>
  <webappconfig>
    <welcome-file-list>
      <welcome-file>index.html</welcome-file>
      <welcome-file>index.htm</welcome-file>
    </welcome-file-list>
    <error-pages>
    </error-pages>
    <servletconfig>
      <servlet-mappings>
        <servlet-mapping>
          <servlet-pattern>/oradb/*</servlet-pattern>
          <servlet-name>DBURIServlet</servlet-name>
        </servlet-mapping>
      </servlet-mappings>
```

```
                <servlet-list>
                  <servlet>
                    <servlet-name>DBURIServlet</servlet-name>
                    <display-name>DBURI</display-name>
                    <servlet-language>C</servlet-language>
                    <description>Servlet for accessing DBURIs</description>
                    <security-role-ref>
                      <role-name>authenticatedUser</role-name>
                      <role-link>authenticatedUser</role-link>
                    </security-role-ref>
                  </servlet>
                </servlet-list>
              </servletconfig>
            </webappconfig>
          </httpconfig>
      </protocolconfig>
     <xdbcore-xobmem-bound>1024</xdbcore-xobmem-bound>
     <xdbcore-loadableunit-size>16</xdbcore-loadableunit-size>
    </sysconfig>

</xdbconfig>
```

Oracle XML DB Configuration API

The Oracle XML DB Configuration application program interface (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 Document Object Model (DOM) APIs for Java, PL/SQL, or C (OCI).

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
```

Update Configuration, cfg_update()

The `cfg_update()` function updates the configuration with a new one:

```
DBMS_XDB.CFG_UPDATE(newconfig SYS.XMLTYPE)
```

`cfg_update()` is auto-commit.

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, then you can use the following:

```
BEGIN
DBMS_XDB.CFG_UPDATE(UPDATEXML(UPDATEXML
    (DBMS_XDB.CFG_GET(),
        /xdbconfig/descendant::ftp-port/text()',   '2121'),
        '/xdbconfig/descendant::http-port/text()',
         19090'))
END;
```

```
/
```

If you have many parameters to update, then 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_XDB.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.

See Also: *Oracle XML API Reference*

B

XML Schema Primer

This appendix includes introductory information about the W3C XML Schema Recommendation.

This appendix contains these topics:

XML Schema and Oracle XML DB

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

> **See Also:** Chapter 1, "Introducing Oracle XML DB", "XML Schema" on page 1-11

Namespaces

Two different XML schemas can define on object, such as an element, attribute, complex type, simple type, and so on, with the same name. Because the two objects are in different XML schemas they cannot be treated as being the same item. This means

that an instance document must identify which XML schema a particular node is based on. The XML Namespace Recommendation defines a mechanism that accomplishes this.

An XML namespace is a collection of names identified by a URI reference. These are used in XML documents as element types and attribute names.

XML Schema and Namespaces

This section describes the basics of using XML schema and Namespaces.

XML Schema Can Specify a targetNamespace Attribute

The XML schema use the `targetNamespace` attribute to define the namespace associated with a given XML schema. The attribute is included in the definition of the 'XML schema' element. If an XML schema:

- Specifies a `targetNamespace`, all elements and types defined by the XML schema are associated with this namespace. This implies that any XML document containing these elements and types must identify which namespace they are associated with.
- Does not specify a `targetNamespace`, elements and types defined by the XML schema are associated with the `NULL` namespace.

XML Instance Documents Declare Which XML Schema to Use in Their Root Element

The XML Schema Recommendation defines a mechanism that allows an XML instance document to identify which XML schemas are required for processing or validating XML documents. The XML schemas in question are identified (in an XML instance document) on a namespace by namespace basis using attributes defined in the W3C XMLSchema-Instance namespace. To use this mechanism the instance XML document must declare the XMLSchema-instance namespace in their root element, as follows:

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

schemaLocation Attribute

Besides this, XML schemas that include a `targetNamespace` declaration are identified by also including a `schemaLocation` attribute in the root node of the instance XML documents.

The `schemaLocation` attribute contains one entry for each XML schema used. Each entry consists of a pair of values:

- The left hand side of each pair is the value of the `targetNamespace` attribute. In the preceding example, this is "xmlns:xxxx"
- The right hand side of each pair is a hint, typically in the form of a URL. In the preceding example, this is: `http://www.w3.org/2001/XML` It describes where to find the XML schema definition document. This hint is often referred to as the "Document Location Hint".

noNamespaceSchemaLocation Attribute

XML schemas that do not include a `targetNamespace` declaration are identified by including the `noNamespaceSchemaLocation` attribute in the root node of the instance document. The `noNamespaceSchemaLocation` attribute contains a hint, typically in the form of a URL, that describes where to find the XML schema document in question.

In the instance XML document. once the XMLSchema-instance namespace has been declared, it must identify the set of XML schemas required to process the document using the appropriate `schemaLocation` and `noNamespaceSchemaLocation` attributes.

Declaring and Identifying XML Schema Namespaces

Consider an XML schema with a defined root element `PurchaseOrder`. Assume that the XML schema does not declare a target namespace. The XML schema is registered under the following URL:

```
http://xmlns.oracle.com/demo/purchaseOrder.xsd
```

For an XML document to be recognized as an instance of this XML schema, specify the root element of the instance document as follows:

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

Registering an XML Schema

Before Oracle XML DB can make use of information in an XML schema, the XML schema must be registered with the database. You register an XML schema by calling the PL/SQL procedure `DBMS_XMLSCHEMA.register_schema()`.

The XML schema is registered under a URL. This URL a used internally as a unique key used to identify the XML schema. Oracle XML DB does not require access to the target of the URL when registering your XML schema, or when processing documents that conform to the XML schema.

Oracle XML DB assumes that any instance documents associated with the XML schema will provide the URL used to register the XML schema as the Document Location Hint.

Oracle XML DB Creates a Default Table

When an XML schema is registered with the database, a default table is created for each globally defined element declared in the XML schema. When an instance document is loaded in the Oracle XML DB repository, the content of the document will be stored in the Default Table. The default tables created by registering an XML schema are XMLType tables, that is, they are Object Tables, where each row in the table is represented as an instance of the XMLType data type.

Deriving an Object Model: Mapping the XML Schema Constructs to SQL Types

Oracle XML DB can also use the information contained in an XML schema to automatically derive an object model that allows XML content compliant with the XML schema to be decomposed and stored in the database as a set of objects. This is achieved by mapping the constructs defined by the XML schema directly into SQL types generated using the SQL 1999 Type framework that is part of Oracle Database.

Using the SQL 1999 type framework to manage XML provides several benefits:

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

- Capturing the XML schema objects as SQL types helps share the abstractions across schemas, and also across their SQL storage counterparts.
- It allows Oracle XML DB to support constructs defined by the XML schema standard that do not easily map directly into the conventional relational model.

Oracle XML DB and DOM Fidelity

Using SQL 1999 objects to persist XML allows Oracle XML DB to guarantee DOM fidelity. The Document Object Model (DOM), is a W3C standard that defines a set of platform- and language-neutral interfaces that allow a program to dynamically access and update the content, structure, and style of a document. To provide DOM fidelity Oracle XML DB ensures that a DOM generated from a document that has been shredded and stored in Oracle XML DB will be identical to a DOM generated from the original document.

Providing DOM Fidelity requires Oracle XML DB to preserve all information contained in an XML document. This includes maintaining the order in which elements appear within a collection and within a document as well as storing and retrieving out-of-band data, such as comments, processing instructions, and mixed text. By guaranteeing DOM fidelity, Oracle XML DB can ensure that there is no loss of information when the database is used to store and manage XML documents.

Annotating an XML Schema

Oracle XML DB provides the application developer or database administrator with control over how much decomposition, or 'shredding', takes place when an XML document is stored in the database. The XML Schema Recommendation allows vendors to define schema annotations that add directives for specific schema processors. Oracle XML DB schema processor recognizes a set of annotations that make it possible to customize the mapping between the XML schema data types and the SQL data types, control how collections are stored in the database, and specify how much of a document should be shredded.

If you do not specify any annotations to your XML Schema to customize the mapping, Oracle XML DB uses default choices that may or may not be optimal for your application.

Identifying and Processing Instance Documents

Oracle XML DB uses the Document Location Hint to determine which XML schemas are relevant to processing the instance document. It assumes that the Document Location Hint will map directly to the URL used when registering the XML schema with the database.

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 are required 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 the XML Schema 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: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:

- D*efinitions* 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 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, then 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, then it must be greater than or equal to the default value of `minOccurs`, that is, 1 or more. If both attributes are omitted, then 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 value in an instance document; if the attribute does not appear in the instance document, then 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 content in the instance document.

If the element appears without any content, then 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, then 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, then its value must be `US`, and if the attribute does not appear, then 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 or attribute must appear once. It may have any value.
(1, 1) 37, -	required, 37, -	Element or 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 or attribute may appear once. It may have any value.
(0, 1) 37, -	optional, 37, -	Element or attribute may appear once. If it does appear, then its value must be 37. If it does not appear, then its value is 37.

Table B–1 (Cont.) Occurrence Constraints for XML Schema Elements and Attributes

Elements (minOccurs, maxOccurs) fixed, default	Attributes use, fixed,default	Notes
(0, 1) -, 37	optional, -, 37	Element or attribute may appear once. If it does not appear, then 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, then it is not provided. If it does appear and it is empty, then 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 or 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`.

Global Elements and Attributes 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, then 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`, then 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`, then 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, then 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`, then 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-12.

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-ins.

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
`string`	`Confirm this is electric`	--
`normalizedString`	`Confirm this is electric`	3
`token`	`Confirm this is electric`	4
`byte`	`-1, 126`	2
`unsignedByte`	`0, 126`	2
`base64Binary`	`GpM7`	--
`hexBinary`	`0FB7`	--
`integer`	`-126789, -1, 0, 1, 126789`	2
`positiveInteger`	`1, 126789`	2
`negativeInteger`	`-126789, -1`	2
`nonNegativeInteger`	`0, 1, 126789`	2
`nonPositiveInteger`	`-126789, -1, 0`	2

Table B–2 (Cont.) Simple Types Built into XML Schema

Simple Type	Examples (delimited by commas)	Notes
int	-1, 126789675	2
unsignedInt	0, 1267896754	2
long	-1, 12678967543233	2
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
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	--

Table B–2 (Cont.) Simple Types Built into XML Schema

Simple Type	Examples (delimited by commas)	Notes
`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, Canada`	XML 1.0 `NMTOKEN` attribute type. Note: 1
`NMTOKENS`	`US UK, Canada Mexique`	XML 1.0 `NMTOKENS` attribute type, that is, a whitespace separated list of `NMTOKEN` values. 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 calender.

New simple types are defined by deriving them from existing simple types (built-ins 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 range of values of the existing type.

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 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` values, 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:

Creating a List of myInteger

```
<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 will 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 ComplexType from a SimpleType

```
<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>
```

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 seen previously. 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 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 required, 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, style sheets 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 and 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) when 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 `asingleUSAddress`. 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 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 not permitted:

'All' Group Example: Not Permitted

```
<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 Document Type Definition (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 weight and preferred shipping method, you can add `weightKg` and `shipBy` attribute declarations to the item element (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 the XML Schema nil mechanism which enables an element to appear with or without a non-nil value.

The XML Schema 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>
</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 must 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.	

Table B–3 (Cont.) XML Schema Features Compared to DTD Features

XML Schema Feature	DTD Features
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:
`<element name="publish-year">`
`<simpleType>`
`<restriction base="short"`
`<minInclusive value="1970"/`
`<maxInclusive value="2000"/>`
`</restriction>`
`</simpleType>`
`</element>`
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.
	--

Occurrence Indicators (Content Model or Structure) In XML Schema, the structure (called `complexType`) 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. 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.	--
Like the datatype, `complexType` 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. The type declaration and definition mechanism in XML Schema is much more flexible and powerful than in DTDs.	--

Table B–3 (Cont.) XML Schema Features Compared to DTD Features

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: ■ ? = zero or one. In "DTD Example" on page B-27, artist? implied that artist is optional. ■ * = zero or more. ■ + = one or more in the "DTD Example" on page B-27, author+ implies that more than one author is possible. ■ (none) = exactly one.
Identity Constraints 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.	None.
Import or 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 are still required to add more typing and validation declarations to the resulting XML schema definition file before it can be useful 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: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: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 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>
```

C

XPath and Namespace Primer

This appendix describes introductory information about the W3C XPath Recommendation, Namespace Recommendation, and the Information Set (infoset).

This appendix contains these topics:

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 facilities 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-8. A summary of XML Namespaces is provided in "Introducing the W3C Namespaces in XML Recommendation" on page C-12.

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, then 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 <, then 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 (`&quot;` or `&apos;`)
- 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-4 describes the semantics of location paths using the unabbreviated syntax
- "Location Path Examples Using Abbreviated Syntax" on page C-5 describes the abbreviated 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

Table C–1 (Cont.) XPath: Location Path Examples Using Unabbreviated Syntax

Unabbreviated Location Path	Description
`preceding-sibling::chapter[position()=1]`	Selects the previous chapter sibling of the context node
`/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
`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

Table C–2 (Cont.) XPath: Location Path Examples Using Abbreviated Syntax

Abbreviated Location Path	Description
..	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` because 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

AbbreviatedAbsoluteLocationPath `::= '//' 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-12

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 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 Document Type Definition (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 character data. 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 treated the same as `&lt;`. 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 `&lt;` and `&amp;` respectively.

> **Note:** When a text node that contains a < character is written out as XML, an escape character must precede the < character must be escaped for example, by 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 Document Object Model (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 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 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 style sheet 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 re-declares 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, then 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 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 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 namespace of the document 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 productions of this document, 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 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 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-15 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

This section describes how to apply 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 un-prefixed 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"?>
  <!-- unprefixed 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'>
     <th><td>Name</td><td>Origin</td><td>Description</td></th>
     <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>
      </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 not permitted 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 the production of this specification 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 must 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-20.

The information set of an XML document 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 un-expanded 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 because 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 application program interface (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 producing namespace declarations that correspond to the namespaces in scope).

D

XSLT Primer

This appendix describes introductory information about the W3C XSL and XSLT Recommendation.

This appendix contains these topics:

Introducing XSL

XML documents have structure but no format. The 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:

- `http://www.oasis-open.org/cover/xsl.html`
- `http://www.mulberrytech.com/xsl/xsl-list/`
- `http://www.zvon.org/HTMLonly/XSLTutorial/Books/Book1/index.html`
- Chapter 8, "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 style-sheet 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 style sheet. This is because, when XSLT is transforming into the XSL formatting vocabulary, the transformation functions as a style sheet.

This appendix does not specify how an XSLT style sheet is associated with an XML document. It is recommended that XSLT processors support the mechanism described in. When this or any other mechanism yields a sequence of more than one XSLT style sheet to be applied simultaneously to a XML document, then the effect should be the same as applying a single style sheet that imports each member of the sequence in order.

A style sheet 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 style sheet 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 style sheet 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). CSS use a simple mechanism for adding style (fonts, colors, spacing, and so on) to Web documents. 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 style sheets

- An extensible set of formatting objects.

See Also: `http://www.w3.org/Style/XSL/`

Namespaces in XML

A namespace is a unique identifier or name. This is needed because XML documents can be authored separately with different Document Type Definitions (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 Style-Sheet Architecture

The XSLT style sheets must include the following syntax:

- Start tag stating the style sheet, 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 style sheet 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 8, "Transforming and Validating XMLType Data"

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 Style Sheets (CSS)

Cascading Style Sheets (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 Style-Sheet Example, PurchaseOrder.xsl

The following example, `PurchaseOrder.xsl`, is an example of an XSLT style sheet. The example style sheet 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="#FFFFCC" link="#FFCC00"
            vlink="#66CC99" alink="#669999">
        <FONT FACE="Arial, Helvetica, sans-serif">
          <xsl:for-each select="PurchaseOrder"/>
          <xsl:for-each select="PurchaseOrder">
            <center>
              <span style="font-family:Arial; font-weight:bold">
                <FONT COLOR="#FF0000">
                  <B>Purchase Order </B>
                </FONT>
              </span>
            </center>
            <br/>
            <center>
              <xsl:for-each select="Reference">
                <span style="font-family:Arial; font-weight:bold">
                  <xsl:apply-templates/>
                </span>
```

```
    </xsl:for-each>
  </center>
</xsl:for-each>
<P>
  <xsl:for-each select="PurchaseOrder">
    <br/>
  </xsl:for-each>
  <P/>
  <P>
    <xsl:for-each select="PurchaseOrder">
      <br/>
    </xsl:for-each>
  </P>
</P>
<xsl:for-each select="PurchaseOrder"/>
<xsl:for-each select="PurchaseOrder">
  <table border="0" width="100%" BGCOLOR="#000000">
    <tbody>
      <tr>
        <td WIDTH="296">
          <P>
            <B>
              <FONT SIZE="+1" COLOR="#FF0000"
               FACE="Arial, Helvetica, sans-serif">Internal
              </FONT>
            </B>
          </P>
          <table border="0" width="98%" BGCOLOR="#000099">
            <tbody>
              <tr>
                <td WIDTH="49%">
                  <B>
                    <FONT COLOR="#FFFF00">Actions</FONT>
                  </B>
                </td>
                <td WIDTH="51%">
                  <xsl:for-each select="Actions">
                    <xsl:for-each select="Action">
                      <table border="1" WIDTH="143">
                        <xsl:if test="position()=1">
                          <thead>
                            <tr>
                              <td HEIGHT="21">
                                <FONT
                                 COLOR="#FFFF00">User</FONT>
                              </td>
                              <td HEIGHT="21">
                                <FONT
                                 COLOR="#FFFF00">Date</FONT>
                              </td>
                            </tr>
                          </thead>
                        </xsl:if>
                        <tbody>
                          <tr>
                            <td>
                              <xsl:for-each select="User">
                                <xsl:apply-templates/>
                              </xsl:for-each>
                            </td>
```

```
                        <td>
                          <xsl:for-each select="Date">
                            <xsl:apply-templates/>
                          </xsl:for-each>
                        </td>
                      </tr>
                    </tbody>
                  </table>
                </xsl:for-each>
              </xsl:for-each>
            </td>
          </tr>
          <tr>
            <td WIDTH="49%">
              <B>
                <FONT COLOR="#FFFF00">Requestor</FONT>
              </B>
            </td>
            <td WIDTH="51%">
              <xsl:for-each select="Requestor">
                <xsl:apply-templates/>
              </xsl:for-each>
            </td>
          </tr>
          <tr>
            <td WIDTH="49%">
              <B>
                <FONT COLOR="#FFFF00">User</FONT>
              </B>
            </td>
            <td WIDTH="51%">
              <xsl:for-each select="User">
                <xsl:apply-templates/>
              </xsl:for-each>
            </td>
          </tr>
          <tr>
            <td WIDTH="49%">
              <B>
                <FONT COLOR="#FFFF00">Cost Center</FONT>
              </B>
            </td>
            <td WIDTH="51%">
              <xsl:for-each select="CostCenter">
                <xsl:apply-templates/>
              </xsl:for-each>
            </td>
          </tr>
        </tbody>
      </table>
    </td>
    <td width="93"/>
    <td valign="top" WIDTH="340">
      <B>
        <FONT COLOR="#FF0000">
          <FONT SIZE="+1">Ship To</FONT>
        </FONT>
      </B>
      <xsl:for-each select="ShippingInstructions">
        <xsl:if test="position()=1"/>
```

```
        </xsl:for-each>
        <xsl:for-each select="ShippingInstructions">
          <xsl:if test="position()=1">
            <table border="0" BGCOLOR="#999900">
              <tbody>
                <tr>
                  <td WIDTH="126" HEIGHT="24">
                    <B>Name</B>
                  </td>
                  <xsl:for-each
                       select="../ShippingInstructions">
                    <td WIDTH="218" HEIGHT="24">
                      <xsl:for-each select="name">
                        <xsl:apply-templates/>
                      </xsl:for-each>
                    </td>
                  </xsl:for-each>
                </tr>
                <tr>
                  <td WIDTH="126" HEIGHT="34">
                    <B>Address</B>
                  </td>
                  <xsl:for-each
                       select="../ShippingInstructions">
                    <td WIDTH="218" HEIGHT="34">
                      <xsl:for-each select="address">
                        <span style="white-space:pre">
                          <xsl:apply-templates/>
                        </span>
                      </xsl:for-each>
                    </td>
                  </xsl:for-each>
                </tr>
                <tr>
                  <td WIDTH="126" HEIGHT="32">
                    <B>Telephone</B>
                  </td>
                  <xsl:for-each
                       select="../ShippingInstructions">
                    <td WIDTH="218" HEIGHT="32">
                      <xsl:for-each select="telephone">
                        <xsl:apply-templates/>
                      </xsl:for-each>
                    </td>
                  </xsl:for-each>
                </tr>
              </tbody>
            </table>
          </xsl:if>
        </xsl:for-each>
      </td>
    </tr>
  </tbody>
</table>
<br/>
<B>
  <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
</B>
<br/>
<br/>
```

```
<table border="0">
  <xsl:for-each select="LineItems">
    <xsl:for-each select="LineItem">
      <xsl:if test="position()=1">
        <thead>
          <tr bgcolor="#C0C0C0">
            <td>
              <FONT COLOR="#FF0000">
                <B>ItemNumber</B>
              </FONT>
            </td>
            <td>
              <FONT COLOR="#FF0000">
                <B>Description</B>
              </FONT>
            </td>
            <td>
              <FONT COLOR="#FF0000">
                <B>PartId</B>
              </FONT>
            </td>
            <td>
              <FONT COLOR="#FF0000">
                <B>Quantity</B>
              </FONT>
            </td>
            <td>
              <FONT COLOR="#FF0000">
                <B>Unit Price</B>
              </FONT>
            </td>
            <td>
              <FONT COLOR="#FF0000">
                <B>Total Price</B>
              </FONT>
            </td>
          </tr>
        </thead>
      </xsl:if>
      <tbody>
        <tr bgcolor="#DADADA">
          <td>
            <FONT COLOR="#000000">
              <xsl:for-each select="@ItemNumber">
                <xsl:value-of select="."/>
              </xsl:for-each>
            </FONT>
          </td>
          <td>
            <FONT COLOR="#000000">
              <xsl:for-each select="Description">
                <xsl:apply-templates/>
              </xsl:for-each>
            </FONT>
          </td>
          <td>
            <FONT COLOR="#000000">
              <xsl:for-each select="Part">
                <xsl:for-each select="@Id">
                  <xsl:value-of select="."/>
```

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

E

Java APIs: Quick Reference

This appendix provides a quick reference for the Oracle XML DB Java application program interfaces (APIs).

This appendix contains these topics:

Java DOM API For XMLType (oracle.xdb and oracle.xdb.dom Classes)

Packages `oracle.xdb` and `oracle.xdb.dom` implements the Java Document Object Model (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.xdb.dom and oracle.xdb`) classes. Note that class `XMLType` is in package `oracle.xdb` and not `oracle.xdb.dom`.

See Also:

- *Oracle XML API Reference*
- Chapter 12, "Java API for XMLType"

Java Methods Not Supported

The following are methods documented in release 2 (9.2.0.1), but not currently supported:

- `XDBDocument.getElementByID`
- `XDBDocument.importNode`
- `XDBNode.normalize`
- `XDBNode.isSupported`
- `XDBDomImplementation.hasFeature`

Table E–1 Java DOM API for XMLType (mostly oracle.xdb.dom) Classes

Java DOM API for XMLType	Description
`XDBAttribute`	Implements the W3C DOM Node interface for interacting with `XOB`s.
`XDBCData`	Implements `org.w3c.dom.CData`, the W3C text interface.
`XDBCharData`	Implements `org.w3c.dom.CharData`, the W3C CharacterData interface.
`XDBComment`	Implements the `org.w3c.dom.Comment` interface.
`XDBDocument`	Implements the `org.w3c.dom.Document` 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.
`XDBDomImplementation`	Implements `org.w3c.dom.DomImplementation`. Methods: `XDBDomImplementation()`- Opens a JDBC connection to the server.
`XDBElement`	Implements `org.w3c.dom.Element`.
`XDBEntity`	Implements `org.w3c.dom.Entity`.
`XDBNodeMap`	Implements `org.w3c.dom.NamedNodeMap`.
`XDBNode`	Implements `org.w3c.dom.Node`, the W3C DOM Node interface for interacting with XOBs. Methods: `write()` -Writes XML for this Node and all subnodes to an OutputStream. If the OutputStream is ServletOutputStream, the servlet output is committed and data is written using native streaming. public void write(OutputStream s, String charEncoding, short indent); Parameters: `s` - stream to write the output to contains XML text `charEncoding` - IANA char code (for example, "ISO-8859") `indent` - number of characters to indent nested elements
`XDBNodeList`	Implements `org.w3c.dom.NodeList`.
`XDBNotation`	Implements `org.w3c.dom.Notation.`
`XDBProcInst`	Implements `org.w3c.dom.ProcInst`, the W3C DOM ProcessingInstruction interface.
`XDBText`	Implements `org.w3c.dom.Text`.

Table E–1 (Cont.) Java DOM API for XMLType (mostly oracle.xdb.dom) Classes

Java DOM API for XMLType	Description

`XMLType` (package `oracle.xdb`)	Implements Java methods for the SQL type `SYS.XMLTYPE`. Methods:
	`createXML()` - Creates an `XMLType`. Use this method when accessing data through JDBC.
	`getStringVal()` - Retrieves string value containing the XML data from the `XMLType`
	`getClobVal()` - Retrieves the CLOB value containing the XML data from the `XMLType`
	`extract()` - Extracts the given set of nodes from the `XMLType`
	`existsNode()` - Checks for the existence of the given set of nodes in the `XMLType`
	`transform()` - Transforms the `XMLType` using the given XSL document
	`isFragment()` - Checks if the `XMLType` is a regular document or a document fragment
	`getDOM()` - Retrieves the DOM document associated with the `XMLType`.

`createXML()`	Creates an `XMLType`. Throws `java.sql.SQLException` if the `XMLType` could not be created: `public static XMLType createXML( OPAQUE opq);` Creates and returns an `XMLType` given the opaque type containing the `XMLType` bytes. `public static XMLType createXML(Connection conn, String xmlval);` Creates and returns an `XMLType` given the string containing the XML data. `public static XMLType createXML( Connection conn, CLOB xmlval);` Creates and returns an `XMLType` given a CLOB containing the XML data. `public static XMLType createXML(Connection conn, Document domdoc);` Creates and returns an `XMLType` given an instance of the DOM document. Parameters: `opq` - opaque object from which the `XMLType` is to be constructed `conn` - connection object to be used, xmlval - contains the XML data `domdoc` - the DOM Document which represents the DOM tree,
`getStringVal()`	Retrieves the string value containing the XML data from the `XMLType`. Throws `java.sql.SQLException.` `public String getStringVal();`
`getClobVal()`	Retrieves the CLOB value containing the XML data from the `XMLType`. Throws `java.sql.SQLException` `public CLOB getClobVal();`
`getBlobVal()`	Retrieves the BLOB value containing the XML data from the XMLType. Throws java.sql.SQLException. `public oracle.sql.BLOB getBlobVal(int csid)`Parameters: `csid` - the character encoding of the returned BLOB. This value must be a valid Oracle character set id.

Table E–1 (Cont.) Java DOM API for XMLType (mostly oracle.xdb.dom) Classes

Java DOM API for XMLType	Description
`extract()`	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 `java.sql.SQLException` `public XMLType extract( String xpath, String nsmap);` Parameters: `xpath` - xpath expression which specifies for which nodes to search `nsmap` - map of namespaces which resolves the prefixes in the xpath expression; format is "xmlns=a.com xmlns:b=b.com"
`existsNode()`	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 `java.sql.SQLException` public boolean existsNode(String xpath, String nsmap); Parameters: `xpath` - xpath expression that specifies for which nodes to search `nsmap` - map of namespaces that resolves prefixes in the xpath expression; format is "xmlns=a.com xmlns:b=b.com",

`transform()`	Transforms and returns the `XMLType` using the given XSL document. The new (transformed) XML document is returned. Throws `java.sql.SQLException.`
	public `XMLType` transform(`XMLType` xsldoc, String parammap);
	Parameters:
	`xsldoc` - The XSL document to be applied to the `XMLType`
	`parammap` - top level parameters to be passed to the XSL transformation. Use format "a=b c=d e=f". Can be `NULL`.

`isFragment()`	Checks if the `XMLType` is a regular document or document fragment. Returns `TRUE` if doc is a fragment; otherwise, returns `FALSE`. Throws `java.sql.SQLException.` `public boolean isFragment();`
`getDOM()`	Retrieves the DOM document associated with the `XMLType`. This document is the org.w3c.dom.Document. The caller can perform all DOM operations on the Document. If the document is binary, `getDOM` returns `NULL`. Throws `java.sql.SQLException.` `public org.w3c.dom.Document getDOM();`

Oracle XML DB Resource API for Java (oracle.xdb.spi Classes)

Oracle XML DB Resource API for Java's WebDAV support is implemented using package `oracle.xdb.spi` classes that render the service provider interface (SPI) drivers. Classes in `oracle.xdb.spi` implement core WebDAV support for Oracle XML DB. Table E–2 lists the oracle.xdb.spi classes.

See Also: Chapter 25, "Writing Oracle XML DB Applications in Java"

Table E–2 Oracle XML DB Resource API for Java (oracle.xdb.spi)

oracle.xdb.spi Class	Description
`XDBContext` Class	Implements the Java naming and context interface for Oracle XML DB, which extends `javax.naming.context`. In this release there is no federation support, in other words, it is completely unaware of the existence of other namespaces. Methods: `XDBContext()` - Class `XDBContext` constructor. ▪ `public XDBContext( Hashtable env);` Creates an instance of `XDBContext` class given the environment. ▪ `public XDBContext( Hashtable env, String path);` Creates an instance of `XDBContext` class given the environment and path. Parameters: `env` - Environment to describe properties of context, path - Initial path for the context.
`XDBContextFactory` Class	Implements `javax.naming.context`. Methods: `XDBContextFactory()` - Constructor for class `XDBContextFactory.public XDBContextFactory();`
`XDBNameParser` Class	Implements `javax.naming.NameParser`
`XDBNamingEnumeration` Class	Implements `javax.naming.NamingEnumeration`

Table E–2 (Cont.) Oracle XML DB Resource API for Java (oracle.xdb.spi)

oracle.xdb.spi Class	Description
`XDBResource` Class	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. Methods: `XDBResource()` - Creates a new instance of XDBResource `getAuthor()` - Returns author of the resource `getComment()` - Returns the DAV comment of the resource `getContent()` - Returns the content of the resource `getContentType()` - Returns the content type of the resource `getCreateDate()` - Returns the create date of the resource `getDisplayName()` - Returns the display name of the resource `getLanguage()` - Returns the language of the resource `getLastModDate()` - Returns the last modification date of the resource `getOwnerId()` - Returns the owner ID of the resource `setACL()` - Sets the ACL on the resource `setAuthor()` - Sets the author of the resource `setComment()` - Sets the DAV comment of the resource `setContent()` - Sets the content of the resource `setContentType()` - Sets the content type of the resource `setCreateDate()` - Sets the creation date of the resource `setDisplayName()` - Sets the display name of the resource `setLanguage()` - Sets the language of the resource `setLastModDate()` - Sets the last modification date of the resource `setOwnerId()` -Sets the owner ID of the resource
`XDBResource()`	Creates a new instance of XDBResource. public Creates a new instance of XDBResource given the environment. `public XDBResource( Hashtable env, String path);` Creates a new instance of XDBResource given the environment and path. Parameters: `env` - Environment passed in, path - Path to the resource
`getAuthor()`	Retrieves the author of the resource. `public String getAuthor();`
`getComment()`	Retrieves the DAV (Web Distributed Authoring and Versioning) 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();`

Table E–2 (Cont.) Oracle XML DB Resource API for Java (oracle.xdb.spi)

oracle.xdb.spi Class	Description
`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 Authoring and Versioning) 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.
`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 credate); Parameter: `credate` - 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 Database 10*g* Release 1 (10.1): New Java APIs

The following lists new Java APIs added for Oracle Database 10*g* release 1(10.1):

New methods to Manage Node Values Added to XDBNode.java

Get a Node Value

- Writes binary data or character data directly to output stream if csid == 0 then data written as binary, else data written in Oracle specified character set id:
```
public void getNodeValue (java.io.OutputStream, int csid)
throws IOException, DOMException;
```
- Writes character data directly into output stream:
```
public void getNodeValue (java.io.Writer) throws
java.io.IOException, DOMException;
```
- Reads binary data from node value. If csid == 0, then data returned in binary, else data returned in Oracle specific character set id:
```
public java.io.InputStream getNodeValue (int csid) throws
java.io.IOException, DOMException;
```
- Reads node value as character data:
```
public java.io.Reader getNodeValue () throws
java.io.IOException, DOMException;
```

Set a Node Value

- Writes binary data or character data directly to node if csid == 0 then data written as binary, else data written in Oracle specified character set id:
```
public java.io.OutputStream setNodeValue ( int csid) throws
IOException, DOMException;
```
- Writes character data directly to node:
```
public java.io.Writer setNodeValue () throws IOException,
DOMException;
```
- Passes binary or character data. If csid == 0, data is read as binary, else data is read in Oracle specified character set id:
```
public void setNodeValue (java.io.InputStream, int csid)
throws IOException, DOMException;
```
- Passes character data:
```
public void setNodeValue (java.io.Reader) throws IOException,
DOMException;
```

Java DOM APIs to Manage an Attribute Added to XDBAttribute.java

Get an Attribute Value

- Writes binary or character data directly to output stream, if csid == 0, then binary data is written to output stream, else character data in Oracle specified character set id: public void getValue (java.io.OutputStream value, int csid) throws IOException, SQLException;
- Writes character data directly to output stream: public getValue (java.io.Writer value) throws IOException, SQLException;
- Reads binary or character data from attribute value. If csid == 0 then data is returned in binary, else data is returned in Oracle specified character set id: public java.io.InputStream getValue (int csid) throws java.io.IOException, SQLException;

- Reads character data from attribute value: public java.io.Reader getValue () throws java.io.IOException, SQLException;

Set an Attribute Value

- Writes binary or character data directly into attribute if csid == 0 then data is written in binary, else data written in specified Oracle character set id: public java.io.OutputStream setValue (int csid) throws IOException, DOMException;
- Writes character data directly into attribute value: public java.io.Writer setValue () throws IOException, DOMException;
- Passes binary or character data to replace attribute value. if csid == 0 data is read as binary, else data is read in specified Oracle character set id: public void setValue (java.io.InputStream int csid) throws IOException, DOMException;
- Passes character data to replace attribute value: public void setValue (java.io.Reader) throws IOException, DOMException;

New Java XMLType APIs

The following lists two new Java XMLType APIs for Oracle Database 10*g* release 1(10.1):

- New XMLType constructor for BLOB with csid argument:

```
public XMLType (Connection conn, BLOB xmlval, int csid) throws SQLException;
```

- New getBlobVal method with csid argument:

```
public BLOB getBlobVal(int csid) throws SQLException;
```

The API s now also cause exceptions in the following cases:

- `XDBElement.setAttributeNode` - throws `DOMException.INUSE_ATTRIBUTE_ERR` if attribute is in use
- `XDBElement.setAttributeNodeNS` - throws `DOMException.INUSE_ATTRIBUTE_ERR` if attribute is in use
- `XDBNamedNodeMap.setNamedItem` - throws `DOMException.INUSE_ATTRIBUTE_ERR` if attribute is in use
- `XDBNamedNodeMap.setNamedItemNS` - throws `DOMException.INUSE_ATTRIBUTE_ERR` if attribute is in use

F

SQL and PL/SQL APIs: Quick Reference

This appendix provides a summary of the Oracle XML DB SQL and PL/SQL application program interfaces (APIs).

This appendix contains these topics:

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:

- *Oracle XML API Reference*
- Chapter 4, "XMLType Operations"

Table F–1 XMLType API

Function	Description

`XMLType()`
`constructor function XMLType(`
`xmlData IN clob, schema IN varchar2 :=`
` NULL, validated IN number := 0,`
` wellformed IN Number := 0)`
` return self as result`
 	
`constructor function XMLType(`
`xmlData IN varchar2, schema IN varchar2`
` := NULL, validated IN number := 0,`
` wellformed IN number := 0)`
` return self as result`
 	
`constructor function XMLType (`
`xmlData IN "<ADT_1>", schema IN varchar2`
` := NULL, element IN varchar2 := NULL,`
` validated IN number := 0)`
`return self as result`
 	
`constructor function XMLType(`
`xmlData IN SYS_REFCURSOR,`
`schema in varchar2 :=`
` NULL, element in varchar2 := NULL,`
` validated in number := 0)`
`return self as result`
	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.
	
	Parameters:
	
	`xmlData` - data in the form of a `CLOB`, `REF` cursor, `VARCHAR2` or object type.
	
	`schema` - optional schema URL used to make the input conform to the given schema.
	
	`validated` - flag to indicate that the instance is valid according to the given XMLSchema. (default `0`)
	
	`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`)
	
	`element` - optional element name in the case of the `ADT_1` or `REF CURSOR` constructors. (default null)

`constructor function XMLType (`
`xmlData IN BLOB,  csid IN number,`
`schema IN varchar2 := NULL,`
`validated IN number := 0,`
`wellformed IN number := 0)`
`return self as result deterministic;`
 	
`constructor function XMLType (`
`xmlData IN BFILE, csid IN number,`
`schema IN varchar2 := NULL,`
`validated IN number := 0,`
`wellformed IN number := 0)`
`return self as result deterministic;`
	Creates an XMLType instance given a `BLOB` or a `BFILE` with the specified character set id.
	
	Parameters: `xmlData (IN)` - input data to generate the XMLType instance.
	
	`csid (IN)` - Character set id in which the input data is encoded. If zero is specified then the input encoding is auto-detected based on the character detection rule defined in the W3C XML Recommendation.
	
	`schema (IN)` - Optional schema URL to be used to make the input conform to the given schema.
	
	`validated (IN)` - Flag to indicate that the instance is valid according to the given XMLSchema. (default `0`)
	
	`wellformed (IN)` - 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`)

Table F–1 (Cont.) XMLType API

Function	Description
createXML() STATIC FUNCTION createXML(xmlval IN varchar2) RETURN XMLType deterministic STATIC FUNCTION createXML(xmlval IN clob) RETURN XMLType STATIC FUNCTION createXML (xmlData IN clob, schema IN varchar2, validated IN number := 0, wellformed IN number := 0) RETURN XMLType deterministic STATIC FUNCTION createXML (xmlData IN varchar2, schema IN varchar2, validated IN number := 0, wellformed IN number := 0) RETURN XMLType deterministic STATIC FUNCTION createXML (xmlData IN "<ADT_1>", schema IN varchar2 := NULL, element IN varchar2 := NULL, validated IN NUMBER := 0) RETURN XMLType deterministic STATIC FUNCTION createXML (xmlData IN SYS_REFCURSOR, schema in varchar2 := NULL, element in varchar2 := NULL, validated in number := 0) RETURN XMLType deterministic STATIC FUNCTION createXML (xmlData IN blob, csid IN number, schema IN varchar2, validated IN number := 0, wellformed IN number := 0) RETURN sys.XMLType STATIC FUNCTION createXML (xmlData IN bfile, csid IN number, schema IN varchar2, validated IN number := 0, wellformed IN number := 0) RETURN sys.XMLType	Static function for creating and returning an XMLType instance. The string and CLOB parameters used to pass in the date must contain well-formed and valid XML documents. The options are described in the following table. Parameters: xmlData - Actual data in the form of a CLOB, BLOB, BFILE, REF cursor, VARCHAR2 or object type. schema - Optional Schema URL to be used to make the input conform to the given schema. validated - Flag to indicate that the instance is valid according to the given XMLSchema. (default 0) 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) element - Optional element name in the case of the ADT_1 or REF CURSOR constructors. (default NULL) csid - Specifies the character set id of the input xmlData if it is nonzero; otherwise, the character encoding of the input is determined based on the character detection rule defined in the W3C XML Recommendation.
existsNode() MEMBER FUNCTION existsNode(xpath IN varchar2) RETURN number deterministic MEMBER FUNCTION existsNode(xpath in varchar2, nsmap in varchar2) RETURN number deterministic;	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. Parameters: xpath - XPath expression to test. nsmap - Optional namespace mapping.
extract() MEMBER FUNCTION extract(xpath IN varchar2) RETURN XMLType deterministic; MEMBER FUNCTION extract(xpath IN varchar2, nsmap IN varchar2) RETURN XMLType deterministic;	Extracts an XMLType fragment and returns an XMLType instance containing the result node(s). If the XPath does not result in any nodes, then it returns NULL. Parameters: xpath - XPath expression to apply. nsmap - Optional prefix to namespace mapping information.

Table F–1 (Cont.) XMLType API

Function	Description

`isFragment()`
`MEMBER FUNCTION isFragment()`
`RETURN number deterministic;`
	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..

`getClobVal()`
`MEMBER FUNCTION getClobVal()`
`RETURN CLOB deterministic;`
	Returns a `CLOB` containing the serialized XML representation; if the return is a temporary `CLOB`, then it must be freed after use.

`getBlobVal()`
`MEMBER FUNCTION getBlobVal(`
`csid IN number)`
`RETURN BLOB deterministic;`
	Returns a `BLOB` value of an `XMLType` instance in the specified character set.
	Parameter `csid (IN)` - The character set id which the returned `BLOB` will be encoded in. It must be greater than zero and a valid Oracle character set id otherwise an error is returned.

`getNumberVal()`
`MEMBER FUNCTION getNumberVal()`
`RETURN number deterministic;`
	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.

`getStringVal()`
`MEMBER FUNCTION getStringVal()`
`RETURN varchar2 deterministic;`
	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.

`transform()`
`MEMBER FUNCTION transform(`
`xsl IN XMLType, parammap in varchar2 :=`
`NULL) RETURN XMLType deterministic`
	Transforms XML data using the XSL style-sheet argument and the top-level parameters passed as a string of name=value pairs. If any argument other than the parammap is `NULL`, then a `NULL` is returned.
	Parameter
	`xsl` - XSLT style sheet describing the transformation
	`parammap` - top level parameters to the XSL - string of name=value pairs

`toObject()`
`MEMBER PROCEDURE toObject(`
`SELF in XMLType, object OUT "<ADT_1>",`
`schema in varchar2 := NULL,`
`element in varchar2 := NULL)`
	Converts XML data into an instance of a user defined type, using the optional schema and top-level element arguments.
	Parameters:
	`SELF` - instance to be converted. Implicit if used as a member procedure.
	`object` - converted object instance of the required type may be passed in to this function.
	`schema` - schema URL. Mapping of the `XMLType` instance to the converted object instance can be specified using a schema.
	`element` - top-level element name. This specifies the top-level element name in the `XMLSchema` document to map the `XMLType` instance.

`isSchemaBased()`
`MEMBER FUNCTION isSchemaBased`
`return number deterministic`
	Determines if the `XMLType` instance is schema-based. Returns 1 or 0 depending on whether the `XMLType` instance is schema-based or not.

`getSchemaURL()`
`MEMBER FUNCTION getSchemaURL`
`return varchar2 deterministic`
	Returns the XML schema URL corresponding to the `XMLType` instance, if the `XMLType` instance is a schema-based document. Otherwise returns `NULL`.

Table F–1 (Cont.) XMLType API

Function	Description

`getRootElement()`
`MEMBER FUNCTION getRootElement`
`return varchar2 deterministic`
	Gets the root element of the `XMLType` instance. Returns `NULL` if the instance is a fragment.

`createSchemaBasedXML()`
`MEMBER FUNCTION createSchemaBasedXML(`
`schema IN varchar2 := NULL)`
`return sys.XMLType deterministic`
	Creates a schema-based `XMLType` instance from a non-schema-based XML and a schema URL.
	Parameter:
	`schema` - schema URL If `NULL`, then the `XMLType` instance must contain a schema URL.

`createNonSchemaBasedXML()`
`MEMBER FUNCTION createNonSchemaBasedXML`
`return XMLType deterministic`
	Creates a non-schema-based XML document from an XML schema-based instance.

`getNamespace()`
`MEMBER FUNCTION getNamespace`
`return varchar2 deterministic`
	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.

`schemaValidate()` `MEMBER PROCEDURE schemaValidate`	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 then an error is raised; else, the document status is changed to validated.

`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, then conformance is checked against the XML schema of the `XMLType` instance.
	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 must check conformance against a specific elements.

PL/SQL DOM API for XMLType (DBMS_XMLDOM)

Table F–2 lists the PL/SQL Document Object Model (DOM) API for `XMLType` (`DBMS_XMLDOM`) methods supported in release 2 (9.2.0.1). These are grouped according to the W3C DOM Recommendation. The following `DBMS_XMLDOM` methods are *not* supported in release 2 (9.2.0.2):

- `hasFeature`
- `getVersion`
- `setVersion`
- `getCharset`

- setCharset
- getStandalone
- setStandalone
- writeExternalDTDToFile
- writeExternalDTDToBuffer
- writeExternalDTDToClob

Table F–3 lists additional methods supported in release 2 (9.2.0.2).

See Also: Chapter 10, "PL/SQL API for XMLType"

Table F–2 Summary of Release 2 (9.2.0.1) 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.
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.

Table F–2 (Cont.) Summary of Release 2 (9.2.0.1) DBMS_XMLDOM Methods

Group/Method	Description
`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 atribute 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.
CData section methods	--
`isNull()`	Tests if the `CDataSection` is `NULL`.
`makeNode()`	Casts the `CDatasection` to a node.
Character data methods	--
`isNull()`	Tests if the `CharacterData` is `NULL`.
`makeNode()`	Casts the `CharacterData` to a node.

Table F–2 (Cont.) Summary of Release 2 (9.2.0.1) DBMS_XMLDOM Methods

Group/Method	Description
`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. [Not supported in this release]
Document fragment methods	--
`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 node map 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.

Table F–2 (Cont.) Summary of Release 2 (9.2.0.1) DBMS_XMLDOM Methods

Group/Method	Description
`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.
`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. [Not supported in this release]
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.
Text methods	--
`isNull()`	Tests if the text is `NULL`.
`makeNode()`	Casts the text to a node.

Table F–2 (Cont.) Summary of Release 2 (9.2.0.1) DBMS_XMLDOM Methods

Group/Method	Description
`splitText()`	Splits the contents of the text node into two 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. [Not supported in this release]
`setVersion()`	Sets the version of the document. [Not supported in this release]
`getCharset()`	Retrieves the Character set of the document. [Not supported in this release]
`setCharset()`	Sets the Character set of the document. [Not supported in this release]
`getStandalone()`	Retrieves if the document is specified as standalone. [Not supported in this release]
`setStandalone()`	Sets the document standalone. [Not supported in this release]
`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. [Not supported in this release]
`writeExternalDTDToBuffer()`	Writes the DTD of the document to a buffer. [Not supported in this release]
`writeExternalDTDToClob()`	Writes the DTD of the document to a clob. [Not supported in this release]
`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.

Table F–3 DBMS_XMLDOM Methods Added in Release 2 (9.2.0.2)

Method	Syntax
createDocument	FUNCTION createDocument (namspaceURI IN VARCHAR2, qualifiedName IN VARCHAR2, doctype IN DOMType :=NULL) REURN DocDocument;
getPrefix	FUNCTION getPrefix(n DOMNode) RETURN VARCHAR2;
setPrefix	PROCEDURE setPrefix (n DOMNode) RETURN VARCHAR2;
hasAttributes	FUNCTION hasAttributes (n DOMNode) RETURN BOOLEAN;
getNamedItem	FUNCTION getNamedItem (nnm DOMNamedNodeMap, name IN VARCHAR2, ns IN VARCHAR2) RETURN DOMNode;
setNamedItem	FUNCTION getNamedItem (nnm DOMNamedNodeMap, arg IN DOMNode, ns IN VARCHAR2) RETURN DOMNode;
removeNamedItem	FUNCTION removeNamedItem (nnm DOMNamesNodeMap, name in VARCHAR2, ns IN VARCHAR2) RETURN DOMNode;
getOwnerElement	FUNCTION getOwnerElement (a DOMAttr) RETURN DOMElement;
getAttribute	FUNCTION getAttribute (elem DOMElement, name IN VARCHAR2, ns IN VARCHAR2) RETURN VARCHAR2;
hasAttribute	FUNCTION hasAttribute (elem DOMElement, name IN VARCHAR2) RETURN BOOLEAN;
hasAttribute	FUNCTION hasAttribute (elem DOMElement, name IN VARCHAR2, ns IN VARCHAR2) RETURN BOOLEAN;
setAttribute	PROCEDURE setAttribute (elem DOMElement, name IN VARCHAR2, newvalue IN VARCHAR2, ns IN VARCHAR2);
removeAttribute	PROCEDURE removeAttribute (elem DOMElement, name IN VARCHAR2, ns IN VARCHAR2);
getAttributeNode	FUNCTION getAttributeNode(elem DOMElement, name IN VARCHAR2, ns IN VARCHAR2) RETURN DOMAttr;
setAttributeNode	FUNCTION setAttributeNode(elem DOMElement, newAttr IN DOMAttr, ns IN VARCHAR2) RETURN DOMAttr;
createElement	FUNCTION createElement (doc DOMDocument, tagname IN VARCHAR2, ns IN VARCHAR2) RETURN DOMElement;
createAttribute	FUNCTION createAttribute (doc DOMDocument, name IN VARCHAR2, ns IN VARCHAR2) RETURN DOMAttr;

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–4 lists the PL/SQL Parser for XMLType (DBMS_XMLPARSER) functions and procedures.

See Also: Chapter 10, "PL/SQL API for XMLType"

Table F–4 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–5 summarizes the PL/SQL XSLT Processor for `XMLType` (`DBMS_XSLPROCESSOR`) functions and procedures.

See Also: Chapter 10, "PL/SQL API for XMLType"

Table F–5 PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) Functions

Functions and Procedures	Description
`newProcessor()`	Returns a new processor instance.
`processXSL()`	Transforms an input XML document.
`showWarnings()`	Turns warnings on or off.
`setErrorLog()`	Sets errors to be sent to the specified file.
`newStylesheet()`	Creates a new style sheet using the given input and reference URLs.
`transformNode()`	Transforms a node in a DOM tree using the given style sheet.
`selectNodes()`	Selects nodes from a DOM tree that match the given pattern.

Table F–5 (Cont.) PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)

Functions and Procedures	Description
`selectSingleNodes()`	Selects the first node from the tree that matches the given pattern.
`valueOf()`	Retrieves the value of the first node from the tree that matches the given pattern
`setParam()`	Sets a top-level parameter in the style sheet
`removeParam()`	Removes a top-level style-sheet parameter
`resetParams()`	Resets the top-level style-sheet parameters
`freeStylesheet()`	Frees a style-sheet object
`freeProcessor()`	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–6 summarizes the `DBMS_XMLSCHEMA` functions and procedures.

> **See Also:** Chapter 5, "XML Schema Storage and Query: The Basics"

Table F–6 DBMS_XMLSCHEMA Functions and Procedures

Constant	Description

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 := FASLE, 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, genTables IN BOOLEAN := TRUE,force IN BOOLEAN := FALSE, owner IN VARCHAR2 := '',csid IN NUMBER);`
 	
`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);`
	Registers the specified XML schema for use by Oracle XML DB. This schema can then be used to store documents that conform to it.
	
	Parameters:
	
	`schemaURL` - URL that uniquely identifies the schema document. This value is used to derive the path name of the schema document within the XML DB hierarchy.
	
	`schemaDoc` - a valid XML schema document
	
	`local` - is this a local or global schema? By default, all schemas are registered as local schemas, that is under `/sys/schemas/<username>/`... If a schema is registered as global, then it is added under `/sys/schemas/PUBLIC/`.... You need write privileges on the preceding directory to be able to register a schema as global.
	
	`genTypes` - should the schema compiler generate object types? By default, TRUE
	
	`genbean` - should the schema compiler generate Java beans? By default, FALSE.
	
	`genTables` - should the schema compiler generate default tables? By default, TRUE
	
	`force` - if this parameter is set to TRUE, then 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.
	
	`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.
	
	`csid` - specifies the character set id of the input `BLOB` or `BFILE`. If zero is specified then the character encoding of the input is auto-detected as defined by the W3C XML Recommendation.

`procedure registerSchema(schemaURL IN varchar2, schemaDoc IN BLOB,local IN BOOLEAN := TRUE, TRUE, genbean IN BOOLEAN := FALSE, genTables IN BOOLEAN := TRUE,force IN BOOLEAN := FALSE, owner IN VARCHAR2 := '', csid IN NUMBER);`	

Table F–6 (Cont.) DBMS_XMLSCHEMA Functions and Procedures

Constant	Description
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);`	Registers an XML schema specified by a URI name.
deleteSchema() `procedure deleteSchema(schemaURL IN varchar2, delete_option IN pls_integer := DELETE_ RESTRICT);`	Removes the XML schema from Oracle XML DB.
generateBean() `procedure generateBean(schemaURL IN varchar2);`	Generates the Java Bean code corresponding to a registered XML schema.
compileSchema() `procedure compileSchema( schemaURL IN varchar2);`	Recompiles an already registered XML schema. Useful for bringing an invalid schema to a valid state.

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;`
	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–7 lists the Oracle XML DB XML schema catalog views.

Table F–7 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 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–8 summarizes the DBMS_XDB functions and procedures.

See Also: Chapter 21, "PL/SQL Access and Management of Data Using DBMS_XDB"

Table F–8 DBMS_XDB Functions and Procedures

Function/Procedure	Description
`getAclDocument()` `FUNCTION getAclDocument(abspath IN VARCHAR2) RETURN sys.xmltype;`	Retrieves ACL document that protects resource given its path name.
`getPrivileges()` `FUNCTION getPrivileges(res_path IN VARCHAR2) RETURN sys.xmltype;`	Gets all privileges granted to the current user on the given XML DB resource.
`changePrivileges()` `FUNCTION changePrivileges(res_path IN VARCHAR2, ace IN XMLType) RETURN pls_integer;`	Adds the given access control entry (ACE) to the given resource access control list (ACL).
`checkPrivileges()` `FUNCTION checkPrivileges(res_path IN VARCHAR2, privs IN XMLType) RETURN pls_integer;`	Checks access privileges granted to the current user on the specified XML DB resource.
`setacl()` `PROCEDURE setacl(res_path IN VARCHAR2, acl_path IN VARCHAR2);`	Sets the ACL on the given XML DB 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 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–8 (Cont.) DBMS_XDB Functions and Procedures

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(abspath IN VARCHAR2,data IN BFILE,csid IN NUMBER:= 0) RETURN BOOLEAN;`
`FUNCTION CreateResource(abspath IN VARCHAR2, data IN BLOB,csid IN NUMBER:= 0) 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 configuration information to the latest configuration.
`CFG_Get()` `FUNCTION CFG_Get RETURN SYS.XMLType;`	Retrieves the session 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–9 summarizes the `DBMS_XMLGEN` functions and procedures.

See Also: Chapter 15, "Generating XML Data from the Database"

Table F–9 DBMS_XMLGEN Functions and Procedures

Function/Procedure	Description
`newContext()`	Creates a new context handle.
`setRowTag()`	Sets the name of the element enclosing each row of the result. The default tag is `ROW`.
`setRowSetTag ()`	Sets the name of the element enclosing the entire result. The default tag is `ROWSET`.
`getXML()`	Gets the XML document.
`getNumRowsProcessed()`	Gets the number of SQL rows that were processed in the last call to `getXML`.
`setMaxRows()`	Sets the maximum number of rows to be fetched each time.
`setSkipRows()`	Sets the number of rows to skip every time before generating the XML. The default is 0.
`setConvertSpecialChars()`	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.
`convert()`	Converts the XML into the escaped or unescaped XML equivalent.
`useItemTagsForColl()`	Forces the use of the collection column name appended with the `tag _ITEM` for collection elements. The default is to set the underlying object type name for the base element of the collection.
`restartQUERY()`	Restarts the query to start fetching from the beginning.
`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 or WebDAV API can be accessed in SQL through `RESOURCE_VIEW` and `PATH_VIEW`.

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–10 summarizes the Oracle XML DB resource API for PL/SQL operators.

> **See Also:** Chapter 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"

Table F–10 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`
`INTEGER EQUALS_PATH(`
`  resource_column, pathname);`
	Finds the resource with the specified path name.

`PATH` `VARCHAR2 PATH(correlation);`	Returns the relative path name of the resource under the specified path name argument.
`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–11 summarizes the `DBMS_XDB_VERSION` functions and procedures.

> **See Also:** Chapter 19, "Managing Oracle XML DB Resource Versions"

Table F–11 DBMS_XDB_VERSION Functions and Procedures

Function/Procedure	Description
`MakeVersioned()` `FUNCTION MakeVersioned(pathname VARCHAR2) RETURN dbms_xdb.resid_type;`	Turns a regular resource whose path name 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.

Table F–11 (Cont.) DBMS_XDB_VERSION Functions and Procedures

Function/Procedure	Description
`GetPredecessors() FUNCTION GetPredecessors(pathname VARCHAR2) RETURN resid_list_ type;`	Retrieves the list of predecessors by path name.
`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 path name.
`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 configuration 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–12 summarizes the `DBMS_XDBT` functions and procedures.

See Also: *Oracle XML API Reference*

Table F–12 DBMS_XDBT Functions and Procedures

Procedure/Function	Description
`dropPreferences()`	Drops any existing preferences.
`createPreferences()`	Creates preferences required for the `ConText` index on the XML DB hierarchy.

Table F–12 (Cont.) DBMS_XDBT Functions and Procedures

Procedure/Function	Description
`createDatastorePref()`	Creates a USER datastore preference for the ConText index.
`createFilterPref()`	Creates a filter preference for the `ConText` index.
`createLexerPref()`	Creates a lexer preference for the `ConText` index.
`createWordlistPref()`	Creates a stoplist for the `ConText` index.
`createStoplistPref()`	Creates a section group for the `ConText` index.
`createStoragePref()`	Creates a wordlist preference for the `ConText` index.
`createSectiongroupPref()`	Creates a storage preference for the `ConText` index.
`createIndex()`	Creates the `ConText` index on the XML DB hierarchy.
`configureAutoSync()`	Configures the `ConText` index for automatic maintenance (`SYNC`).

New PL/SQL APIs to Support XML Data in Different Character Sets

The following lists the PL/SQL APIs added for this release to load, register schema, and retrieve XML data encoded in different character sets.:

New DBMS_XDB APIs

New CreateResource for BFILE and BLOB with csid argument

```
FUNCTION CreateResource(abspath IN VARCHAR2,
                        data IN BLOB,
                        csid IN NUMBER := 0)
   RETURN BOOLEAN;
FUNCTION CreateResource(abspath IN VARCHAR2,
                        data IN BFILE,
                        csid IN NUMBER := 0)
   RETURN BOOLEAN;
```

New DBMS_XMLSCHEMA APIs

New `registerSchema` for `BLOB` and `BFILE` with `csid` argument.

```
procedure registerSchema(schemaURL IN varchar2,
                         schemaDoc IN BLOB,
                         local IN BOOLEAN := TRUE,
                         genTypes IN BOOLEAN := TRUE,
                         genbean IN BOOLEAN := FALSE,
                         genTables IN BOOLEAN := TRUE,
                         force IN BOOLEAN := FALSE,
                         owner IN VARCHAR2 := '',
                         csid IN NUMBER);

procedure registerSchema(schemaURL IN varchar2,
                         schemaDoc IN BFILE,
                         local IN BOOLEAN := TRUE,
                         genTypes IN BOOLEAN := TRUE,
                         genbean IN BOOLEAN := FALSE,
                         genTables IN BOOLEAN := TRUE,
                         force IN BOOLEAN := FALSE,
                         owner IN VARCHAR2 := '',
```

```
                              csid IN NUMBER);
```

New *URIType APIs

New `getBlob` function with `csid` argument.

```
URIType: MEMBER FUNCTION getBlob(csid IN NUMBER) RETURN blob,
FTPURIType: OVERRIDING MEMBER FUNCTION getBlob(csid IN NUMBER) RETURN blob,
HttpUriType: OVERRIDING MEMBER FUNCTION getBlob(csid IN NUMBER) RETURN blob,
DBURIType: OVERRIDING MEMBER FUNCTION getBlob(csid IN NUMBER) RETURN blob,
XDBURIType: OVERRIDING MEMBER FUNCTION getBlob(csid IN NUMBER) RETURN blob,
```

New DBMS_XSLPROCESSOR

Modified read2Clob and clob2file routines to accept csid argument.

```
FUNCTION read2clob(flocation VARCHAR2, fname VARCHAR2, csid IN NUMBER := 0)
  RETURN CLOB;
PROCEDURE clob2file(cl CLOB, flocation VARCHAR2,
                    fname VARCHAR2, csid IN NUMBER := 0);
```

New PL/SQL XMLType APIs

New XMLType constructors for BLOB and BFILE with csid:

```
CONSTRUCTOR FUNCTION XMLType(xmlData IN blob, csid IN number,
                             schema IN varchar2 := NULL,
                             validated IN number := 0,
                             wellformed IN number := 0)
              RETURN self AS result,
CONSTRUCTOR FUNCTION XMLType(xmlData IN bfile, csid IN number,
                            schema IN varchar2 := NULL,
                            validated IN number := 0,
                            wellformed IN number := 0)
              RETURN self AS result,
```

New createXML methods for BLOB and BFILE with csid.

```
STATIC FUNCTION createXML(xmlData IN blob, csid IN number, schema IN varchar2,
                          validated IN number := 0, wellformed IN number := 0)
          RETURN sys.XMLType
STATIC FUNCTION createXML(xmlData IN BFILE, csid IN number, schema IN varchar2,
                          validated IN number := 0, wellformed IN number := 0)
          RETURN sys.XMLType.
MEMBER FUNCTION getBlobVal(csid IN number) RETURN BLOB
```

G

C API for XML (Oracle XML DB): Quick Reference

This appendix provides a quick reference for the C API for XML, Oracle XML DB specific functions.

This appendix contains this topic:

- XML Context

XML Context

An XML Context is a required parameter to all the C DOM API functions. This context encapsulates information pertaining to data encoding, error message language and such. This contents of this opaque context are different for XDK applications and Oracle XML DB.

For Oracle XML DB, there are two OCI functions provided to initialize an XML Context. The function `OCIXmlDbInitXmlCtx()` is used to initialize the context while `OCIXmlDbFreeXmlCtx()` tears it down.

OCIXmlDbFreeXmlCtx() Syntax

Here is the `OCIXmlDbFreeXmlCtx()` syntax:

```
void OCIXmlDbFreeXMlCtx ( xmlctx   *xctx );
```

where parameter, `xctx (IN)` is the XML context to terminate.

OCIXmlDbInitXmlCtx() Syntax

Here is the `OCIXmlDbInitXmlCtx()` syntax:

```
xmlctx *OCIXmlDb_CtxInit (   OCIEnv              *envhp,
                             OCISvcHp            *svchp,
                             OCIError            *errhp,
                             ocixmldbparam       *params,
                             ub4                 num_params );
```

H

Oracle XML DB-Supplied XML Schemas and Additional Examples

This appendix includes the definition and structure of RESOURCE_VIEW and PATH_VIEW and the Oracle XML DB-supplied XML schemas. It also includes a full listing of the C example for loading XML content into Oracle XML DB.

This appendix contains these topics:

- 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 DB ACLs
- xdbconfig.xsd: XML Schema for Configuring Oracle XML DB
- Loading XML Using C (OCI)

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
RESID       RAW          Resource OID which is a unique handle to the resource
```

See Also: Chapter 20, "SQL Access Using 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
RESID       RAW          Resource OID
```

See Also: Chapter 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"

XDBResource.xsd: XML Schema for Representing Oracle XML DB Resources

Here is the listing for the Oracle XML DB supplied 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"/>
    <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 pname="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>

  <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"/>
  <attribute name="StickyRef" type="boolean"/>
```

```
</complexType>
<element name="Resource" type="res:ResourceType"/>
</schema>
```

acl.xsd: XML Schema for Representing Oracle XML DB ACLs

This section describes the Oracle XML DB supplied XML schema used to represent Oracle XML DB access control lists (ACLs):

ACL Representation XML Schema, acl.xsd

XML schema, `acl.xsd`, represents Oracle XML DB access control lists (ACLs):

acl.xsd

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://xmlns.oracle.com/xdb/acl.xsd" version="1.0"
        xmlns:xdb="http://xmlns.oracle.com/xdb"
        xmlns:xdbacl="http://xmlns.oracle.com/xdb/acl.xsd"
        elementFormDefault="qualified">

 <annotation>
     <documentation>
        This XML schema describes the structure of XML DB ACL documents.

        Note : The following "systemPrivileges" element lists all supported
          system privileges and their aggregations.
          See dav.xsd for description of DAV privileges
        Note : The elements and attributes marked "hidden" are for
          internal use only.
     </documentation>
     <appinfo>
       <xdb:systemPrivileges>
        <xdbacl:all>
          <xdbacl:read-properties/>
          <xdbacl:read-contents/>
          <xdbacl:read-acl/>
          <xdbacl:update/>
          <xdbacl:link/>
          <xdbacl:unlink/>
          <xdbacl:unlink-from/>
          <xdbacl:write-acl-ref/>
          <xdbacl:update-acl/>
          <xdbacl:link-to/>
          <xdbacl:resolve/>
        </xdbacl: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 XML DB ACL namespace -->
  <element name = "read-properties" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "read-contents" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "read-acl" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "update" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "link" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "unlink" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "unlink-from" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "write-acl-ref" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "update-acl" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "link-to" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "resolve" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  <element name = "all" type="xdbacl:privilegeNameType"
          substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>

<!-- privilege element -->
  <element name = "privilege" xdb:SQLType = "XDB$PRIV_T" xdb:defaultTable="">
    <complexType>
      <choice maxOccurs="unbounded">
        <any xdb:transient="generated"/>
        <!-- HIDDEN ELEMENTS -->
        <element name = "privNum" type = "hexBinary" xdb:baseProp="true"
                 xdb:hidden="true"/>
      </choice>
    </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"
                xdb:transient="generated"/>
       <element ref="xdbacl:privilege" minOccurs="1"/>
       <!-- HIDDEN ELEMENTS -->
       <element name = "principalID" type = "hexBinary" minOccurs="0"
                xdb:baseProp="true" xdb:hidden="true"/>
       <element name = "flags" type = "unsignedInt" minOccurs="0"
                xdb:baseProp="true" xdb:hidden="true"/>
     </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"
```

```
                xdb:transient="generated"/>
        <element name = "elementName" type = "string" minOccurs="0"
                xdb:transient="generated"/>
        <element ref = "xdbacl:ace" minOccurs="1" maxOccurs = "unbounded"
                xdb:SQLCollType="XDB$ACE_LIST_T"/>

        <!-- HIDDEN ELEMENTS -->
        <element name = "schemaOID" type = "hexBinary" minOccurs="0"
                xdb:baseProp="true" xdb:hidden="true"/>
        <element name = "elementNum" type = "unsignedInt" minOccurs="0"
                xdb:baseProp="true" xdb:hidden="true"/>
       </sequence>
       <attribute name = "shared" type = "boolean" default="true"/>
       <attribute name = "description" type = "string"/>
      </complexType>
    </element>

   </schema>';
```

xdbconfig.xsd: XML Schema for Configuring Oracle XML DB

`xdbconfig.xsd`, is the Oracle XML DB supplied XML schema used to configure Oracle XML DB:

xdbconfig.xsd

```
<schema targetNamespace="http://xmlns.oracle.com/xdb/xdbconfig.xsd"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xdbc="http://xmlns.oracle.com/xdb/xdbconfig.xsd"
        xmlns:xdb="http://xmlns.oracle.com/xdb"
        version="1.0" elementFormDefault="qualified">

  <element name="xdbconfig" xdb:defaultTable="XDB$CONFIG">

  <complexType><sequence>

    <!-- predefined XML DB properties - these should NOT be changed -->
    <element name="sysconfig">
    <complexType><sequence>
        <!-- generic XML DB properties -->
        <element name="acl-max-age" type="unsignedInt" default="1000"/>
        <element name="acl-cache-size" type="unsignedInt" default="32"/>
        <element name="invalid-pathname-chars" type="string" default=""/>
        <element name="case-sensitive" type="boolean" default="true"/>
        <element name="call-timeout"   type="unsignedInt" default="300"/>
        <element name="max-link-queue" type="unsignedInt" default="65536"/>
        <element name="max-session-use" type="unsignedInt" default="100"/>
        <element name="persistent-sessions" type="boolean" default="false"/>
        <element name="default-lock-timeout" type="unsignedInt"
                                               default="3600"/>
        <element name="xdbcore-logfile-path" type="string"
                                  default="/sys/log/xdblog.xml"/>
        <element name="xdbcore-log-level" type="unsignedInt" default="0"/>
        <element name="resource-view-cache-size" type="unsignedInt"
                                              default="1048576"/>
        <element name="case-sensitive-index-clause" type="string" minOccurs="0"/>

        <!-- protocol specific properties -->
        <element name="protocolconfig">
```

```
<complexType><sequence>

    <!-- these apply to all protocols -->
    <element name="common">
    <complexType><sequence>
        <element name="extension-mappings">
        <complexType><sequence>
          <element name="mime-mappings" type="xdbc:mime-mapping-type"/>
          <element name="lang-mappings" type="xdbc:lang-mapping-type"/>
          <element name="charset-mappings"
                                    type="xdbc:charset-mapping-type"/>
          <element name="encoding-mappings"
                                   type="xdbc:encoding-mapping-type"/>
        </sequence></complexType>
        </element>
        <element name="session-pool-size" type="unsignedInt"
                                                default="50"/>
        <element name="session-timeout"   type="unsignedInt"
                                                default="6000"/>

    </sequence></complexType>
    </element>

    <!-- FTP specific -->
    <element name="ftpconfig">
    <complexType><sequence>
      <element name="ftp-port" type="unsignedShort" default="2100"/>
      <element name="ftp-listener" type="string"/>
      <element name="ftp-protocol" type="string"/>
      <element name="logfile-path" type="string"
                                      default="/sys/log/ftplog.xml"/>
      <element name="log-level" type="unsignedInt" default="0"/>
      <element name="session-timeout"  type="unsignedInt"
                                             default="6000"/>
      <element name="buffer-size" default="8192">
        <simpleType>
          <restriction base="unsignedInt">
            <minInclusive value="1024"/>       <!-- 1KB -->
            <maxInclusive value="1048496"/>    <!-- 1MB -->
          </restriction>
        </simpleType>
      </element>
    </sequence></complexType>
    </element>

    <!-- HTTP specific -->
    <element name="httpconfig">
    <complexType><sequence>
        <element name="http-port" type="unsignedShort" default="8080"/>
        <element name="http-listener" type="string"/>
        <element name="http-protocol" type="string"/>
        <element name="max-http-headers" type="unsignedInt"
                                                 default="64"/>
        <element name="max-header-size" type="unsignedInt"
                                                 default="4096"/>
        <element name="max-request-body" type="unsignedInt"
                                  default="2000000000" minOccurs="1"/>
        <element name="session-timeout"  type="unsignedInt"
                                               default="6000"/>
        <element name="server-name" type="string"/>
```

```
                <element name="logfile-path" type="string"
                                          default="/sys/log/httplog.xml"/>
                <element name="log-level" type="unsignedInt" default="0"/>
                <element name="servlet-realm" type="string" minOccurs="0"/>

                <element name="webappconfig">
                <complexType><sequence>
                  <element name="welcome-file-list"
                                            type="xdbc:welcome-file-type"/>
                  <element name="error-pages" type="xdbc:error-page-type"/>
                  <element name="servletconfig"
                                           type="xdbc:servlet-config-type"/>
                </sequence></complexType>
                </element>
                <element name="default-url-charset" type="string"
                                                              minOccurs="0"/>
            </sequence></complexType>
            </element>

        </sequence></complexType>
        </element>

    </sequence></complexType>
    </element>


  <!-- users can add any properties they want here -->
  <element name="userconfig" minOccurs="0">
       <complexType><sequence>
          <any maxOccurs="unbounded" namespace="##other"/>
       </sequence></complexType>
  </element>

</sequence></complexType>

</element>

<complexType name="welcome-file-type">
   <sequence>
       <element name="welcome-file" minOccurs="0" maxOccurs="unbounded">
        <simpleType>
        <restriction base="string">
          <pattern value="[^/]*"/>
        </restriction>
        </simpleType>
       </element>
   </sequence>
</complexType>

<!-- customized error pages -->
<complexType name="error-page-type">
<sequence>
      <element name="error-page" minOccurs="0" maxOccurs="unbounded">
      <complexType><sequence>
            <choice>
              <element name="error-code">
                <simpleType>
                  <restriction base="positiveInteger">
                    <minInclusive value="100"/>
                    <maxInclusive value="999"/>
```

```
                    </restriction>
                  </simpleType>
                </element>

                <!-- Fully qualified classname of a Java exception type -->
                <element name="exception-type" type="string"/>

                <element name="OracleError">
                 <complexType><sequence>
                    <element name="facility" type="string" default="ORA"/>
                    <element name="errnum" type="unsignedInt"/>
                 </sequence></complexType>
                </element>
              </choice>

              <element name="location" type="anyURI"/>

       </sequence></complexType>
       </element>
</sequence>
</complexType>


<!-- parameter for a servlet: name, value pair and a description  -->
<complexType name="param">
  <sequence>
    <element name="param-name" type="string"/>
    <element name="param-value" type="string"/>
    <element name="description" type="string"/>
  </sequence>
</complexType>

<complexType name="servlet-config-type">
  <sequence>
       <element name="servlet-mappings">
         <complexType><sequence>
            <element name="servlet-mapping" minOccurs="0"
                     maxOccurs="unbounded">
            <complexType><sequence>
                <element name="servlet-pattern" type="string"/>
                <element name="servlet-name" type="string"/>
            </sequence></complexType>
            </element>
         </sequence></complexType>
        </element>

    <element name="servlet-list">
      <complexType><sequence>
            <element name="servlet" minOccurs="0" maxOccurs="unbounded">
              <complexType><sequence>
                <element name="servlet-name" type="string"/>
                <element name="servlet-language">
                  <simpleType>
                    <restriction base="string">
                      <enumeration value="C"/>
                      <enumeration value="Java"/>
                      <enumeration value="PL/SQL"/>
                    </restriction>
                  </simpleType>
                </element>
```

```
                    <element name="icon" type="string" minOccurs="0"/>
                    <element name="display-name" type="string"/>
                    <element name="description" type="string" minOccurs="0"/>
                    <choice>
                      <element name="servlet-class" type="string" minOccurs="0"/>
                      <element name="jsp-file" type="string" minOccurs="0"/>
                    </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>
```

Loading XML Using C (OCI)

The following example is also partially listed in Chapter 3, "Using Oracle XML DB", "Loading XML Content into Oracle XML DB Using C" on page 3-7:

Example H–1 Loading XML Content into Oracle XML DB Using C

```
#include <xml.h>
#include <string.h>
#include <ocixmldb.h>

OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIServer *srvhp;
OCIDuration dur;
OCISession *sesshp;

oratext *username;
oratext *password;
oratext *filename;
oratext *schemaloc;

/*---------------------------------------------------------*/
/* execute a sql statement which binds xml data */
/*---------------------------------------------------------*/

sword exec_bind_xml(OCISvcCtx *svchp, OCIError *errhp, OCIStmt *stmthp,
                    void *xml, OCIType *xmltdo, OraText *sqlstmt)
{

  OCIBind *bndhp1 = (OCIBind *) 0;
  sword  status = 0;
  OCIInd ind = OCI_IND_NOTNULL;
  OCIInd *indp = &ind;

  if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                             (ub4)strlen((const char *)sqlstmt),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    return OCI_ERROR;

  if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *) 0,
                           (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                           (ub2 *)0, (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    return OCI_ERROR;

  if(status = OCIBindObject(bndhp1, errhp, (CONST OCIType *) xmltdo,
```

```
                            (dvoid **) &xml, (ub4 *) 0,
                            (dvoid **) &indp, (ub4 *) 0))
    return OCI_ERROR;

  if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                             (ub4) OCI_DEFAULT))
    return OCI_ERROR;

  return OCI_SUCCESS;
}


/*---------------------------------------------------------*/
/* initialize oci handles and connect                      */
/*---------------------------------------------------------*/

sword init_oci_connect()
{

  sword status;
  if (OCIEnvCreate((OCIEnv **) &(envhp), (ub4) OCI_OBJECT,
                   (dvoid *) 0, (dvoid * (*)(dvoid *,size_t)) 0,
                   (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
                   (void (*)(dvoid *, dvoid *)) 0, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIEnvCreate()\n");
    return OCI_ERROR;
  }

  /* allocate error handle */
  if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &(errhp),
                     (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on errhp\n");
    return OCI_ERROR;
  }

  /* allocate server handle */
  if (status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp,
                           (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on srvhp\n");
    return OCI_ERROR;
  }

  /* allocate service context handle */
  if (status = OCIHandleAlloc((dvoid *) envhp,
                              (dvoid **) &(svchp), (ub4) OCI_HTYPE_SVCCTX,
                              (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on svchp\n");
    return OCI_ERROR;
  }

  /* allocate session handle */
  if (status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &sesshp ,
                           (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on sesshp\n");
```

```
  return OCI_ERROR;
}

/* allocate statement handle */
if (OCIHandleAlloc((dvoid *)envhp, (dvoid **) &stmthp,
                   (ub4)OCI_HTYPE_STMT, (CONST size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on stmthp\n");
  return status;
}

if (status = OCIServerAttach((OCIServer *) srvhp, (OCIError *) errhp,
                             (CONST oratext *)"", 0, (ub4) OCI_DEFAULT))
{
  printf("FAILED: OCIServerAttach() on srvhp\n");
  return OCI_ERROR;
}

/* set server attribute to service context */
if (status = OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                        (dvoid *) srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER,
                        (OCIError *) errhp))
{
  printf("FAILED: OCIAttrSet() on svchp\n");
  return OCI_ERROR;
}

/* set user attribute to session */
if (status = OCIAttrSet((dvoid *)sesshp, (ub4) OCI_HTYPE_SESSION,
                        (dvoid *)username,
                        (ub4) strlen((const char *)username),
                        (ub4) OCI_ATTR_USERNAME, (OCIError *) errhp))
{
  printf("FAILED: OCIAttrSet() on authp for user\n");
  return OCI_ERROR;
}

/* set password attribute to session */
if (status = OCIAttrSet((dvoid *) sesshp, (ub4) OCI_HTYPE_SESSION,
                        (dvoid *)password,
                        (ub4) strlen((const char *)password),
                        (ub4) OCI_ATTR_PASSWORD, (OCIError *) errhp))
{
  printf("FAILED: OCIAttrSet() on authp for password\n");
  return OCI_ERROR;
}

/* Begin a session  */
if (status = OCISessionBegin((OCISvcCtx *) svchp,
                             (OCIError *) errhp,
                             (OCISession *) sesshp, (ub4) OCI_CRED_RDBMS,
                             (ub4) OCI_STMT_CACHE))
{
  printf("FAILED: OCISessionBegin(). Make sure database is up and
          the username/password is valid. \n");
  return OCI_ERROR;
}

/* set session attribute to service context */
if (status = OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
```

```
                          (dvoid *)sesshp, (ub4) 0, (ub4) OCI_ATTR_SESSION,
                          (OCIError *) errhp))
  {
    printf("FAILED: OCIAttrSet() on svchp\n");
    return OCI_ERROR;
  }
}


/*---------------------------------------------------------*/
/* free oci handles and disconnect                         */
/*---------------------------------------------------------*/

void free_oci()
{

  sword  status = 0;

  /* End the session */
  if (status = OCISessionEnd((OCISvcCtx *)svchp, (OCIError *)errhp,
                             (OCISession *)sesshp, (ub4) OCI_DEFAULT))
  {
    if (envhp)
      OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
    return;
  }

  /* Detach from the server */
  if (status = OCIServerDetach((OCIServer *)srvhp, (OCIError *)errhp,
                               (ub4)OCI_DEFAULT))
  {
    if (envhp)
      OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
    return;
  }

  /* Free the handles */
  if (stmthp)
    OCIHandleFree((dvoid *)stmthp, (ub4) OCI_HTYPE_STMT);

  if (sesshp)
    OCIHandleFree((dvoid *)sesshp, (ub4) OCI_HTYPE_SESSION);

  if (svchp)
    OCIHandleFree((dvoid *)svchp, (ub4) OCI_HTYPE_SVCCTX);

  if (srvhp)
    OCIHandleFree((dvoid *)srvhp, (ub4) OCI_HTYPE_SERVER);

  if (errhp)
    OCIHandleFree((dvoid *)errhp, (ub4) OCI_HTYPE_ERROR);

  if (envhp)
    OCIHandleFree((dvoid *)envhp, (ub4) OCI_HTYPE_ENV);

  return;
}


void main()
```

```
{

    OCIType *xmltdo;

    xmldocnode  *doc;
    ocixmldbparam params[1];
    xmlerr       err;
    xmlctx  *xctx;

    oratext *ins_stmt;


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

    /* Get an xml context */
    params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
    params[0].value_ocixmldbparam = &dur;
    xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, 1);

    if (!(doc = XmlLoadDom(xctx, &err, "file", filename,
                           "schema_location", schemaloc, NULL)))
    {
       printf("Parse failed.\n");
       return;
    }
    else
       printf("Parse succeeded.\n");

    printf("The xml document is :\n");
    XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

    /* Insert the document to my_table */
    ins_stmt = (oratext *)"insert into PURCHASEORDER values (:1)";

    status = OCITypeByName(envhp, errhp, svchp, (const text *) "SYS",
                  (ub4) strlen((const char *)"SYS"), (const text *) "XMLTYPE",
                  (ub4) strlen((const char *)"XMLTYPE"), (CONST text *) 0,
                  (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                  (OCIType **) &xmltdo);

    if (status == OCI_SUCCESS)
    {
      status = exec_bind_xml(svchp, errhp, stmthp, (void *)doc,
                             xmltdo, ins_stmt);
    }

    if (status == OCI_SUCCESS)
      printf ("Insert successful\n");
    else
      printf ("Insert failed\n");

    /* free xml instances */
    if (doc)
      XmlFreeDocument((xmlctx *)xctx, (xmldocnode *)doc);
    /* free xml ctx */
    OCIXmlDbFreeXmlCtx(xctx);
    free_oci();
}
```

Oracle XML DB Feature Summary

This appendix describes the Oracle XML DB feature summary.

This appendix contains these topics:

Oracle XML DB Feature Summary

This Oracle XML DB feature list includes XML features available in the database since Oracle9*i* release 1 (9.0.1).

XMLType Features

The Oracle XML DB native datatype `XMLType` helps store and manipulate XML. Multiple storage options (Character Large Object (CLOB) or structured XML) are available with `XMLType`, 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.

With `XMLType`, you can perform SQL operations such as:

- 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

You can build regular SQL indexes or Oracle Text indexes on `XMLType` for high performance for a very broad spectrum of applications. See Chapter 4, "XMLType Operations".

DOM Fidelity

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. Document Object Model (DOM) fidelity does not, however, imply maintenance of whitespace; if you want to preserve the exact layout of XML, including whitespace, you can use CLOB storage. See Chapter 3, "Using Oracle XML DB" and Chapter 5, "XML Schema Storage and Query: The Basics".

Document Fidelity

For applications that need to store XML while maintaining complete fidelity to the original, including whitespace characters, the CLOB storage option is available.

XML Schema

You can constrain XML documents to W3C XML Schemas. This provides a standard data model for all your data (structured and unstructured). You can use the database to enforce this data model. See Chapter 5, "XML Schema Storage and Query: The Basics" and Appendix B, "XML Schema Primer".

- 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, "XML Schema Storage and Query: The Basics". Note: If you choose CLOB storage option, available with `XMLType` since Oracle9*i* release 1 (9.0.1), you can preserve whitespace.
- 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 8, "Transforming and Validating XMLType Data".

XML Piecewise Updates

In Oracle XML DB 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, "XML Schema Storage and Query: The Basics".

XPath Search

Use XPath syntax (embedded in a SQL statement or as part of an HTTP request) to query XML content in the database. See Chapter 4, "XMLType Operations" and Chapter 9, "Full Text Search Over XML".

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, "XMLType Operations".

SQL/XML Operators

SQL/XML operators comply with the emerging ANSI standard. For example, `XMLElement()` to create XML elements on the fly, to make XML queries and on-the-fly XML generation easy. These render SQL and XML metaphors interoperable.See Chapter 15, "Generating XML Data from the Database".

XSL Transformations for XMLType

Use XSLT to transform XML documents through a SQL operator. Database-resident, high-performance XSL transformations. See Chapter 8, "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. Use this for 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 10, "PL/SQL API for XMLType".

XML Views

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 16, "XMLType Views".

PL/SQL, Java, and OCI Interfaces

Use DOM-based and other Application Program Interfaces for accessing and manipulating XML data. You can get static and dynamic access to XML. See Chapter 10, "PL/SQL API for XMLType", Chapter 12, "Java API for XMLType", and Chapter 13, "Using C API for XML With Oracle XML DB".

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, "XML Schema Storage and Query: The Basics".

Generating XML

Use SQL operators such as `SYS_XMLGEN` and `SYS_XMLAGG` provide native for high-performance generation of XML from SQL queries. SQL/XML operators such as `XMLElement()` create XML tables and elements on the fly and make XML generation more flexible. See Chapter 15, "Generating XML Data from the Database".

Oracle XML DB Repository Features

Oracle XML DB repository is an XML repository built in the database for foldering. The repository structure enables you to view XML content stored in Oracle XML DB as a hierarchy of directory-like folders. See Chapter 18, "Accessing Oracle XML DB Repository Data".

- The repository supports *access control lists* (ACLs) for any `XMLType` object, and lets you define your own privileges in addition to providing certain system-defined ones. See Chapter 23, "Oracle XML DB Resource Security".
- 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 *resources*.
- 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 `CONNECT BY`).

Searching the Repository Using SQL

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 20, "SQL Access Using RESOURCE_VIEW and PATH_VIEW".

Accessing Repository Data Using WebDAV, HTTP, and FTP

You can access any foldered `XMLType` row using WebDAV and FTP. Users manipulating XML data in the database can use HTTP. See Chapter 24, "FTP, HTTP, and WebDAV Access to Repository Data".

Versioning

Oracle XML DB provides versioning and version-management capabilities over resources managed by the XML repository. See Chapter 19, "Managing Oracle XML DB Resource Versions".

Standards Supported

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

- W3C XML Schema 1.0 Recommendation
- W3C XPath 1.0 Recommendation
- W3C XSL 1.0 Recommendation
- W3C DOM Recommendation Levels 1.0 and 2.0 Core
- Protocol support: HTTP, FTP, IETF WebDAV, as well as Oracle Net
- 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)
- Web Services and Simple Object Access Protocol (SOAP). You can access XML stored in the server from SOAP requests
- ISO-ANSI Working Draft for XML-Related Specifications (SQL/XML) [ISO/IEC 9075 Part 14 and ANSI]. Emerging ANSI SQL/XML functions to query XML from SQL. The task force defining these specifications falls under the auspices of the International Committee for Information Technology Standards (INCITS). The SQL/XML specification will be fully aligned with SQL:2003
- Java Database Connectivity (JDBC)

Oracle XML DB Limitations

The following lists Oracle XML DB limitations:

Replication

Oracle XML DB does not support replication of `XMLType` tables.

Extending Resource Metadata Properties

In the current release, you cannot extend the resource schema. However, you can set and access custom properties belonging to other namespaces, other than `XDBResource.xsd`, using DOM operations on the <Resource> document.

References Within Scalars

Oracle does not currently support references within a scalar, `XMLType,` or LOB data column.

Thin JDBC Driver is Not Supported by All XMLType Functions

`extract()`, `transform()`, and `existsNode()` methods only work with the Thick JDBC driver. Not all `oracle.xdb.XMLType` functions are supported by the Thin JDBC driver. However, if you do not use `oracle.xdb.XMLType` classes and OCI driver, you could lose performance benefits associated with the intelligent handling of XML.

NVARCHAR and NCHAR SQLTypes are Not Supported

Oracle XML DB does not support NVARCHAR or NCHAR as a SQLType when registering an XML schema. In other words in the XML schema .xsd file you cannot specify that an element should be of type NVARCHAR or NCHAR. Also, if you provide your own type you should not use these datatypes.

SubstitutionGroup Limited to 2048 Elements

If a schema document uses more than 2048 substitutable elements for a given head element, error ORA-31160 occurs. Rewrite the schema to use less than 2048 substitutable elements for each head element.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X