Oracle® XML DB

Developer's Guide

10*g* Release 2 (10.2)

B14259-02

August 2005

This manual describes Oracle XML DB. It includes guidelines and examples for loading, storing, generating, manipulating, managing, accessing, and querying XML data in Oracle Database.

ORACLE®

Oracle XML DB Developer's Guide, 10*g* Release 2 (10.2)

B14259-02

Primary Author: Drew Adams

Contributing Author: Nipun Agarwal, Abhay Agrawal, Omar Alonso, David Anniss, Sandeepan Banerjee, Mark Bauer, Ravinder Booreddy, Stephen Buxton, Yuen Chan, Sivasankaran Chandrasekar, Vincent Chao, Ravindranath Chennoju, Dan Chiba, Mark Drake, Fei Ge, Wenyun He, Shelley Higgins, 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

Contents

2 Getting Started with Oracle XML DB

3 Using Oracle XML DB

5 XML Schema Storage and Query: Basic

6 XPath Rewrite

7 XML Schema Storage and Query: Advanced

8 XML Schema Evolution

9 Transforming and Validating XMLType Data

10 Full-Text Search Over XML

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

11 PL/SQL API for XMLType

12 Package DBMS_XMLSTORE

13 Java API for XMLType

14 Using the C API for XML

17 Using XQuery with Oracle XML DB

18 XMLType Views

19 Accessing Data Through URIs

21 Managing Resource Versions

22 SQL Access Using RESOURCE_VIEW and PATH_VIEW

23 PL/SQL Access Using DBMS_XDB

24 Repository Resource Security

25 FTP, HTTP(S), and WebDAV Access to Repository Data

26 User-Defined Repository Metadata

27 Writing Oracle XML DB Applications in Java

Part VI Oracle Tools that Support Oracle XML DB

28 Administering Oracle XML DB

29 Loading XML Data Using SQL*Loader

30 Importing and Exporting XMLType Tables

31 Exchanging XML Data with Oracle Streams AQ

Part VII Appendixes

A XML Schema Primer

B XPath and Namespace Primer

List of Examples

Preface

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

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

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

This Preface contains these topics:

Audience

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

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. A working knowledge of one 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. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

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

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

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, "Introduction to 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: Basic"

Describes how to use Oracle XML DB mapping from SQL to XML and back, provides an overview of how to register, delete, and update XML schemas, and how to use the default mapping of Oracle XML DB or specify your own.

Chapter 6, "XPath Rewrite"

Describes the rewriting of XPath expressions to achieve optimal performance. XPath rewrite is detailed for SQL functions `existsNode`, `extract`, `extractValue`, `XMLSequence`, `updateXML`, `insertChildXML`, and `deleteXML`.

Chapter 7, "XML Schema Storage and Query: Advanced"

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

Chapter 8, "XML Schema Evolution"

Describes how to update an XML schema registered with Oracle XML DB manually or using PL/SQL procedure `DBMS_XMLSCHEMA.copyEvolve`.

Chapter 9, "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 10, "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 SQL functions `contains` (Oracle Text) and `existsNode`. 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 11, "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.

Chapter 12, "Package DBMS_XMLSTORE"

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

Chapter 13, "Java API for XMLType"

Describes how to use the Java Database Connectivity (JDBC) API for `XMLType`.

Chapter 14, "Using the C API for XML"

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 15, "Using Oracle Data Provider for .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

Introduces you to ways you can view your existing data as XML.

Chapter 16, "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`. This chapter also describes how to use package `DBMS_XMLGEN`, XSQL Pages Publishing Framework, and XML SQL Utility (XSU) to generate XML data from data stored in the database.

Chapter 17, "Using XQuery with Oracle XML DB"

Describes Oracle XML DB support for the XQuery 1.0 language developed by W3C. SQL function `XMLQuery` is used to construct or query XML data using XQuery. SQL function `XMTable` is used to create relational values from XQuery results.

Chapter 18, "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 19, "Accessing Data Through URIs"

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 package `URIFACTORY`, how to use SQL function `sys_DburiGen`, 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 20, "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 21, "Managing Resource Versions"

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

Chapter 22, "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 23, "PL/SQL Access Using DBMS_XDB"

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

Chapter 24, "Repository Resource Security"

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

Chapter 25, "FTP, HTTP(S), 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 26, "User-Defined Repository Metadata"

Describes how to create and use custom XML metadata, which you associate with XML data and store in Oracle XML DB Repository. You can use such metadata to group or classify data, in order to process it in different ways.

Chapter 27, "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 28, "Administering Oracle XML DB"

Describes how to install, update, and configure Oracle XML DB.

Chapter 29, "Loading XML Data Using SQL*Loader"

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

Chapter 30, "Importing and Exporting XMLType Tables"

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

Chapter 31, "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.

Part VII, Appendixes

Provides background material for developing XML applications with Oracle XML DB.

Appendix A, "XML Schema Primer"

Provides a summary of the W3C XML Schema Recommendation.

Appendix B, "XPath and Namespace Primer"

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

Appendix C, "XSLT Primer"

Provides an introduction to the W3C XSL/XSLT Recommendation.

Appendix D, "Oracle-Supplied XML Schemas and Examples"

Describes the `RESOURCE_VIEW` and `PATH_VIEW` structures and lists the sample resource XML schema supplied by Oracle XML DB.

Appendix E, "Oracle XML DB Restrictions"

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 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 Database XML Java API Reference*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database Error Messages*. Oracle Database error message documentation is available only as HTML. If you have access to only printed or PDF Oracle Database documentation, you can browse the error messages by range. Once you find the specific range, use the "find in page" feature of your Web browser to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle Database online documentation.
- *Oracle Text Application Developer's Guide*
- *Oracle Text Reference*
- *Oracle Database Concepts.*
- *Oracle Database Java Developer's Guide*
- *Oracle Database Application Developer's Guide - Fundamentals*
- *Oracle Streams Advanced Queuing User's Guide and Reference*
- *Oracle Database PL/SQL Packages and Types Reference*

Many of the examples in this book use the sample schemas, which are installed by default when you select the Basic Installation option with an Oracle Database installation. Refer to *Oracle Database Sample Schemas* for information 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://www.oracle.com/technology/membership/`

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

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

For additional information, see:

- http://www.w3.org/XML/Schema – XML Schema
- http://www.w3.org/2001/XMLSchema – XML Schema
- http://www.w3.org/TR/xmlschema-0/ – XML Schema: primer
- http://www.w3.org/TR/xmlschema-1/ – XML Schema: structures
- http://www.w3.org/TR/xmlschema-2/ – XML Schema: datatypes
- http://www.oasis-open.org/cover/schemas.html – XML Schema
- http://www.xml.com/pub/a/2000/11/29/schemas/part1.html – XML Schema
- http://xml.coverpages.org/xmlMediaMIME.html – media/MIME types
- http://www.w3.org/TR/xptr/ – XPointer
- http://www.w3.org/TR/xpath – XPath 1.0
- http://www.w3.org/TR/xpath20/ – XPath 2.0
- http://www.zvon.org/xxl/XPathTutorial/General/examples.html – XPath
- http://www.mulberrytech.com/quickref/XSLTquickref.pdf – XSLT and XPath
- *XML In a Nutshell*, by Elliotte Rusty Harold and W. Scott Means, O'Reilly, January 2001, http://www.oreilly.com/catalog/xmlnut/chapter/ch09.html
- http://www.w3.org/TR/2002/NOTE-unicode-xml-20020218/ – Unicode in XML
- http://www.w3.org/TR/REC-xml-names/ – namespaces
- http://www.w3.org/TR/xml-infoset/ – information sets
- http://www.w3.org/TR/xslt – XSLT
- http://www.oasis-open.org/cover/xsl.html – XSL
- http://www.zvon.org/HTMLonly/XSLTutorial/Books/Book1/index.html – XSL
- http://www.mulberrytech.com/xsl/xsl-list/ – XSL (forum)
- http://www.w3.org/2002/ws/Activity.html – Web services
- http://www.ietf.org/rfc/rfc959.txt – RFC 959: FTP Protocol Specification
- ISO/IEC 13249-2:2000, Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text, International Organization For Standardization, 2000

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

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, Recovery Manager 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 executable programs, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names and connect identifiers, 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';
```

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

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

Convention	Meaning	Example
`[ ]`	Anything enclosed in brackets is optional.	`DECIMAL (digits [ , precision ])`
`{ }`	Braces are used for grouping items.	`{ENABLE | DISABLE}`
`|`	A vertical bar represents a choice of two options.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
`...`	Ellipsis points mean repetition in syntax descriptions.	`SELECT col1, col2, ... , coln FROM employees;`
	In addition, ellipsis points can mean an omission in code examples or text.	`CREATE TABLE ... AS subquery;`
Other symbols	You must use symbols other than brackets ([ ]), braces ({ }), vertical bars (|), and ellipsis points (...) exactly 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. Because these terms are not case sensitive, you can use them in either UPPERCASE or lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates user-defined programmatic elements, such as 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 section describes the new features and functionality, enhancements, APIs, and product integration support added to Oracle XML DB for Oracle Database 10*g* Release 2 (10.2). New features information from previous releases is also retained to help those users migrating to the current release.

The following sections describe the new features in Oracle XML DB:

- Oracle Database 10g Release 2 (10.2) New Features in Oracle XML DB
- Oracle Database 10g Release 1 (10.1) New Features in Oracle XML DB

Oracle Database 10*g* Release 2 (10.2) New Features in Oracle XML DB

Support for XQuery Language

XQuery, the new W3C XML query language, is supported. SQL functions `XMLQuery` and `XMLTable` have been added: `XMLQuery` lets you construct XML data and query XML and relational data using the XQuery language. `XMLTable` lets you create relational tables and columns from XQuery query results. SQL*Plus command `xquery` has also been added, to let you execute XQuery expressions directly. See Chapter 17, "Using XQuery with Oracle XML DB".

New SQL Functions for Updating XML (DML)

New SQL functions have been added to help you update XML data in the database: `insertChildXML`, `appendChildXML`, `insertXMLbefore`, and `deleteXML`. These new functions complement the functionality already provided by SQL function `updateXML`. Like `updateXML`, they are generally used in SQL DML statements. The new functions let you add and remove XML nodes in various ways. They can perform updates that are more localized than with `updateXML`, which can greatly improve performance and make source code clearer and more concise. All of the functions are optimized using XPath Rewrite. See "Updating XML Instances and XML Data in Tables" on page 4-15.

SQL/XML Standard Compliance (SQL:2005 Standard Part 14)

Support for the developing SQL/XML standard has been extended. The following SQL functions have been added: `XMLPI`, `XMLComment`, `XMLRoot`, `XMLSerialize`, `XMLCDATA`, and `XMLParse`. Escaping of identifiers has also been updated, in accordance with a change to the SQL/XML standard. See "Generating XML Using SQL Functions" on page 16-3.

XML Schema-Based Resource Metadata

You can now add and manipulate custom metadata for Oracle XML DB Repository resources that are XML Schema-based (in addition to non-schema-based). Resource metadata can be used to improve query performance and resource management. See Chapter 26, "User-Defined Repository Metadata".

XPath Rewrite Enhancements

XPath Rewrite can handle additional operations on more complex XML Schema constructs, including substitution groups, derived XML Schema types (inheritance), and SQL/XML collections. Performance has improved significantly for querying and updating XML Schema-based `XMLType` data and SQL/XML views. See Chapter 6, "XPath Rewrite".

Support for HTTPS

Oracle Database can now be used with HTTPS (HyperText Transfer Protocol, HTTP 1.1 as defined in the RFC2616 specification). HTTPS is a secure-access protocol. It can be configured for the database using the Oracle XML DB configuration file, `xdbconfig.xml`. See "Configuring Secure HTTP (HTTPS)" on page 25-6.

Deprecation of Oracle XDK PL/SQL Packages

The (Java-based) Oracle XDK PL/SQL packages `XMLDOM`, `XMLPARSER`, and `XSL_PROCESSOR` have been deprecated in favor of the (C-based) Oracle XML DB packages `DBMS_XMLDOM`, `DBMS_XMLPARSER`, and `DBMX_XSLPROCESSOR`. Synonyms have been provided to smooth the migration of legacy applications. See "APIs for XML" on page 1-5.

ASM Virtual Folders

Automatic Storage Management (ASM) organizes database files into disk groups for simplified management and added benefits such as database mirroring and I/O balancing. DBAs can now access ASM resources in Oracle XML DB Repository using protocols and resource APIs (such as `DBMS_XDB`). ASM files are accessed in the virtual repository folder `/sys/asm`. See "Accessing ASM Files Using Protocols and Resource APIs – For DBAs" on page 20-10.

Support for Transportable Tablespaces

The Transportable Tablespace feature works with `XMLType` tables in Oracle XML DB. In particular, XML schemas are treated like any other database objects with respect to import and export: they are moved along with their associated tablespaces. See "Using Transportable Tablespaces with Oracle XML DB" on page 30-2.

Enterprise Manager Support for Oracle Database

Oracle Enterprise Manager can now be used to manage the following Oracle XML DB features:

- configuration parameters
- repository resources
- repository access control lists (ACLs)
- XML Schemas
- `XMLType` tables and columns

Oracle Database 10*g* Release 1 (10.1) New Features in Oracle XML DB

This section summarizes the Oracle XML DB enhancements provided with Oracle Database 10*g* Release 2 (10.2).

See Also:

- *Oracle Database 10g Release Notes*
- `http://www.oracle.com/technology/tech/xml/` for the latest Oracle XML DB updates and notes

Exporting and Importing XML Data

The IMPORT/EXPORT utility has been enhanced to help you load XML data into Oracle XML DB. See Chapter 30, "Importing and Exporting XMLType Tables".

XML Schema Evolution Support

Oracle Database 10*g* supports XML schema evolution by providing PL/SQL procedure `copyEvolve` as part of package `DBMS_XMLSCHEMA`.

In prior releases there was no standard procedure for schema evolution. Once registered with Oracle XML DB at a particular URL, an XML schema could not be modified, in case there were `XMLType` tables dependent on the schema.

Hierarchical Queries with DBMS_XMLGEN

Package `DBMS_XMLGEN` now supports hierarchical queries. See Generating XML Using DBMS_XMLGEN on page 16-24.

Character Conversion and Multibyte Characters

In Oracle Database 10*g* Release 1 (10.1), XML data retrieved from the database is automatically converted to your client character set. In addition, using FTP or HTTP, you can use multibyte characters in a directory name, filename, or URL, and you can transfer or receive data encoded in a different character set from the database. 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 Oracle XML Developer's Kit (XDK) and Oracle XML DB. This is a DOM API that can be used with XML inside or outside the database. See Chapter 14, "Using the C API for XML".

See Also:

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

SQL*Loader Supports XMLType Tables and Columns Independent of Storage

In Oracle Database 10*g* Release 1 (10.1), SQL*Loader supports `XMLType` tables and columns. It can load `XMLType` data, regardless of whether the data is stored as LOBs or in an object-relational manner. See Chapter 29, "Loading XML Data Using SQL*Loader".

Disabling Pretty-Printing with DBMS_XMLGEN

Package `DBMS_XMLGEN` now has an option to turn off pretty-printing.

Oracle Text Enhancements

Oracle Database 10*g* Release 1 (10.1) offers the following Oracle Text enhancements:

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

See Also: Chapter 10, "Full-Text Search Over XML"

Oracle Streams Advanced Queuing (AQ) Support

Oracle Streams Advanced Queuing (AQ) Internet Data Access Presentation (iDAP) has been enhanced: you can now use the AQ XML servlet to access Oracle Database AQ using HTTP and Simple Object Access Protocol (SOAP). IDAP facilitates using AQ over the Internet.

IDAP is now the SOAP implementation for AQ operations; it 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 embedding `XMLType` as an attribute in an Oracle Database object type.

See Also:

- Chapter 31, "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 16-56
- "Generating XML Using XML SQL Utility (XSU)" on page 16-59
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database XML Java API Reference*
- *Oracle Database XML C API Reference*

Part I

Oracle XML DB Basics

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

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

1

Introduction to Oracle XML DB

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

Features of Oracle XML DB

Oracle XML DB is the name for a set of Oracle Database technologies related to high-performance XML storage and retrieval. It provides native XML support by encompassing both SQL and XML data models in an interoperable manner.

Oracle XML DB includes the following features:

- Support for the World Wide Web Consortium (W3C) XML and XML Schema data models and standard access methods for navigating and querying XML. The data models are incorporated into Oracle Database.
- Ways to store, query, update, and transform XML data while accessing it using SQL.
- Ways to perform XML operations on SQL data.
- A simple, lightweight XML repository where you can organize and manage database content, including XML, using a file/folder/URL metaphor.
- A storage-independent, content-independent and programming language-independent infrastructure for storing and managing XML data. This provides new ways of navigating and querying XML content stored in the database. For example, Oracle XML DB Repository facilitates this by managing XML document hierarchies.

- Industry-standard ways to access and update XML. The standards include the W3C XPath recommendation and the ISO-ANSI SQL/XML standard. FTP, HTTP(S), and WebDAV can be used to move XML content into and out of Oracle Database. Industry-standard APIs provide programmatic access and manipulation of XML content using Java, C, and PL/SQL.
- XML-specific memory management and optimizations.
- Enterprise-level Oracle Database features for XML content: reliability, availability, scalability, and security.

Oracle XML DB can be used in conjunction with Oracle XML Developer's Kit (XDK) to build applications that run in the middle tier in either Oracle Application Server or Oracle Database.

See Also: *Oracle XML Developer's Kit Programmer's Guide*

Oracle XML DB Architecture

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

- Storage of `XMLType` tables and views
- Oracle XML DB Repository

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

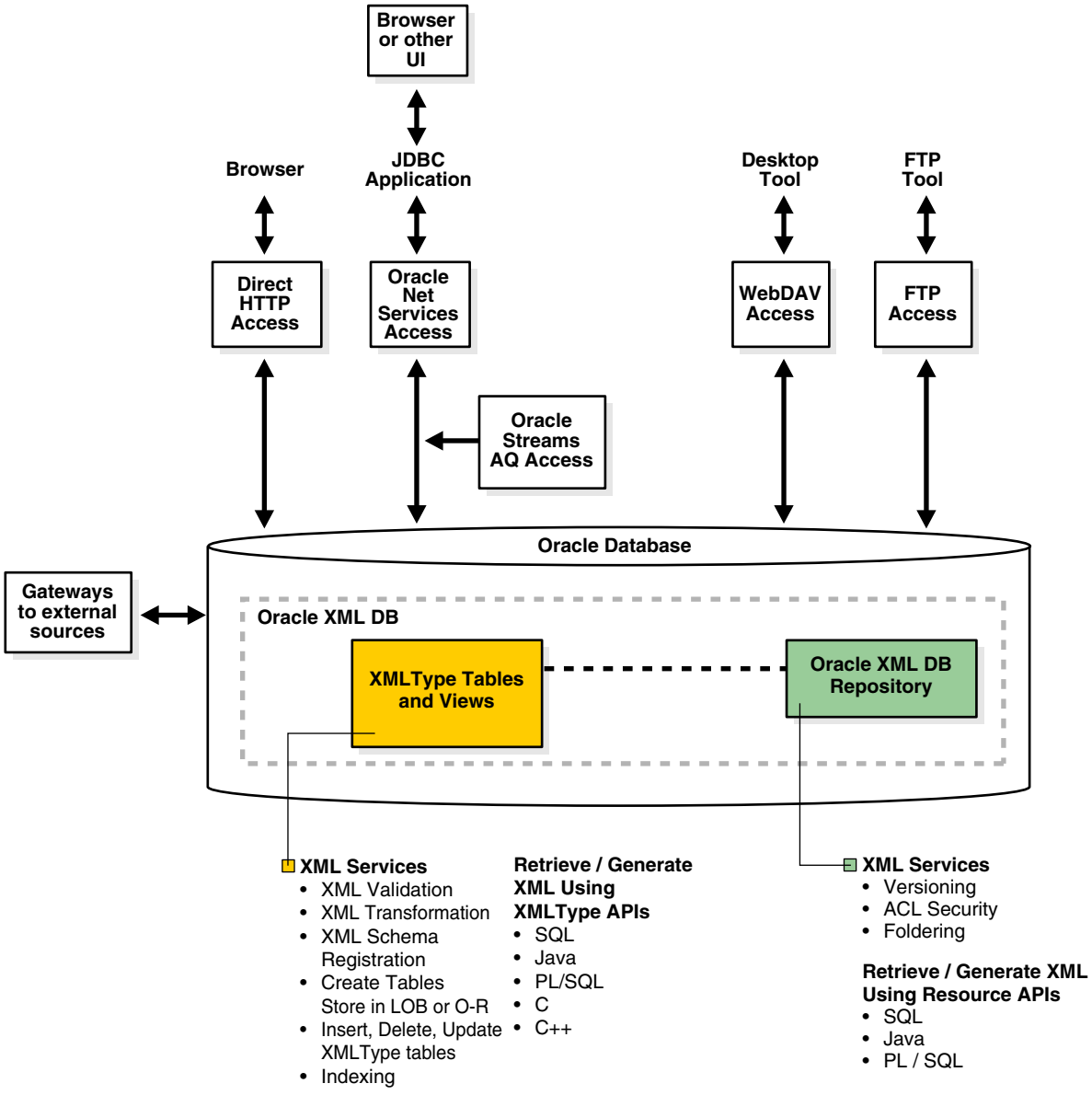
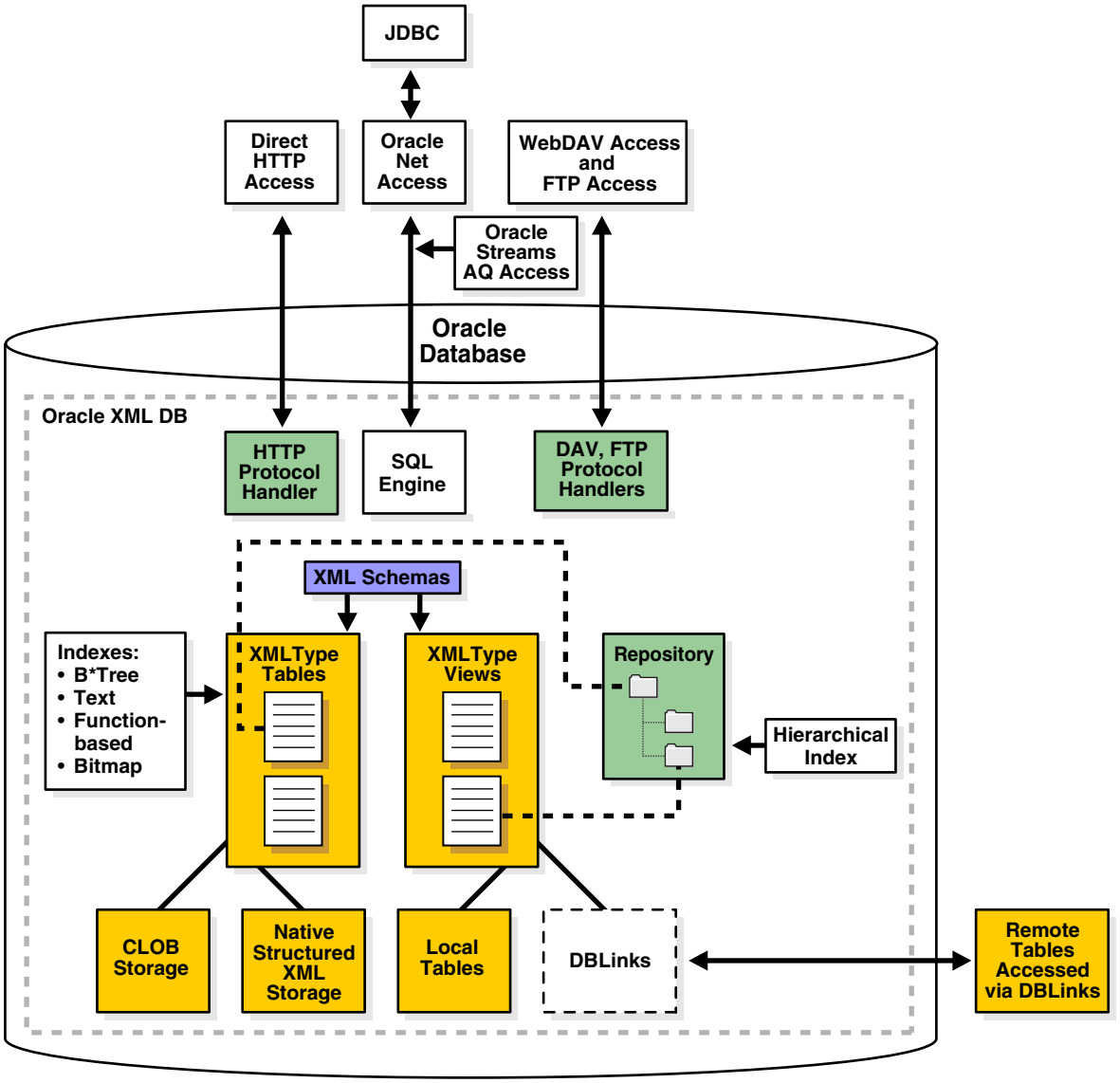


Figure 1–2 Oracle XML DB Architecture: XMLType Storage



XMLType Storage

Figure 1–2 shows `XMLType` storage in Oracle XML DB.

When XML schemas are registered with Oracle XML DB, a set of default tables are created and used to store XML instance documents associated with the schema. These documents can be viewed and accessed in Oracle XML DB Repository.

`XMLType` tables and columns can be stored as Character Large Object (`CLOB`) values or as a set of objects. When stored as a set of objects, we refer to **structured**, or **shredded** storage.

natively, using structured XML, or in Character Large Object (`CLOB`) values.

Data in `XMLType` views can be stored in local or remote tables. Remote tables can be accessed through database links.

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

You can access data in Oracle XML DB Repository using any of the following:

- HTTP(S), through the HTTP protocol handler.
- WebDAV and FTP, through the WebDAV and FTP protocol server.
- SQL, through Oracle Net Services, including Java Database Connectivity (JDBC).

Oracle XML DB 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 25, "FTP, HTTP(S), and WebDAV Access to Repository Data"
- Chapter 31, "Exchanging XML Data with Oracle Streams AQ"

APIs for XML

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

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

Table 1–1 APIs Related to XML

API	Documentation	Description
`XMLType`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "XMLType", *Oracle Database XML C API Reference*, and *Oracle Database XML C++ API Reference*	PL/SQL, C, and C++ APIs with XML operations on `XMLType` data – validation, transformation.
Database URI types	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "Database URI TYPEs"	Functions used for various URI types.
`DBMS_XDB`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDB"	PL/SQL API for managing Oracle XML DB Repository resources, ACL-based security, and configuration sessions.
`DBMS_XDB_VERSION`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDB_VERSION"	PL/SQL API for version management of repository resources.
`DBMS_XDBT`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDBT"	PL/SQL API for creation of text indexes onrepository resources.
`DBMS_XDBZ`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDBZ"	Oracle XML DB Repository ACL-based security.
`DBMS_XMLDOM`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLDOM"	PL/SQL implementation of the DOM API for `XMLType`.
`DBMS_XMLGEN`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLGEN"	PL/SQL API for transformation of SQL query results into canonical XML format.

Table 1–1 (Cont.) APIs Related to XML

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

XML Schema Catalog Views

Table 1–2 lists the XML schema catalog views for Oracle XML DB, which provide access to metadata about XML schemas that are registered with Oracle XML DB. Information about a given view can be obtained by using the SQL command DESCRIBE. Example:

```
DESCRIBE USER_XML_SCHEMAS
```

Table 1–2 XML Schema Catalog Views

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

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Views RESOURCE_VIEW and PATH_VIEW

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

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

See Also: Chapter 22, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"

Overview of Oracle XML DB Repository

Oracle XML DB Repository is a component of Oracle Database that is optimized for handling XML data. The Oracle XML DB repository contains **resources**, which can be either **folders** (directories, containers) or files. Each resource has these properties:

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

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

See Also:

- Part V. "Oracle XML DB Repository: Foldering, Security, and Protocols"
- Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data" for information on accessing XML data in `XMLType` tables and columns using external protocols
- Chapter 26, "User-Defined Repository Metadata"

Accessing and Manipulating XML in the Oracle XML DB Repository

You can access data in Oracle XML DB Repository in the following ways (see Figure 1–1):

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

XML Services

Besides supporting APIs that access and manipulate data, Oracle XML DB Repository provides APIs 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 a resource create a new version (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 Access Control Lists (ACLs). Every resource 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 manages a persistent hierarchy of folder (directory) resources that contain other resources (files or folders). Oracle XML DB modules, such as protocol servers, the schema manager, and the Oracle XML DB `RESOURCE_VIEW` API, use foldering to map path names to resources.

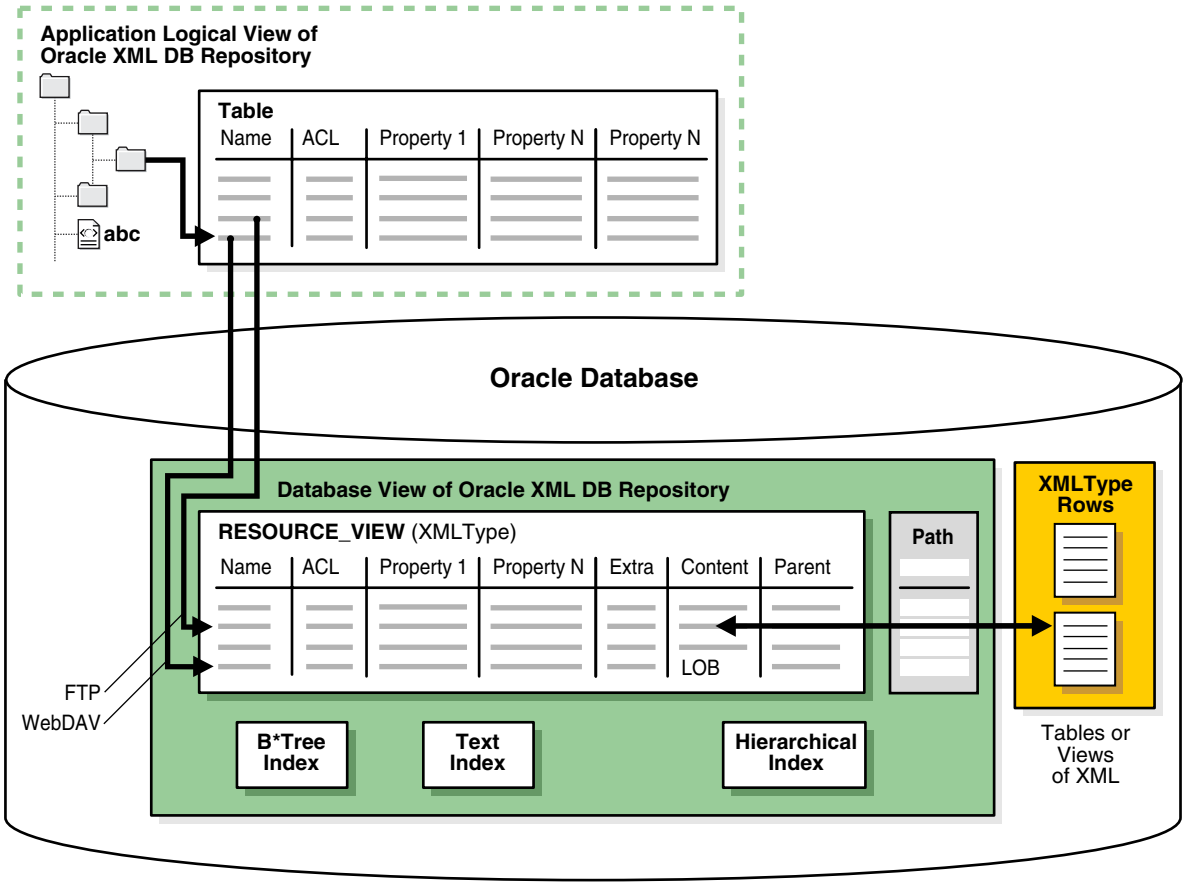
Oracle XML DB Repository Architecture

Figure 1–3 describes the Oracle XML DB Repository architecture. You can access the repository in SQL, for example, 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 20, "Accessing Oracle XML DB Repository Data"
- Chapter 22, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"

Figure 1–3 Oracle XML DB Repository Architecture



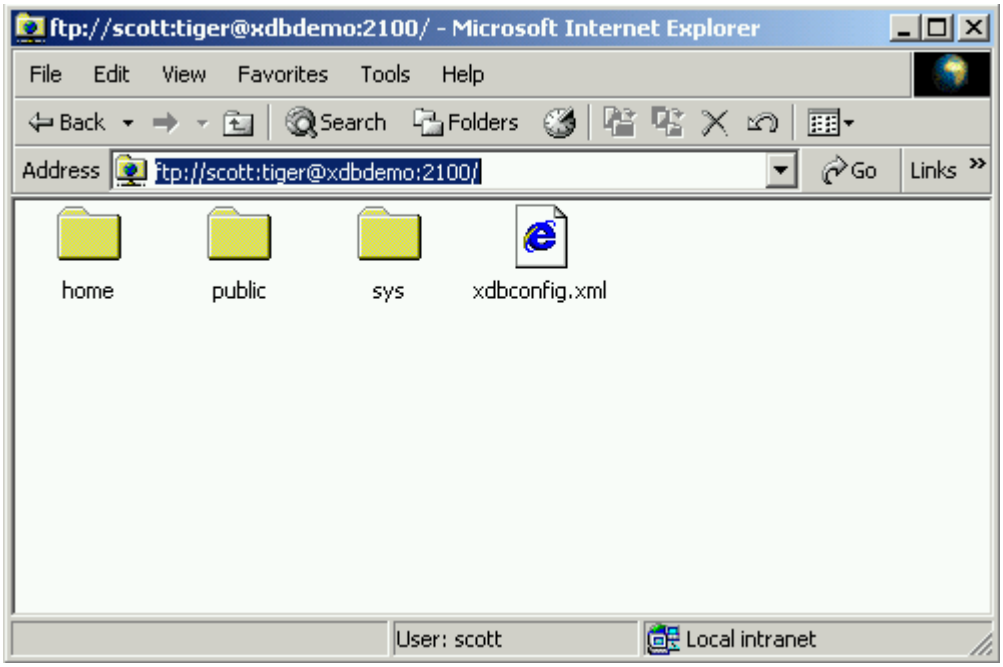
How Does Oracle XML DB Repository Work?

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 semistructured 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 poor at managing hierarchical structures and traversing a path or URL. Oracle XML DB provides a hierarchically organized 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.
- You can access XML documents in Oracle XML DB Repository using standard connect-access protocols such as FTP, HTTP(S), and WebDAV, in addition to languages SQL, PL/SQL, Java, and C. The repository provides content authors and editors direct access to XML content stored in Oracle Database.

- A resource in this context is a file or folder, identified by a URL. WebDAV is an IETF standard that defines a set of extensions to the HTTP protocol. It allows an HTTP server to act as a file server for a DAV-enabled client. For example, a WebDAV-enabled editor can interact with an HTTP/WebDAV server as if it were a file system. The WebDAV standard uses the term **resource** to describe a file or a folder. Each resource managed by a WebDAV server is identified by a URL. Oracle XML DB adds native support to Oracle Database for these protocols. The protocols were designed for document-centric operations. By providing support for these protocols, Oracle XML DB 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 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"

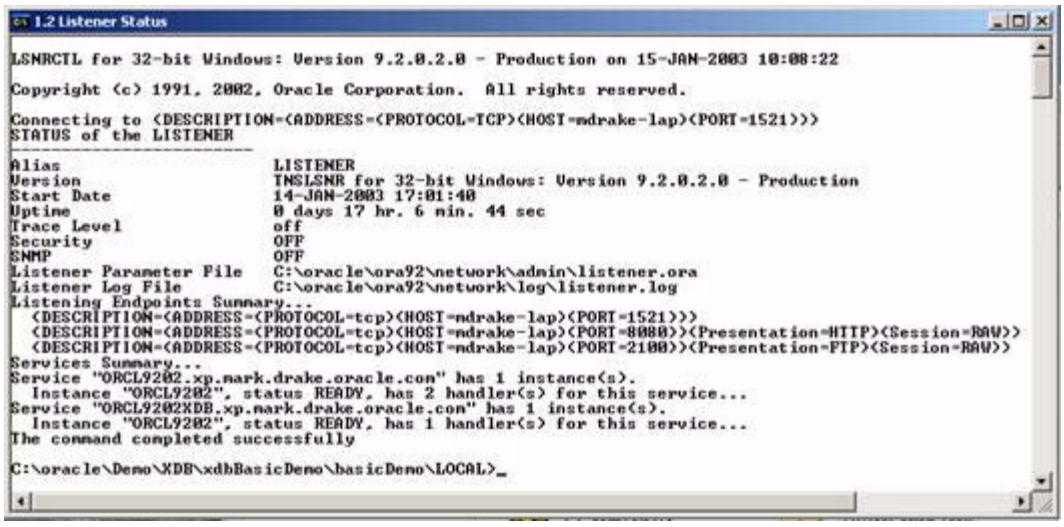
Hence, WebDAV clients such as Microsoft Windows Explorer can connect directly to Oracle XML DB Repository. No additional Oracle Database or Microsoft-specific software or other complex middleware is needed. End users can work directly with Oracle XML DB Repository using familiar tools and interfaces.

Oracle XML DB Protocol Architecture

One key feature of the Oracle XML DB architecture is that HTTP(S), 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(S) and FTP requests in the same way that it listens for ODP .NET service requests. When the Listener receives an HTTP(S) or FTP request, it hands it off to an Oracle Database shared server process which services it and sends the appropriate response back to the client.

You can use the TNS Listener command `lsnrctl status` to verify that HTTP(S) and FTP support has been enabled – see Figure 1–5.

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



See Also: Chapter 25, "FTP, HTTP(S), 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. The most popular ways to build web-based applications are these:

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

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

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

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

Oracle XML DB Features

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

The hierarchical nature of XML presents the traditional relational database with 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 the *node hierarchy* of an XML document to access part of an XML document.

Oracle XML DB addresses these challenges by introducing new SQL functions 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 these:

- XMLType Datatype
- XML Schema Support
- Structured Versus Unstructured Storage
- XML/SQL Duality
- SQL/XML INCITS Standard SQL Functions
- Rewriting of XPath Expressions: XPath Rewrite
- XMLType Storage. This was described previously on page 1-4.
- Overview of Oracle XML DB Repository. This was described previously on page 1-7.

XMLType Datatype

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

You can use `XMLType` 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

Datatype `XMLType` provides the following:

- **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 operate on `XMLType` instances. `XMLType` methods provide support for the following common operations:
  - Extract a subset of nodes contained in the `XMLType` – method `extract()`.
  - Check whether or not a particular node exists in the `XMLType` – method `existsNode()`.
  - Validate the contents of the `XMLType` against an XML schema – method `schemaValidate()`.
  - Perform an XSL Transformation – method `transform()`.

See Also: Chapter 4, "XMLType Operations" and Chapter 9, "Transforming and Validating XMLType Data"

XML Schema Support

Support for the Worldwide Web Consortium (W3C) XML Schema Recommendation is a key feature in Oracle XML DB. XML Schema specifies the structure, content, and certain semantics of 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 semistructured. 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 schemas
- 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 object that is based on an XML schema or not based on an XML schema:

- ***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-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 an XML schema, and the 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 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 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 schemas are most commonly used as a mechanism for checking whether instance documents conform with their specifications (validation). Oracle XML DB includes methods and SQL functions that allow an XML schema to be used for this.

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

See Also: Appendix A, "XML Schema Primer" and Chapter 5, "XML Schema Storage and Query: Basic" for more information about using XML schemas 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 storage* 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 1–3 outlines the merits of structured and unstructured storage.

Table 1–3 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 documents that conform to 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 XPath 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 XPath rewrite, leading to significantly improved performance, particularly with large collections of documents.

Table 1–3 (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 semistructured 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 semistructured and unstructured content.

Replacing proprietary file formats with XML allows organizations to achieve much higher levels of reuse of their semistructured and unstructured data. The content can be accurately described using XML schemas. 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 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 for 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 semistructured 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 functions 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 INCITS Standard SQL Functions

Oracle XML DB provides the SQL functions defined in the SQL/XML standard. The SQL/XML standard is defined by specifications prepared by the International Committee for Information Technology Standards (INCITS) Technical Committee H2. INCITS is the main standards body for developing standards for the syntax and semantics of database languages, including SQL.

The SQL/XML standard is an evolving standard, so the syntax and semantics of its functions are subject to change in the future. The Oracle XML DB implementation of SQL/XML functions will evolve accordingly.

SQL/XML functions fall into two categories:

- Functions that make it possible to *query and access XML* content as part of normal SQL operations.
- Functions that provide a standard way of *generating XML* from the result of a SQL `SELECT` statement.

With the SQL/XML functions 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 ability to embed XPath expressions in SQL statements greatly simplifies XML access. The following describes briefly some of the more important SQL/XML functions:

- **existsNode** – This is used in the `WHERE` clause of a SQL statement to restrict the set of documents returned by a query. The `existsNode` SQL function takes an XPath expression and applies it to an XML document. The function returns true (`1`) or false (`0`), depending on whether or not the document contains a node that 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, then the result is a well-formed XML document. If multiple nodes match the XPath expression, then the result is a document fragment.
- **extractValue** – This takes an XPath expression and returns the corresponding leaf node. The XPath expression passed to `extractValue` should identify a single attribute or an element that has precisely one text node child. The result is returned in the appropriate SQL data type. Function `extractValue` is essentially a shortcut for `extract` plus either `getStringVal()` or `getNumberVal()`.
- **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. SQL function `updateXML` 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

See Also:

- `http://www.incits.org/tc_home/h2.htm` for information on INCITS Technical Committee H2
- `http://www.w3.org/TR/xpath` for the XPath recommendation
- Chapter 4, "XMLType Operations" for detailed descriptions of the SQL/XML standard functions for *querying* XML data
- Generating XML Using SQL Functions on page 16-3 for detailed descriptions of the SQL/XML standard functions for *generating* XML data
- Chapter 3, "Using Oracle XML DB" for *examples* that use the SQL/XML standard functions

Rewriting of XPath Expressions: XPath Rewrite

The SQL/XML SQL functions and their 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.

See Also: "Generating XML Using SQL Functions" on page 16-3 for information on SQL/XML functions

How XPath Expressions are Evaluated by Oracle XML DB

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

- **Structured-storage XML data:** Oracle XML DB attempts to translate the XPath expression in a SQL/XML function into an equivalent SQL query. The SQL query references the object-relational data structures that underpin a schema-based `XMLType`. This process is referred to as **XPath rewrite**. It can occur when performing queries and `UPDATE` operations.
- **Unstructured-storage XML data:** Oracle XML DB evaluates the XPath expression using functional evaluation. Functional evaluation builds a DOM tree for each XML document, and then resolves the XPath programmatically using the methods provided by the DOM API. If the operation involves updating the DOM tree, the entire XML document has to be written back to disc when the operation is completed.

Efficient Processing of SQL That Contains XPath Expressions

Oracle XML DB can rewrite SQL statements that contain XPath expressions to purely relational SQL statements, which can be processed efficiently. In this way, Oracle XML DB insulates the database optimizer from having to understand XPath notation and the XML data model. The database optimizer simply processes the rewritten 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 XPath rewrites, and Oracle XML DB can execute XPath-based queries at near-relational speed, while preserving the XML abstraction.

When Can XPath Rewrite Occur?

XPath rewrite is possible when:

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

What is the XPath-Rewrite Process?

XPath rewrite performs the following tasks:

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

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

In general, functional evaluation of a SQL statement is more expensive than XPath rewrite, particularly if the number of documents 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` value), functional evaluation is necessary any time SQL functions except `existsNode` are used. Function `existsNode` 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 XPath rewrite, and the conditions under which it can take place, is a key step in developing Oracle XML DB applications that will deliver the required levels of scalability and performance.

See Also: Chapter 6, "XPath Rewrite"

Oracle XML DB Benefits

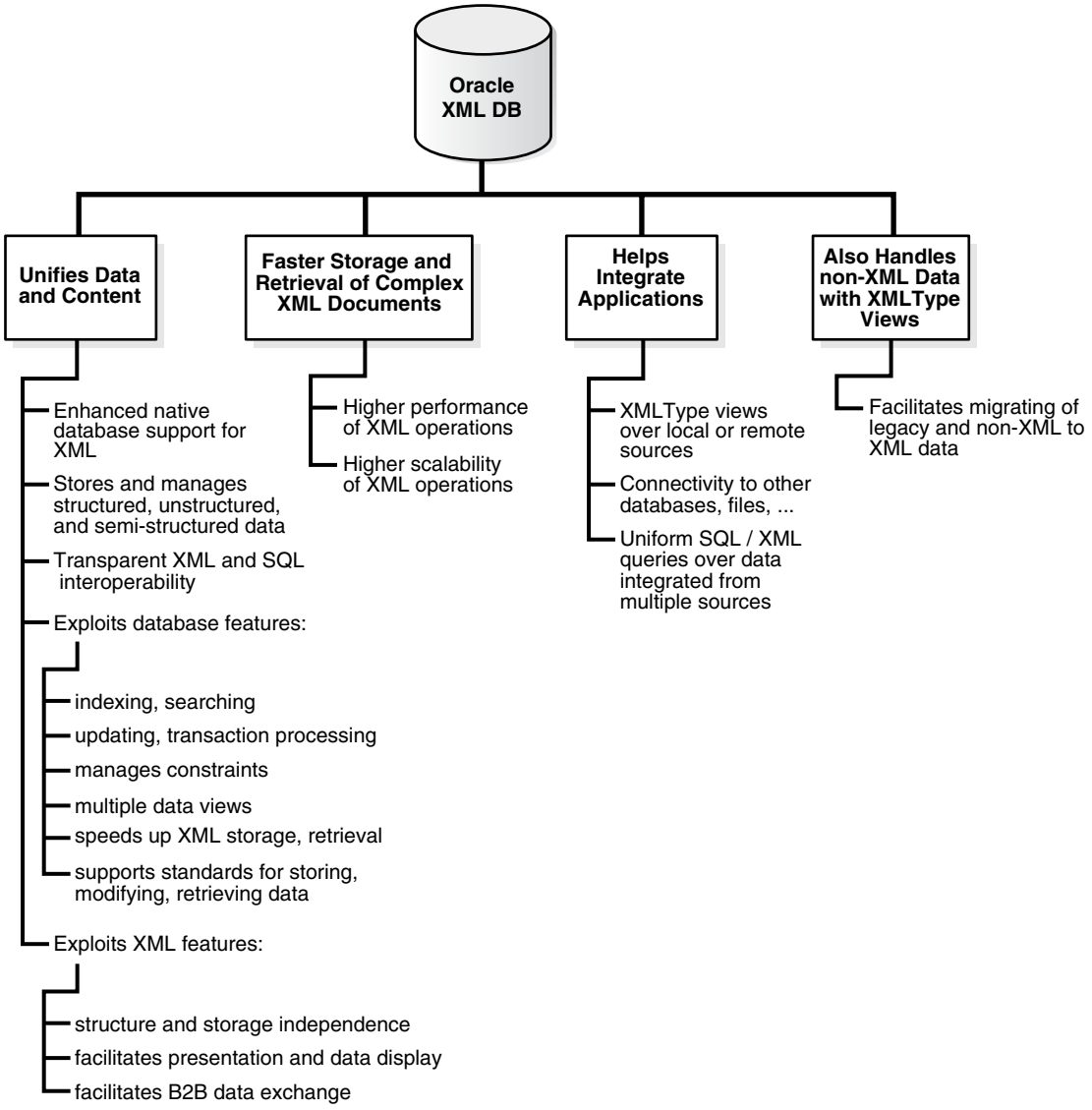
The following sections describe several benefits for using Oracle XML DB advantages including:

- 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.

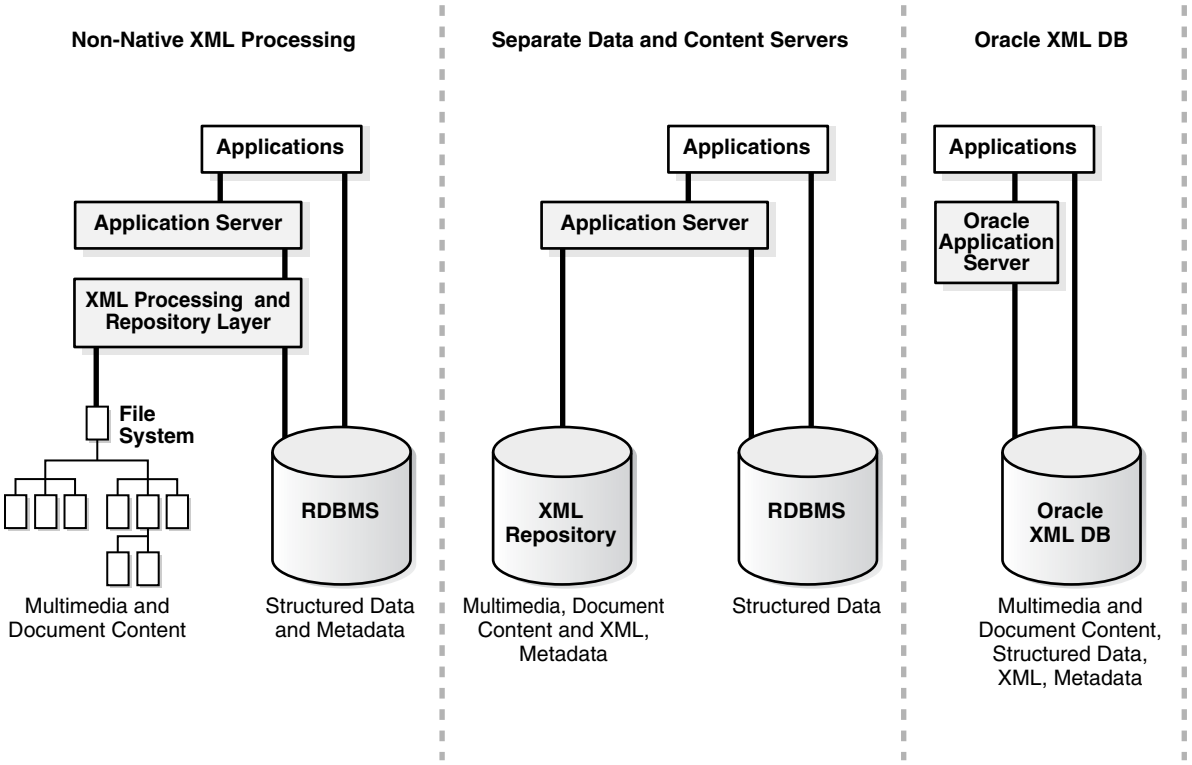
Figure 1–6 Oracle XML DB Benefits



Unifying Data and Content with Oracle XML DB

Most application data and Web content is stored in a relational database or a file system, or both. XML is often used for transport, and it 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.

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 data that is structured, unstructured, and semistructured using a standard data model and standard SQL and XML. You can perform SQL operations on XML documents and XML operations on object-relational (such as table) data.

Exploiting Database Capabilities

Oracle Database has 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 10, "Full-Text Search Over XML", and Chapter 16, "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: Basic" and the purchase-order examples in Chapter 4, "XMLType Operations".
- ***Multiple Views of Data***: Most enterprise applications need to group data together in different ways for different modules. This is why relational views are necessary—to allow for these multiple ways to combine data. By allowing views on XML, Oracle XML DB creates different logical abstractions on XML for, say, consumption by different types of applications. See Chapter 18, "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 13, "Java API for XMLType"
  - Chapter 22, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"
  - Chapter 23, "PL/SQL Access 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: Basic".
- ***Storage Independence***: When you use relational design, your client programs must know where your data is stored, in what format, what table, and what the relationships are among those tables. `XMLType` enables you to write applications without that knowledge and allows DBAs to map structured data to physical table and column storage. See Chapter 5, "XML Schema Storage and Query: Basic" and Chapter 20, "Accessing Oracle XML DB Repository Data".
- ***Ease of Presentation***: XML is understood natively by Web browsers, many popular desktop applications, and most Internet applications. Relational data is not generally accessible directly from applications; programming is required to make relational data accessible to standard clients. Oracle XML DB stores data as XML and makes it available as XML outside the database; no extra programming is required to display database content. See:

- Chapter 9, "Transforming and Validating XMLType Data".
- Chapter 16, "Generating XML Data from the Database".
- Chapter 18, "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 9, "Transforming and Validating XMLType Data"
  - Chapter 18, "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 11, "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 9, "Transforming and Validating XMLType Data".
- XML schema-caching support. See Chapter 5, "XML Schema Storage and Query: Basic".
- CTXPATH Text indexing. See Chapter 10, "Full-Text Search Over XML".
- The hierarchical index over Oracle XML DB Repository. See Chapter 20, "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 18, "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 SQL functions 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. SQL functions `hasPath` and `inPath` are designed to optimize XML data searches where you can search within XML text for substring matches.

Oracle XML DB also provides:

- SQL function `contains` and XPath function `ora:contains`, which can be used with SQL function `existsNode` for XPath-based searches.
- The ability to create indexes on `URIType` and `XDBURIType` columns.
- Index type `CTXXPATH`, which allows higher performance XPath searching using `existsNode`.

See Also:

- Chapter 10, "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 31, "Exchanging XML Data with Oracle Streams AQ"

Requirements for Running Oracle XML DB

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

See Also:

- `http://www.oracle.com/technology/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(S) 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(S), FTP, and 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).
- 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 Also:

- "SQL/XML INCITS Standard SQL Functions" for more information on the ANSI SQL/XML functions
- Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data" for more information on protocol support
- Chapter 27, "Writing Oracle XML DB Applications in Java" for information on using the Java servlet
- Chapter 31, "Exchanging XML Data with Oracle Streams AQ" and *Oracle Streams Advanced Queuing User's Guide and Reference.* for information on using SOAP

Oracle XML DB Technical Support

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

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

Oracle XML DB Examples Used in This Manual

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

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

Further Oracle XML DB Case Studies and Demonstrations

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

Oracle XML DB Examples and Tutorials

You can peruse more Oracle XML DB examples on OTN:

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

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

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://www.oracle.com/technology/sample_code/tech/xml/xmldb/cmsxdb/content.html.
- XML Dynamic News. This is a complete J2EE 1.3 based application that demonstrates Java and Oracle XML DB features for an online news portal. News feeds are stored and managed persistently in Oracle XML DB. Various other news portals can customize this application to provide static or dynamic news services to end users. End users can personalize their news pages by setting their preferences. The application also demonstrates the use of Model View Controller (MVC) architecture and various J2EE design patterns. See http://www.oracle.com/technology/sample_code/tech/xml/xmlnews/content.html
- SAX Loader Application. This demonstrates an efficient way to break up large files containing multiple XML documents outside the database and insert them into the database as a set of separate documents. This is provided as a standalone and a web-based application.
- Oracle XML DB Utilities package. This highlights the subprograms provided with the `XDB_Utilities` package. These subprograms operate on `BFILE` values,

`CLOB` values, DOM, and Oracle XML DB Resource APIs. With this package, you can perform basic Oracle XML DB foldering operations, read and load XML files into a database, and perform basic DOM operations through PL/SQL.

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

2

Getting Started with Oracle XML DB

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

This chapter contains these topics:

Oracle XML DB Installation

Oracle XML DB is installed automatically in the following situations:

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

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

- User XDB exists. To check: `SELECT * FROM ALL_USERS;`
- View `RESOURCE_VIEW` exists. To check: `DESCRIBE RESOURCE_VIEW`

For a *manual* installation or de-installation of Oracle XML DB, see Chapter 28, "Administering 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(S)/WebDAV, or Java Database Connectivity (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 unstructured 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), semistructured, or mostly unstructured? If highly structured, will your tables 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 16, "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 9, "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 2-6.

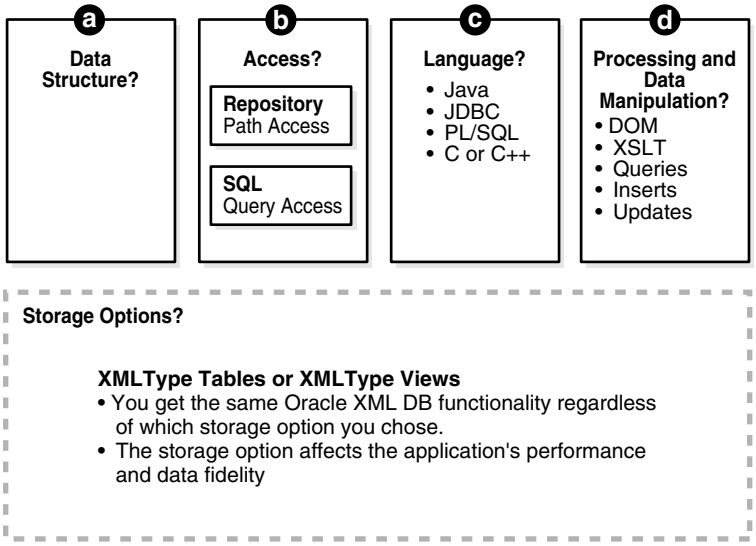
Will you be exchanging XML data with other applications, across gateways? Will you need Advanced Queuing (AQ) or SOAP compliance? See Chapter 31, "Exchanging XML Data with Oracle Streams AQ".

E. Storage

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

Note: The choices you make for A–D are typically interdependent, but they are not dependent on the storage model you choose (E).

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:

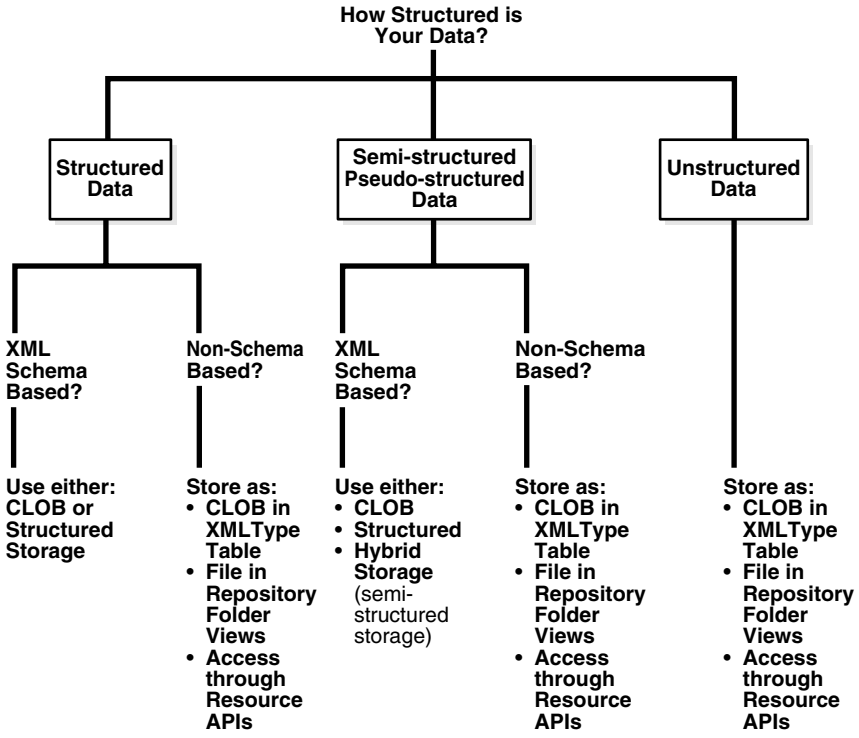
- **Structured data.** Is your data mostly XML data?
- **Semi/pseudo-structured data**. Does your data include some XML data?
- **Unstructured data**. Is most of your data not XML data?

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 semistructured data, you can use either `CLOB`, structured, or hybrid storage. Here your XML schema can be more loosely coupled. See also "Oracle XML DB Application Design: F. 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 Oracle XML DB 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 and path views. See Chapter 22, "SQL Access Using RESOURCE_VIEW and PATH_VIEW".
  - PL/SQL access through `DBMS_XDB`. See Chapter 23, "PL/SQL Access Using DBMS_XDB".
  - Protocol-based access using HTTP(S)/WebDAV or FTP, most suited to content-oriented applications. See Chapter 25, "FTP, HTTP(S), 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 access (through JDBC). See Java API for XMLType.
  - PL/SQL access. See Chapter 11, "PL/SQL API for XMLType".

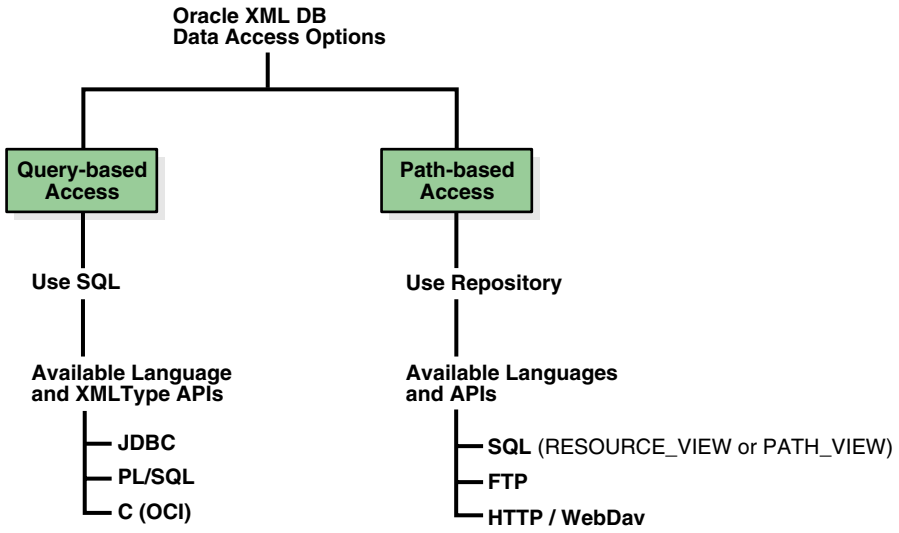
These options for accessing Oracle XML DB Repository data are also discussed in Chapter 20, "Accessing Oracle XML DB Repository Data".

You can also consider the following access model criteria:

- What level of security do you need? See Chapter 24, "Repository 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 10, "Full-Text Search Over XML".
- Do you need to version the data? If yes, see Chapter 21, "Managing 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 13, "Java API for XMLType"
  - Chapter 27, "Writing Oracle XML DB Applications in Java"

- PL/SQL

  See Also:

  - Chapter 11, "PL/SQL API for XMLType"
  - Chapter 23, "PL/SQL Access Using DBMS_XDB"
  - "APIs for XML" on page 1-5

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 with (validates against) specific XML schemas. See Chapter 9, "Transforming and Validating XMLType Data".

- DOM. See Chapter 11, "PL/SQL API for XMLType". Use object-relational columns, varray values, nested tables, as well as LOB values to store any element or element-subtree in your XML schema, and still maintain DOM fidelity.

  Note: If you choose the `CLOB` storage option, available with `XMLType` since Oracle9i release 1 (9.0.1), you can preserve whitespace. If you are using an XML schema, see the discussion on DOM fidelity in Chapter 5, "XML Schema Storage and Query: Basic".

- XPath searching. You can use XPath syntax embedded in a SQL statement or as part of an HTTP(S) request to query XML content in the database. See Chapter 4, "XMLType Operations", Chapter 10, "Full-Text Search Over XML", Chapter 20, "Accessing Oracle XML DB Repository Data", and Chapter 22, "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 16, "Generating XML Data from the Database".

How often will XML documents be accessed, updated, and manipulated? See Chapter 4, "XMLType Operations" and Chapter 22, "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: Basic".

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

Messaging 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 `CLOB` values.
- Selectively dequeue messages with `XMLType` attributes using SQL functions such as `existsNode` and `extract`.
- Define rule-based subscribers that query message content using SQL functions such as `existsNode` and `extract`.
- Define transformations to convert Oracle Database objects to `XMLType`.

See Also:

- Chapter 31, "Exchanging XML Data with Oracle Streams AQ"
- *Oracle Streams Advanced Queuing User's Guide and Reference*

Oracle XML DB Application Design: F. 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 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 (document) 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
- **Semistructured storage** – Semistructured, or 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, "Introduction to Oracle XML DB", "XMLType Storage" on page 1-4
- Chapter 3, "Using Oracle XML DB"
- Chapter 4, "XMLType Operations"
- Chapter 5, "XML Schema Storage and Query: Basic", "DOM Fidelity" on page 5-14

Using XMLType Tables

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

- `CLOB` (unstructured) storage
- Structured storage
- Semistructured storage

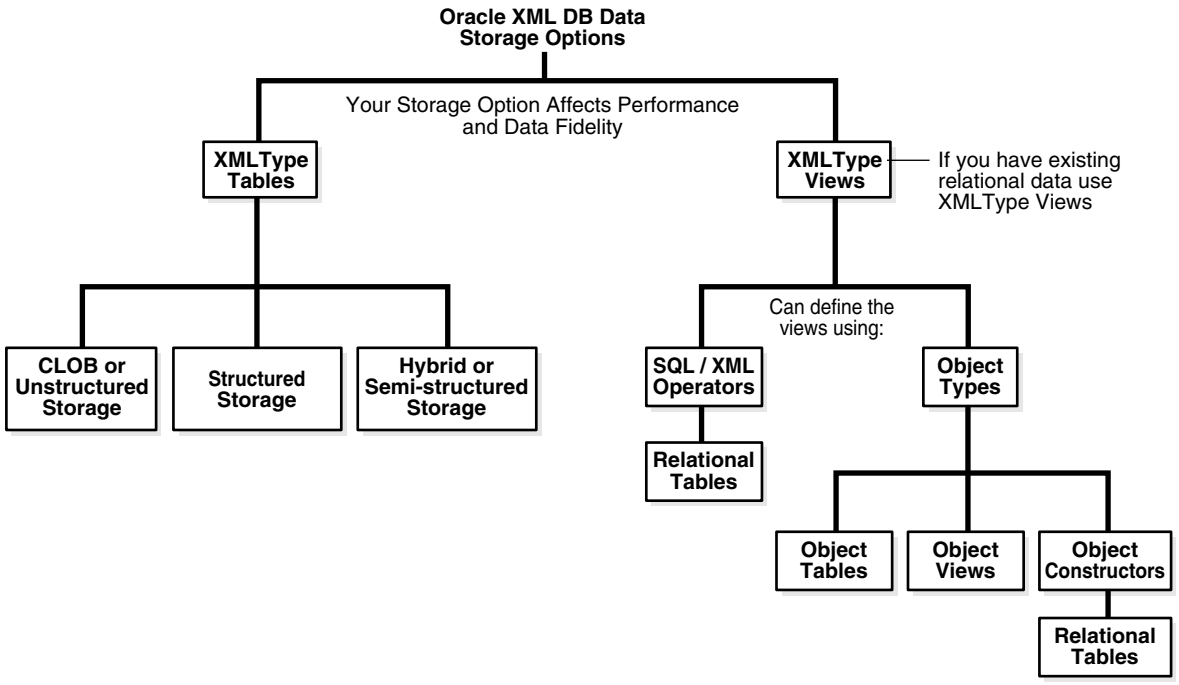
Note: Use the *thick* JDBC driver with schema-based `XMLType` values stored object-relationally. (You can use either the thin or the thick driver with `CLOB` storage of `XMLType` values.)

Using XMLType Views

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

- SQL/XML SQL functions. Using these functions you can store the data in relational tables and also generate/regenerate the XML. See Chapter 16, "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 Memory Management

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

XML Storage Requirements

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 main reasons for this:

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

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.

Minimizing XML Schema Overhead After a Schema 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 Java Database Connectivity (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 Bottlenecks From Code That Is Not Native

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

Oracle XML DB 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

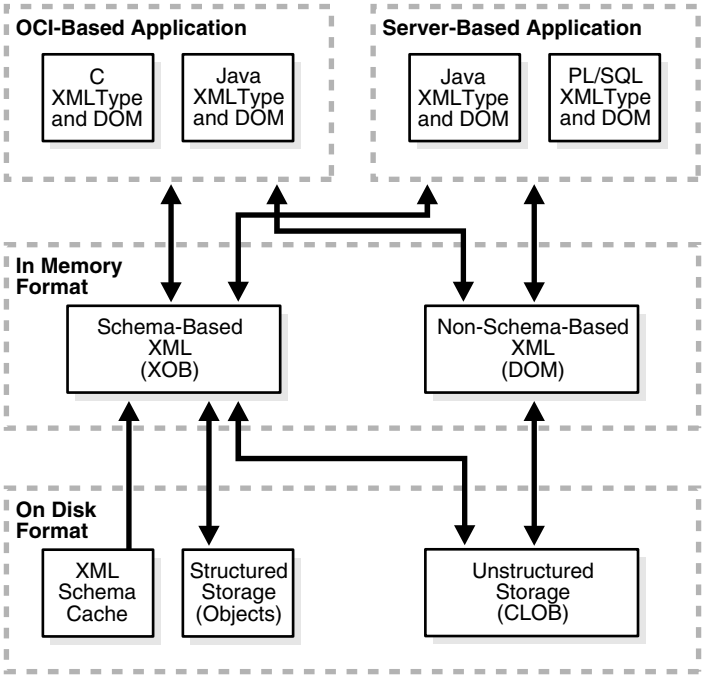
One of the biggest bottlenecks when using Java and XML is with type conversions. Internally Java uses UCS-2 to represent character data. Most XML files and databases do not contain UCS-2 encoded data. This means that all data contained in an XML file has to be converted from 8-Bit or UTF-8 encoding to 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. The format and data of XML purchase orders are well suited for Oracle XML DB storage and processing techniques because purchase orders 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 unstructured or semistructured data. This chapter also further explains Oracle XML DB concepts introduced in Chapter 1, "Introduction to 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:

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

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

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:

- `XMLType` can represent an XML document in the database, so it is accessible in SQL.
- `XMLType` has built-in methods that operate on XML content. For example, you can use `XMLType` methods to create, extract, and index XML data stored in Oracle Database.
- `XMLType` functionality is also available through a set of Application Program Interfaces (APIs) provided in PL/SQL and Java.
- `XMLType` can be used in PL/SQL stored procedures for parameters, return values, and variables

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

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

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

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

Benefits of the XMLType Datatype and API

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

- ***Versatile API.*** `XMLType` has a versatile API for application development that includes built-in functions, indexing, and navigation support.
- ***XMLType and SQL***. You can use `XMLType` in SQL statements, combined with other datatypes. For example, you can query `XMLType` columns and join the result of the extraction with a relational column. Oracle Database determines an optimal way to run such queries.
- ***Indexing:***
  - Oracle XML DB lets you create *B*Tree indexes* on the object-relational tables that 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.
  - *Function-based indexes* can be used to create indexes on explicit XPath expressions for both structured and unstructured `XMLType` storage.

When to Use XMLType

Use `XMLType` any time you want to use the database as a persistent storage of XML. `XMLType` functionality includes the following:

- *SQL queries on part of or the whole XML document* – SQL functions `existsNode` and `extract` provide the necessary SQL query functions over XML documents.
- *XPath access using SQL functions* `existsNode` *and* `extract` – `XMLType` uses the built-in C XML parser and processor and hence provides better performance and scalability when used inside the server.
- *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.
- *Indexing on XPath document queries* – `XMLType` has methods that you can use to create function-based indexes that optimize searches.
- *Separation of applications from storage models* – Using `XMLType` instead of `CLOB` values 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.
- *Support 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.

Two 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, also called document fidelity. The original XML is preserved, including whitespace. An entire XML document is stored as a whole in a LOB. 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 fit into the SQL object model. This information can be accessed through APIs in SQL or Java, using 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 need 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 1–3, " 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, whitespace within tags, and data format is lost. DOM fidelity is maintained.
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 mytable1 (key_column VARCHAR2(10) PRIMARY KEY,
                       xml_column XMLType);

Table created.
```

Example 3–2 Creating a Table of XMLType

```
CREATE TABLE mytable2 OF XMLType;

Table created.
```

Loading XML Content into Oracle XML DB

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

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

Loading XML Content Using SQL or PL/SQL

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

See Also:

- Chapter 4, "XMLType Operations"
- "APIs for XML" on page 1-5
- *Oracle Database PL/SQL Packages and Types Reference* for a description of the `XMLType` constructors

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

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

Create a SQL Directory That Points to the Needed Directory

Example 3–3 shows how to insert XML content into an `XMLType` table. Before making this insertion, 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 path_to_folder_containing_XML_file';
```

Example 3–3 Inserting XML Content into an XMLType Table

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

1 row created.
```

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

Loading XML Content Using Java

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

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

```
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://www.oracle.com/technology/sample_code/tech/xml/xmldb/content.html` demonstrates how to load a directory of XML files into Oracle XML DB using Java Database Connectivity (JDBC). JDBC is a set of Java interfaces to Oracle Database.

Loading XML Content Using C

Example 3–5 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 "stdio.h"
#include <xml.h>
#include <stdlib.h>
#include <string.h>
#include <ocixmldb.h>
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
```

```
OCIServer *srvhp;
OCIDuration dur;
OCISession *sesshp;
oratext *username = "QUINE";
oratext *password = "CURRY";
oratext *filename = "AMCEWEN-20021009123336171PDT.xml";
oratext *schemaloc = "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd";

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

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

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

sword init_oci_connect()
{
. . .
}

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

void free_oci()
{
. . .
}

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

```
  oratext buf[10000];

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

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

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

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

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

Loading Large XML Files That Contain Small XML Documents

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

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

See Also: http://www.oracle.com/technology/sample_code/tech/xml/xmldb/content.html, "SAX Loader Application" for an example of how to do this

Loading Large XML Files Using SQL*Loader

Use SQL*Loader to load large amounts of XML data into Oracle Database. SQL*Loader loads in one of two modes, conventional or direct path. Table 3–2 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 29, "Loading XML Data Using SQL*Loader"
- Example 29–1, "Loading Very Large XML Documents Into Oracle Database Using SQL*Loader" on page 29-4 for an example of direct loading of XML data.

Loading XML Documents into the Repository Using DBMS_XDB

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

Example 3–6 Inserting XML Content into the Repository Using PL/SQL DBMS_XDB

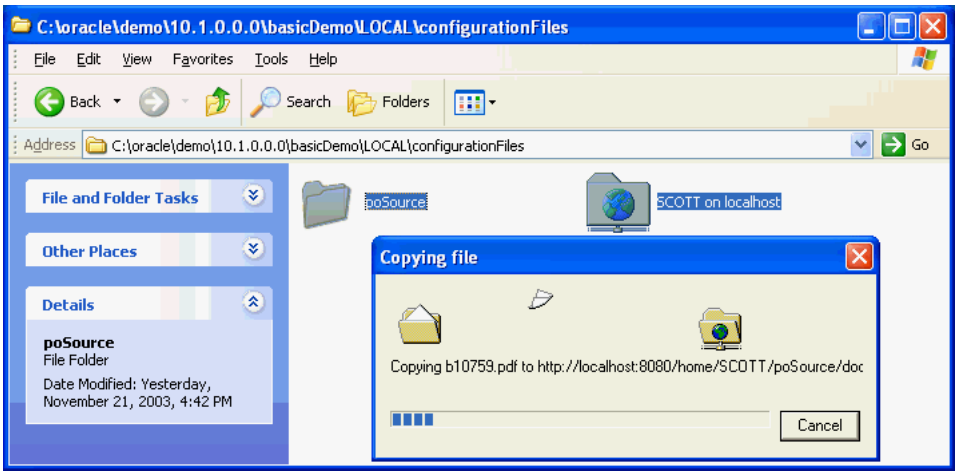
```
DECLARE
  res BOOLEAN;
BEGIN
  res := DBMS_XDB.createResource('/home/QUINE/purchaseOrder.xml',
                                 bfilename('XMLDIR', 'purchaseOrder.xml'),
                                 nls_charset_id('AL32UTF8'));
END;
/
```

Many operations for configuring and using Oracle XML DB are based on processing one or more XML documents – 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 the Repository Using Protocols

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

Figure 3–1 Using Windows Explorer to Load Content into the Repository



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

Note: Oracle XML DB Repository can also store content that is not XML data, such as HTML files, JPEG images, word documents, as well as XML documents (schema-based and non-schema-based).

Character Sets of XML Documents

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

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

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

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

XML Encoding Declaration

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

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

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

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

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

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

Character-Set Determination When Loading XML Documents into the Database

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

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

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

The following examples show different ways to specify external encoding:

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

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

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

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

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

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

Character-Set Determination When Retrieving XML Documents from the Database

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

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

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

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

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

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

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

  ```
  CREATE OR REPLACE FUNCTION getXML(pathname VARCHAR2, charset VARCHAR2)
                      RETURN BLOB IS
      xblob  BLOB;
  BEGIN
      SELECT e.RES.getBlobVal(nls_charset_id(charset)) INTO xblob
        FROM RESOURCE_VIEW e WHERE ANY_PATH = pathname;
     RETURN xblob;
  END;
  /
  ```

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

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

  > **See Also:** FTP Quote Methods on page 25-9

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

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

  > **See Also:** Controlling Character Sets for HTTP(S) on page 25-15

Overview of the W3C XML Schema Recommendation

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

XML Instance Documents

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

XML Schema for Schemas

The W3C Schema working group publishes an XML schema, often referred to as the "Schema for Schemas". This XML schema provides the definition, or vocabulary, of the XML Schema language. All valid XML schemas can be considered 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 emacs or vi
- An XML schema-aware editor, such as the XML editor included with Oracle JDeveloper
- An explicit XML schema-authoring tool, such as XMLSpy from Altova Corporation

XML Schema Features

The XML Schema language defines 47 scalar 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 Purchase-Order XML Schema, purchaseOrder.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
  <xs:element name="PurchaseOrder" type="PurchaseOrderType"/>
  <xs:complexType name="PurchaseOrderType">
    <xs:sequence>
```

```
    <xs:element name="Reference" type="ReferenceType"/>
    <xs:element name="Actions" type="ActionsType"/>
    <xs:element name="Reject" type="RejectionType" minOccurs="0"/>
    <xs:element name="Requestor" type="RequestorType"/>
    <xs:element name="User" type="UserType"/>
    <xs:element name="CostCenter" type="CostCenterType"/>
    <xs:element name="ShippingInstructions" type="ShippingInstructionsType"/>
    <xs:element name="SpecialInstructions" type="SpecialInstructionsType"/>
    <xs:element name="LineItems" type="LineItemsType"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemsType">
  <xs:sequence>
    <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemType">
  <xs:sequence>
    <xs:element name="Description" type="DescriptionType"/>
    <xs:element name="Part" type="PartType"/>
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer"/>
</xs:complexType>
<xs:complexType name="PartType">
  <xs:attribute name="Id">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:minLength value="10"/>
        <xs:maxLength value="14"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="Quantity" type="moneyType"/>
  <xs:attribute name="UnitPrice" type="quantityType"/>
</xs:complexType>
<xs:simpleType name="ReferenceType">
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="User" type="UserType"/>
          <xs:element name="Date" type="DateType" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0"/>
    <xs:element name="Date" type="DateType" minOccurs="0"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0"/>
  </xs:all>
</xs:complexType>
```

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

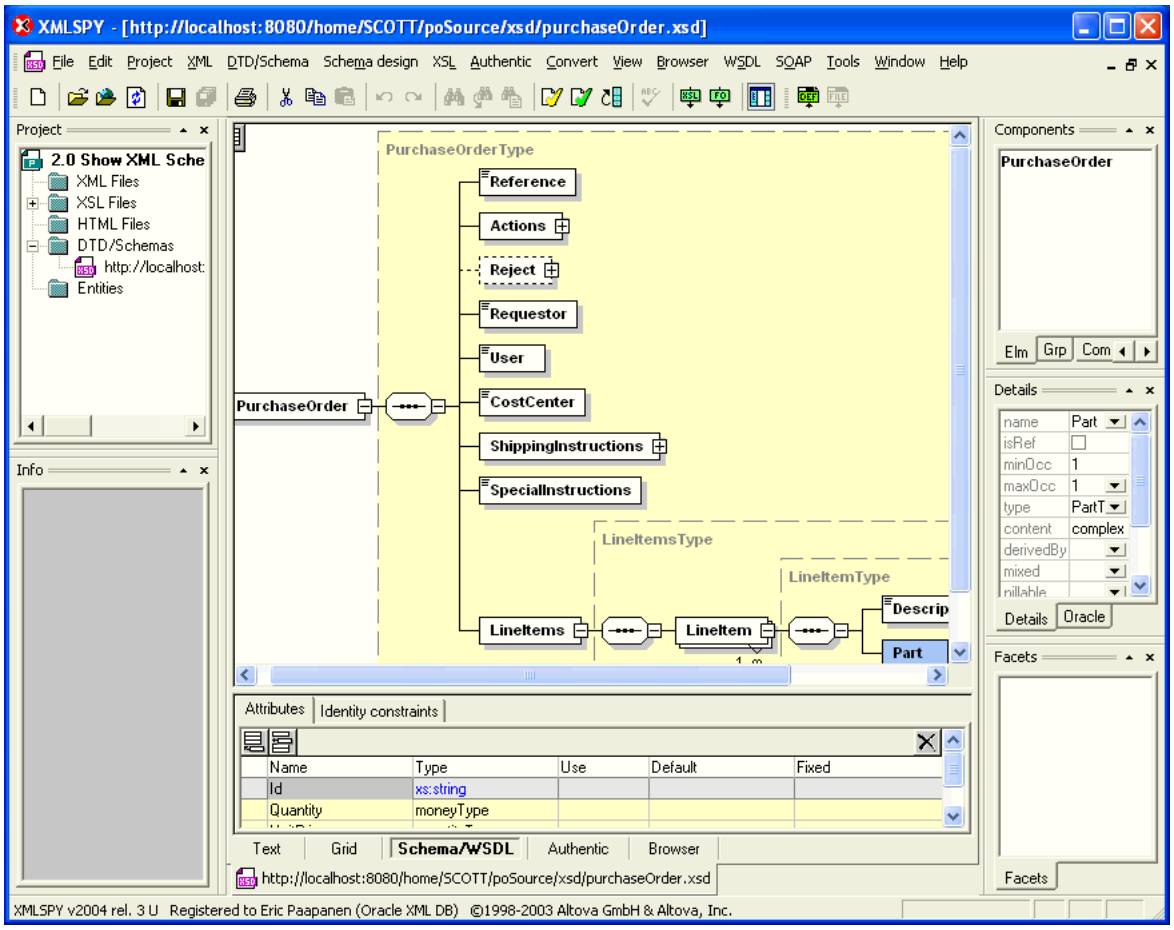
```
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="TelephoneType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="24"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DateType">
    <xs:restriction base="xs:date"/>
  </xs:simpleType>
  <xs:simpleType name="CommentsType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DescriptionType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

See Also: Appendix A, "XML Schema Primer" for a more detailed listing of XML schema `purchaseOrder.xsd`

Graphical Representation of the Purchase-Order XML Schema

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

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



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

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

Using XML Schema with Oracle XML DB

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

Why Use XML Schema With Oracle XML DB?

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

Validating Instance Documents with XML Schema

The most common usage of XML Schema is as a mechanism for validating that instance documents conform to a given XML schema. The `XMLType` 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.

Note: As always:

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

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

Controlling How XML Collections are Stored in the Database

Annotations are also used to control how collections in an XML document 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 a 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 a set of `XMLType` values. Each element in the collection is treated as a separate `XMLType` value. 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: Basic"

Collections: Default Mapping

When no annotations are supplied by the user, Oracle XML DB stores a collection as a varray 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 the following to the root element of the XML schema:

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

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

Example 3–8 shows the beginning of the `PurchaseOrder` XML schema with annotations. See Example D–1 on page D-13 for the complete schema listing.

Example 3–8 Annotated Purchase-Order XML Schema, purchaseOrder.xsd

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

```
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION" xdb:SQLCollType="ACTION_V">
      <xs:complexType xdb:SQLType="ACTION_T">
        <xs:sequence>
          <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
          <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_ACTIONED"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
  <xs:sequence>
    <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
    <xs:element name="address" type="AddressType" minOccurs="0" xdb:SQLName="SHIP_TO_ADDRESS"/>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostCenterType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="4"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="VendorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PurchaseOrderNumberType">
  <xs:restriction base="xs:integer"/>
```

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

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

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

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

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

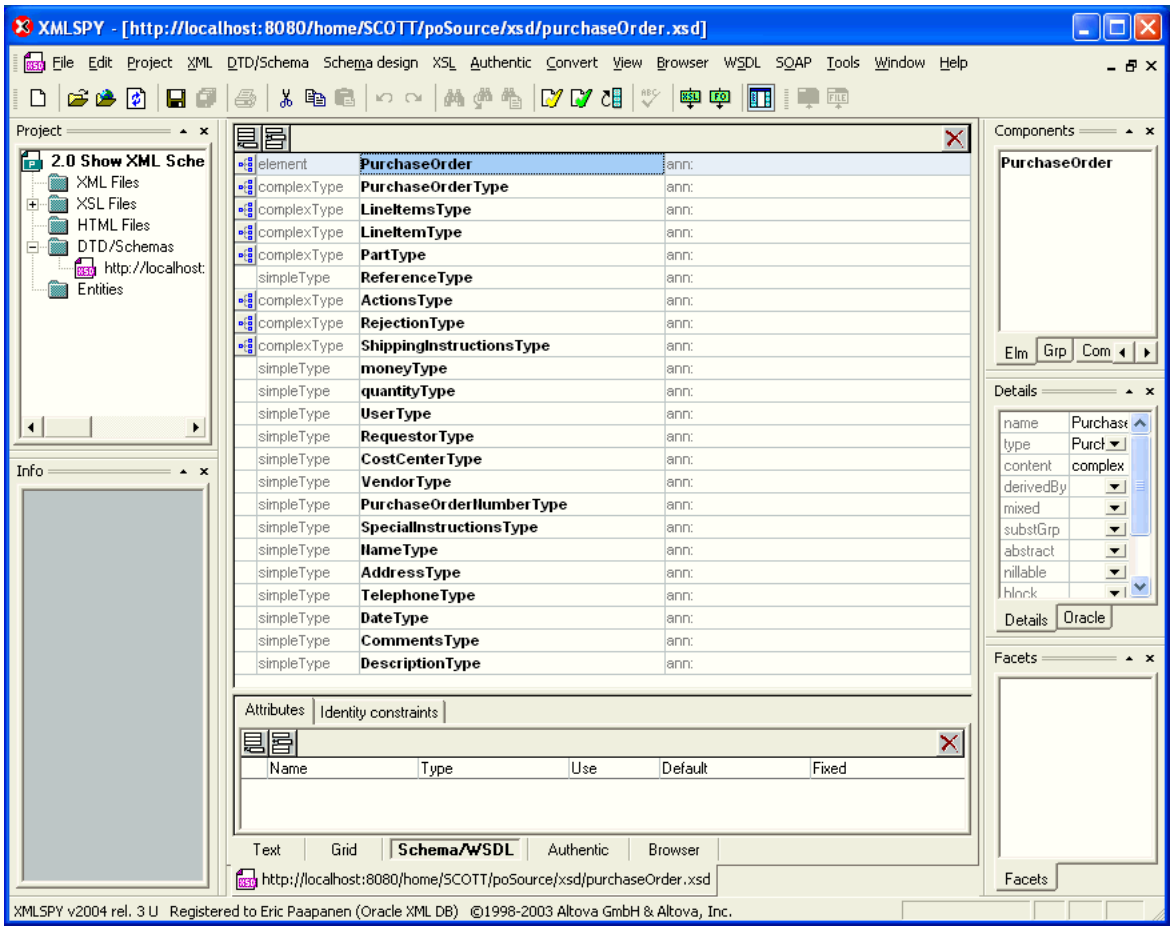
  The first occurrence of `SQLType` specifies that the name of the SQL type generated from complexType `PurchaseOrderType` is `purchaseorder_t`.

  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`.

- `SQLName` annotation. This provides an explicit name for each SQL attribute of `purchaseorder_t`.

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

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



Registering an XML Schema with Oracle XML DB

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

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

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

You register an XML schema with PL/SQL procedure `DBMS_XMLSCHEMA.registerschema`. See Example 3–9. By default, when an XML schema is registered, Oracle XML DB automatically generates all of the SQL object types and `XMLType` tables required to manage the instance documents.

XML schemas can be registered as global or local.

See Also:

- "Schema Registration Considerations" on page 5-7 for considerations to keep in mind when you register an XML schema
- Chapter 5, "XML Schema Storage and Query: Basic" for a discussion of the differences between global and local schemas
- *Oracle Database PL/SQL Packages and Types Reference* for information on `DBMS_XMLSCHEMA.registerschema`

Example 3–9 Registering an XML Schema with DBMS_XMLSCHEMA.registerSchema

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    'http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd',
    XDBURIType('/source/schemas/poSource/xsd/purchaseOrder.xsd').getClob(),
    TRUE,
    TRUE,
    FALSE,
    TRUE);
END;
/
```

In this example, the unique identifier for the XML schema is:

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

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

During XML schema registration, an `XDBURIType` accesses the content of the XML schema document, based on its location in the repository. Flags passed to procedure `registerSchema` 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.

Procedure `DBMS_XMLSCHEMA.registerSchema` performs the following operations:

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

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

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

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

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

Working with Large XML Schemas

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.

As the total number of elements and attributes defined by a `complexType` reaches 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:

- Use 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.
- Use 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 subelements in the XML document to be stored as rows in a separate `XMLType` table. Oracle XML DB maintains the relationship between the two tables using a `REF` of `XMLType`. Good candidates for this approach are XML schemas that 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 the type of queries and updates to be performed against the data.

Working with Global Elements

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

You can use the `xdb:defaultTable` attribute to specify the name of the default table for a given global element. Each `xdb:defaultTable` attribute value you provide must be *unique* among *all schemas* registered by a given database user. If you do *not* supply a nonempty default table name for some element, then a unique name is provided automatically.

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

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

- Add the annotation `xdb:defaultTable=""` to the definition of *each* global element that will *not* appear as the root element of an XML instance document. Using this approach, you allow automatic default-table creation, in general, and you prohibit it explicitly where needed, using `xdb:defaultTable=""`.

- Set 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. Using this approach, you inhibit automatic default-table creation, and you create only the tables that are needed, by hand.

Creating XML Schema-Based XMLType Columns and Tables

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

Example 3–11 shows how to manually create the `PurchaseOrder` table, the default table for `PurchaseOrder` elements.

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

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

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`. Because of the column specification `ORGANIZATION INDEX OVERFLOW`, the nested tables are *index-organized*. Because of the `PRIMARY KEY` specification, they automatically contain pseudocolumn `NESTED_TABLE_ID` and column `SYS_NC_ARRAY_INDEX$`, which are required to link them back to the parent column.

This `CREATE TABLE` statement is equivalent to the `CREATE TABLE` statement automatically generated by Oracle XML DB when the schema annotation `storeVarrayAsTable="true"` is included in the root element of the `PurchaseOrder` XML schema (and `genTables="true"` is set during schema registration). When this annotation is used, the nested tables generated by the XML schema registration process are given system-generated names, which can be difficult to work with. You can give them more meaningful names using the SQL statement `RENAME TABLE`.

Note: Annotation `storeVarrayAsTable="true"` causes element collections to be persisted as rows in an index-organized table (IOT). Oracle Text does not support IOTs. Do *not* use this annotation if you will need to use Oracle Text indexes for text-based `ora:contains` searches over a collection of elements. See "ora:contains Searches Over a Collection of Elements" on page 10-23. To provide for searching with Oracle Text indexes:

1. Set `genTables="false"` during schema registration.
2. Create the necessary tables *manually*, without using the clause `ORGANIZATION INDEX OVERFLOW`, so the tables will be *heap-organized* instead of index-organized (IOT).

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

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

```
DESCRIBE purchaseorder
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------

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

The output of the `DESCRIBE` statement 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. 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 XML Schema Instance Documents

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

- Explicitly identify the XML schema when creating the `XMLType`. This can be done by passing the name of the XML schema to the `XMLType` constructor, or by invoking 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. 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.

Attributes noNamespaceSchemaLocation and schemaLocation

If the target XML schema does not declare a target namespace, the noNamespaceSchemaLocation attribute is used to identify the XML schema. The value of the attribute is the *schema location hint*. This is the unique identifier passed to PL/SQL procedure DBMS_XMLSCHEMA.registerSchema when the 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/source/schemas/poSource/xsd/purchaseOrder.xsd">
```

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

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

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

```
<xs:schema targetNamespace="http://demo.oracle.com/xdb/purchaseOrder"
```

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

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

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

Dealing with Multiple Namespaces

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

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` raises an error. Example 3–13 illustrates this.

Example 3–13 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-30937: No schema definition for 'Invoice' (namespace '') in parent '#document'
```

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

- The name of the XML document root element must match the name of global element used to define the `XMLType` table or column.

- The XML document must include the appropriate attributes from the `XMLSchema-instance` namespace, or the XML document must be explicitly associated with the XML schema using the `XMLType` constructor or 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 only within individual XML documents. Database (SQL) constraints are enforced across sets of XML documents.

Comparing Partial to Full XML Schema Validation

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

Partial Validation

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. It ensures only that all the mandatory elements and attributes are present and that there are no unexpected elements or attributes in the document. That is, it ensures only that the structure of the XML document conforms to the SQL type definitions that were derived from the XML schema. Because complete schema validation is very costly, Oracle XML DB does *not* try to ensure that the instance document is fully compliant with the XML schema. Example 3–14 provides an example of failing 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, so it 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 Processing Time and Memory Usage Full XML schema validation costs processing time and memory. By leaving the decision of 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 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 (QUINE.VALIDATE_PURCHASEORDER) violated
```

The pseudocolumn name OBJECT_VALUE can be used to access the content of an XMLType table from within a trigger.

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

This example 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.

```
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')));
          *
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 "QUINE.VALIDATE_PURCHASEORDER", line 3
ORA-04088: error during execution of trigger 'QUINE.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 ability to ensure that the value of an element or attribute:

- Is unique across a set of XML documents (a `UNIQUE` constraint)
- Exists in a particular data source outside the current document (`FOREIGN KEY` constraint)

The mechanisms used to enforce integrity on XML are the same mechanisms used to enforce integrity on conventional relational data. 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 use the database to enforce 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

```
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 (QUINE.REFERENCE_IS_UNIQUE) violated

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

INSERT INTO purchaseorder
*

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

The uniqueness constraint `reference_is_unique` ensures 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` ensures 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 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 violates the constraint `reference_is_unique`.

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

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

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

This example shows that database integrity is also enforced when a protocol, such as FTP, is used to upload XML schema-based XML content into Oracle XML DB Repository.

```
$ ftp localhost 2100
Connected to localhost.
220 mdrake-sun FTP Server (Oracle XML DB/Oracle Database 10g Enterprise Edition
Release 10.1.0.0.0 - Beta) ready.
Name (localhost:oracle10): QUINE
331 pass required for QUINE
```

```
Password:
230 QUINE logged in
ftp> cd /source/schemas
250 CWD Command successful
ftp> put InvalidReference.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-31154: invalid XML document
ORA-19202: Error occurred in XML processing
LSX-00221: "SBELL-20021009" is too short (minimum length is 18)
ORA-06512: at "SYS.XMLTYPE", line 333
ORA-06512: at "QUINE.VALIDATE_PURCHASEORDER", line 3
ORA-04088: error during execution of trigger 'QUINE.VALIDATE_PURCHASEORDER'
550 End Error Response
ftp> put InvalidElement.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-30937: No schema definition for 'UserName' (namespace '##local') in parent
'PurchaseOrder'
550 End Error Response
ftp> put DuplicateReference.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-00001: unique constraint (QUINE.REFERENCE_IS_UNIQUE) violated
550 End Error Response
ftp> put InvalidUser.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-02291: integrity constraint (QUINE.USER_IS_VALID) violated - parent key not
 found
550 End Error Response
```

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.

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

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

Querying XML Content Stored in Oracle XML DB

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

PurchaseOrder XML Document

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

Example 3–19 PurchaseOrder XML Instance Document

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

```
  xsi:noNamespaceSchemaLocation=
    "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd">
  <Reference>SBELL-2002100912333601PDT</Reference>
  <Actions>
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Sarah J. Bell</Requestor>
  <User>SBELL</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Sarah J. Bell</name>
    <address>400 Oracle Parkway
      Redwood Shores
      CA
      94065
      USA</address>
    <telephone>650 506 7400</telephone>
  </ShippingInstructions>
  <SpecialInstructions>Air Mail</SpecialInstructions>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="2">
      <Description>The Unbearable Lightness Of Being</Description>
      <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>Sisters</Description>
      <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
    </LineItem>
  </LineItems>
</PurchaseOrder>
```

Retrieving the Content of an XML Document Using Pseudocolumn OBJECT_VALUE

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

Example 3–20 Using OBJECT_VALUE to Retrieve an Entire XML Document

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

```
SET LONG 10000
SET PAGESIZE 100

SELECT OBJECT_VALUE FROM purchaseorder;

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

```
/poSource/xsd/purchaseOrder.xsd">
  <Reference>SBELL-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

SQL function `extract` returns the nodes that match an XPath expression. Nodes are returned as an instance of `XMLType`. The result of `extract` can be either a complete document or an XML fragment. The functionality of SQL function `extract` is also available through `XMLType` method `extract()`.

Example 3–21 Accessing XML Fragments Using EXTRACT

This query returns an `XMLType` value containing the `Reference` element that matches the XPath expression.

```
SELECT extract(OBJECT_VALUE, '/PurchaseOrder/Reference')
  FROM purchaseorder;

EXTRACT(OBJECT_VALUE, '/PURCHASEORDER/REFERENCE')
--------------------------------------------------
<Reference>SBELL-20021009123336O1PDT</Reference>

1 row selected.
```

This query returns an XMLType value 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 query returns an XMLType value containing the three Description elements that match the XPath expression. These elements are returned as nodes in a single XMLType, so the XMLType value does not have a single root node. It is treated as an XML *fragment*.

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

Example 3–22 Accessing a Text Node Value Using EXTRACTVALUE

This query returns the value of the text node associated with the Reference element that matches the XPath expression. The value is returned as a VARCHAR2 value:

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference')
FROM purchaseorder;

EXTRACTVALUE(OBJECT_VALUE, '/PURCHASEORDER/REFERENCE)
--------------------------------------------------------------------------
SBELL-20021000912333601PDT

1 row selected.
```

The followingquery returns the value of the text node associated with the Description element associated with the first LineItem element. The value is returned as a VARCHAR2 value. The *first* LineItem element is indicated by the index [1].

```
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 query returns the value of the text node associated with a `Description` element contained in a `LineItem` element. The particular `LineItem` element is specified by an `Id` attribute value. The value returned is of type `VARCHAR2`. The predicate that identifies which `LineItem` element 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 SQL function `extractValue`. In the first example, the XPath expression matches *three* nodes in the document (to be valid, it must match only *one*). In the second example, the XPath expression identifies a *parent* node, not a leaf node (text node or attribute value).

Example 3–23 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 3:
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()')
 *
ERROR at line 1:
ORA-01427: single-row subquery returns more than one row
```

Depending on whether or not XPath rewrite takes place, the last two queries 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 SQL function `existsNode` evaluates whether or not a given document contains a node that matches a W3C XPath expression. Function `existsNode` 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, and whether or not a node with the specified value exists in the document. The functionality provided by SQL function `existsNode` is also available through `XMLType` method `existsNode`.

Example 3–24 Searching XML Content Using EXISTSNODE

This query uses SQL function `existsNode` to check if the XML document contains an element named `Reference` that is a child of the root element `PurchaseOrder`:

```
SELECT COUNT(*)
  FROM purchaseorder
WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder/Reference') = 1;

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


1 row selected.
```

This query 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.
```

This query 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.
```

This query checks if the XML document contains a root element `PurchaseOrder` that contains a `LineItems` element that contains a `LineItem` element that contains a `Part` element with an `Id` attribute:

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

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

1 row selected.
```

This query checks if the XML document contains a root element `PurchaseOrder` that contains a `LineItems` element that contains a `LineItem` element that contains a `Part` element with `Id` attribute value `715515009058`:

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

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

This query checks if the XML document contains a root element `PurchaseOrder` that contains a `LineItems` element whose third `LineItem` element contains a `Part` element with `Id` attribute value `715515009058`:

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

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

This query uses SQL function `extractValue` to limit the results of the `SELECT` statement to 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.
```

This query uses `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 a WHERE Clause

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

You can use SQL function `existsNode` to restrict the result set to documents containing nodes that match an XPath expression. You can use SQL function `extractValue` when joining across multiple tables based on the value of one or more nodes in the XML document. Also, you can use use `extractValue` whenever specifying a condition is easier with SQL (for example, using keyword `LIKE` for pattern matching) than with XPath.

Example 3–25 Limiting the Results of a SELECT Using EXISTSNODE in a WHERE Clause

This query shows how to use SQL function `existsNode` to limit the results of the `SELECT` statement to rows where the text node associated of 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 3–26 Finding the Reference for any PurchaseOrder Using extractValue and existsNode

This example uses SQL functions `extractValue` and `existsNode` to find the `Reference` element for any `PurchaseOrder` element whose first `LineItem` element contains an order for the item with `Id` `715515009058`. Function `existsNode` is used in the `WHERE` clause to determine which rows are selected, and `extractValue` is used in the `SELECT` list to control which part of the selected documents appears 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 3–21 demonstrates how the SQL function `extract` returns an `XMLType` value 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 an XML *fragment* containing all of the matching nodes. A fragment differs from a document in that it has no single root element.

This kind of result is common when `extract` 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 can be of different types).

SQL function `XMLSequence` can perform SQL operations on an `XMLType` value that contains a fragment. It generates a collection of `XMLType` objects from an `XMLType` containing a fragment. The collection contains one `XMLType` value for each of the top-level 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 call that returns multiple nodes.

Example 3–27 Using XMLSEQUENCE and TABLE to View Description Nodes

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

An initial attempt uses SQL function `extractValue`. It fails, because there is more than 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-20021009123336O1PDT"]')
        = 1;
SELECT extractValue(p.OBJECT_VALUE,
                    '/PurchaseOrder/LineItems/LineItem/Description')
 *
ERROR at line 1:
ORA-01427: single-row subquery returns more than one row
```

A second attempt uses SQL function `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-20021009123336O1PDT"]')
        = 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 SQL functions `XMLSequence` and `table`. 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(des)
  FROM purchaseorder p,
       table(XMLSequence(
               extract(p.OBJECT_VALUE,
                       '/PurchaseOrder/LineItems/LineItem/Description'))) des
  WHERE existsNode(p.OBJECT_VALUE,
                   '/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]')
        = 1;

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

3 rows selected.
```

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

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

EXTRACTVALUE(VALUE(DES),'/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 SQL function table and the row operated on by the SQL function extract. The table that provides input to extract must appear *before* the table expression in the FROM list. The correlated join ensures a one-to-many (1:N) relationship between the rows generated by the SQL function table and the row containing the value that is processed by extract.

Example 3–28 Counting the Number of Elements in a Collection Using XMLSEQUENCE

This example demonstrates how to use SQL function XMLSequence to count the number of elements in a collection. It also shows how SQL keywords such as ORDER BY and GROUP BY can be applied to results returned by the SQL function extractValue.

In this case, the query first locates the set of XML documents that match the XPath argument to SQL function existsNode. It then generates 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 LineItem elements belong to which PurchaseOrder element.

```
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-2002100912333601PDT               3
SBELL-20021009123336231PDT             25
```

```
SBELL-20021009123336331PDT             10
SBELL-20021009123336362PDT             15
SBELL-20021009123336532PDT             14
SBELL-20021009123337353PDT             10
SBELL-2002100912333763PDT              21
SBELL-20021009123337673PDT             10
SBELL-20021009123338204PDT             14
SBELL-20021009123338304PDT             24
SBELL-20021009123338505PDT             20

13 rows selected.
```

Example 3–29 Counting the Number of Child Elements in an Element Using XMLSEQUENCE

The following example demonstrates how to use SQL function `XMLSequence` to count the number of *child* elements of a given element. The XPath expression passed to the SQL function `extract` contains a wildcard (`*`) that matches the elements that are direct descendants of a `PurchaseOrder` element. The `XMLType` value returned by `extract` contains the set of nodes that match the XPath expression. Function `XMLSequence` transforms each top-level element in the fragment into a separate `XMLType` object, and the SQL function `table` 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-20021009123336601PDT"]')
        = 1;

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

1 row selected.
```

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 relational views can use XPath expressions and SQL functions such as `extractValue` to define a 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` instead of `CLOB`; that is, when they are stored using object-relational storage techniques.

Example 3–30 Creating Relational Views On XML Content

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

```
CREATE OR REPLACE VIEW
  purchaseorder_master_view(reference, requestor, userid, costcenter,
                            ship_to_name, ship_to_address, ship_to_phone,
                            instructions)
AS SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference'),
```

```
          extractValue(OBJECT_VALUE, '/PurchaseOrder/Requestor'),
          extractValue(OBJECT_VALUE, '/PurchaseOrder/User'),
          extractValue(OBJECT_VALUE, '/PurchaseOrder/CostCenter'),
          extractValue(OBJECT_VALUE, '/PurchaseOrder/ShippingInstructions/name'),
          extractValue(OBJECT_VALUE,
                       '/PurchaseOrder/ShippingInstructions/address'),
          extractValue(OBJECT_VALUE,
                       '/PurchaseOrder/ShippingInstructions/telephone'),
          extractValue(OBJECT_VALUE, '/PurchaseOrder/SpecialInstructions')
     FROM purchaseorder;

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 creates view `purchaseorder_master_view`. There is one row in the view for each row in table `purchaseorder`.

The `CREATE OR REPLACE VIEW` statement defines the set of columns that make up the view. The `SELECT` statement uses XPath expressions and function `extractValue` to map the nodes in the XML document to the columns defined by the view. This technique can be used when there is a one-to-one (1:1) relationship between documents in the `XMLType` table and the rows in the view.

Example 3–31 Using a View to Access Individual Members of a Collection

This example shows how to use SQL functions `extract` and `XMLSequence` for a one-to-many (1:N) 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 of rows.

```
CREATE OR REPLACE VIEW
  purchaseorder_detail_view(reference, itemno, description,
                            partno, quantity, unitprice)
AS SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference'),
          extractValue(value(ll), '/LineItem/@ItemNumber'),
          extractValue(value(ll), '/LineItem/Description'),
          extractValue(value(ll), '/LineItem/Part/@Id'),
          extractValue(value(ll), '/LineItem/Part/@Quantity'),
          extractValue(value(ll), '/LineItem/Part/@UnitPrice')
     FROM purchaseorder p,
          table(XMLSequence(extract(OBJECT_VALUE,
                                    '/PurchaseOrder/LineItems/LineItem'))) ll;

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 that occurs in the XML documents stored in table purchaseorder.

The CREATE OR REPLACE VIEW statement defines the set of columns that make up the view. The SELECT statement uses extract to access the set of LineItem elements in each PurchaseOrder document. It then uses SQL functions 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 SQL function extractValue are used to map the nodes in the LineItem documents to 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 one-to-many (1:N) 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 OR REPLACE 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 3–32 SQL queries on XML Content Using Views

This example uses a simple query against the master view. A conventional SELECT statement selects rows where the userid column starts with S.

```
SELECT reference, costcenter, ship_to_name
  FROM purchaseorder_master_view
  WHERE userid LIKE 'S%';

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

The following query is based on a join between the master view and the detail view. A conventional SELECT statement finds the purchaseorder_detail_view rows where the value of the itemno column is 1 and the corresponding purchaseorder_master_view row contains a userid column with the value SBELL.

```
SELECT d.reference, d.itemno, d.partno, d.description
  FROM purchaseorder_detail_view d, purchaseorder_master_view m
  WHERE m.reference = d.reference
    AND m.userid = 'SBELL'
```

```
    AND d.itemno = 1;

REFERENCE                          ITEMNO PARTNO         DESCRIPTION
------------------------------ ---------- -------------------------------------------
SBELL-20021009123336231PDT              1 37429165829    Juliet of the Spirits
SBELL-20021009123336331PDT              1 715515009225   Salo
SBELL-20021009123337353PDT              1 37429141625    The Third Man
SBELL-20021009123338304PDT              1 715515009829   Nanook of the North
SBELL-20021009123338505PDT              1 37429122228    The 400 Blows
SBELL-20021009123335771PDT              1 37429139028    And the Ship Sails on
SBELL-20021009123335280PDT              1 715515011426   All That Heaven Allows
SBELL-20021009012333763PDT              1 715515010320   Life of Brian - Python
SBELL-20021009012333601PDT              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.
```

Because 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 or the generated result set.

By exposing XML content as relational data, Oracle XML DB allows advanced database features, 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 3–33 Querying XML Using Views of XML Content

This example demonstrates how to use relational views over XML content to perform business-intelligence queries on XML documents. The query selects `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 ordered in each `PurchaseOrder` document. For part number `715515009126`, there are four `PurchaseOrder` documents where one copy of the item is ordered and seven `PurchaseOrder` documents where three copies of the item are ordered.

See Also:

- Chapter 4, "XMLType Operations" for a description of `XMLType` datatype and functions
- Appendix B, "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 of a document 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.

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

Example 3–34 Updating XML Content Using UPDATEXML

This example uses SQL function `updateXML` to update the text node associated with the `User` element.

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/User')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]')
        = 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-2002100912333601PDT"]')
      = 1;

1 row updated.

SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/User')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]')
        = 1;

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

1 row selected.
```

Example 3–35 Replacing an Entire Element Using UPDATEXML

This example uses SQL function `updateXML` to replace an entire element within the XML document. 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-2002100912333601PDT"]')
        = 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 3–36 Incorrectly Updating a Node That Occurs Multiple Times In a Collection

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

```
SELECT extractValue(value(li), '/Description')
```

```
  FROM purchaseorder p,
       table(XMLSequence(
               extract(p.OBJECT_VALUE,
                       '/PurchaseOrder/LineItems/LineItem/Description'))) li
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]')
        = 1;

EXTRACTVALUE(VALUE(LI),'/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(li), '/Description')
  FROM purchaseorder p,
       table(XMLSequence(
               extract(p.OBJECT_VALUE,
                       '/PurchaseOrder/LineItems/LineItem/Description'))) li
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]')
        = 1;

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

3 rows selected.
```

Instead of updating the required node, SQL function `updateXML` updates the values of all text nodes that belong to the `Description` element. This is the correct behavior, but it is not what was intended. *The `WHERE` clause can only be used to identify which **documents** must be updated, not which nodes within the document must be updated.*

After the document has been selected, the *XPath expression* passed to `updateXML` determines which *nodes* within the document must be updated. In this case, the XPath expression identified all three `Description` nodes, so all three of the associated text nodes were updated. See Example 3–37 for the correct way to update the nodes.

Example 3–37 Correctly Updating a Node That Occurs Multiple Times In a Collection

To correctly use SQL function `updateXML` to update a node that occurs multiple times within a collection, use the *XPath expression* passed to `updateXML` to identify which nodes in the XML document to update. By introducing the appropriate predicate into the XPath expression, you can limit which nodes in the document are updated. This example shows the correct way of updating one node within a collection:

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

EXTRACTVALUE(OBJECT_VALUE,'/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-2002100912333601PDT"]')
       = 1;

1 row updated.

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

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

3 rows selected.
```

Example 3–38 Changing Text Node Values Using UPDATEXML

SQL function `updateXML` allows multiple changes to be made to the document in one statement. This example shows how to change the values of text nodes belonging to the `User` and `SpecialInstructions` elements in one statement.

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/CostCenter') "Cost Center",
       extractValue(OBJECT_VALUE,
                    '/PurchaseOrder/SpecialInstructions') "Instructions"
```

```
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;

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

1 row selected.
```

This single `UPDATE` SQL statement 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-2002100912333601PDT"]') = 1;

1 row updated.

SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/CostCenter') "Cost Center",
       extractValue(OBJECT_VALUE,
                    '/PurchaseOrder/SpecialInstructions') "Instructions"
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 1;

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

1 row selected.
```

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

The way SQL functions like `updateXML` modify an XML document is determined mainly by whether or not the XML document is based on an XML schema, and how the XML document is stored:

- **XML documents stored in CLOB values.** When a SQL function like `updateXML` modifies an XML document stored as a `CLOB` (whether schema-based or not), Oracle XML DB performs the update by creating a Document Object Model (DOM) from the XML document and using DOM API methods to modify the appropriate XML data. After modification, the updated DOM is returned back to the underlying `CLOB` object.
- **XML documents stored object-relationally.** When a SQL function like `updateXML` modifies a *schema-based* XML document that is stored object-relationally, Oracle XML DB can use XPath rewrite to modify the underlying object in place. This is a *partial* update. Partial updates translate the XPath argument to the SQL function into an equivalent SQL operation. The SQL operation then directly modifies the attributes of underlying objects. Such a partial update can be much quicker than a DOM-based update. This can make a significant difference when executing a SQL statement that applies a SQL function like `updateXML` to a large number of documents.

See Also: Chapter 6, "XPath Rewrite"

Namespace Support in Oracle XML DB

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

The `namespace` parameter is required whenever the provided XPath expression contains namespace prefixes. When 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 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 SQL functions such as `extract`, `extractValue`, and `existsNode`—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` object. It uses methods provided by the DOM API to perform the required operation on the DOM. If the operation involves updating the DOM tree, then the entire XML document has to be written back to 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 *does 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**. See Chapter 6, "XPath Rewrite".

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. This insulates the database optimizer from having to understand the XPath notation and the XML data model. The database optimizer processes the rewritten 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 the conditions under which XPath rewrite takes place is key to developing Oracle XML DB applications that deliver satisfactory levels of scalability and performance.

See Also: Chapter 6, "XPath Rewrite"

Using EXPLAIN PLAN to Tune XPath Rewrites

XPath rewrite on its own cannot guarantee scalable and performant applications. The performance of SQL statements generated by XPath rewrite is ultimately determined by the available indexes and the way data is stored on disk. 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.

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 of the tools that DBAs typically use with SQL-based applications can be applied to XML-based applications using Oracle XML DB functions.

Example 3–39 Using EXPLAIN PLAN to Analyze the Selection of PurchaseOrders

This example shows how to use an EXPLAIN PLAN to look at the execution plan for selecting the set of PurchaseOrders created by user SBELL.

```
EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference') "Reference"
    FROM purchaseorder
```

```
    WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[User="SBELL"]') = 1;

Explained.

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 841749721

-------------------------------------------------------------------------------------
| Id  | Operation         | Name          | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |    1 |    24 |     5   (0)| 00:00:01|
|*  1 |  TABLE ACCESS FULL| PURCHASEORDER |    1 |    24 |     5   (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.
- *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 SQL function `extractValue`, although it is also possible to create indexes based on other 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 3–40 Creating an Index on a Text Node

This example shows creation of index `purchaseorder_user_index` on the value of the `User` element text node.

```
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 EXPLAIN PLAN is generated when the same query used in Example 3–39 is executed after the index has been created. It shows that the query plan will make use of the newly created index. The new execution plan is much more scalable—compare the EXPLAIN PLAN of Example 3–39.

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

Explained.

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
Plan hash value: 713050960

------------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                      | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                           |     1 |    24 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER             |     1 |    24 |     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')

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. The optimizer automatically uses the index whenever it is 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 provide 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 the following ways to manage the members of a collection:

- When a collection is stored as a CLOB value, you *cannot* directly access its members.
- When a varray is stored as a LOB, you *cannot* directly access members of the collection.

  Storing the members as XML text in a CLOB value means that any operation on the collection requires 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.

- When a varray is stored as a nested table, you can directly access members of the collection.
- When a varray is stored as an XMLType value, you can directly access members of the collection.

In the latter two cases (nested table and XMLType), each member of the varray becomes a row in a table, so you can access it directly though SQL.

Using Indexes to Tune Queries on Collections Stored as Nested Tables

Example 3–41 shows the execution plan for a query to find the Reference element from any document that contains an order for part number 717951002372 (Part element with an Id attribute of value 717951002372).

Example 3–41 EXPLAIN PLAN For a Selection of LineItem Elements

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.

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 47905112

-------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                         |    22 | 14300 |    10  (10)| 00:00:01 |
|   1 |  NESTED LOOPS                 |                         |    22 | 14300 |    10  (10)| 00:00:01 |
|   2 |   SORT UNIQUE                 |                         |    22 |  2640 |     8   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN          | LINEITEM_TABLE_DATA     |    22 |  2640 |     8   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | LINEITEM_PART_INDEX     |     9 |       |     2   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| PURCHASEORDER           |     1 |   530 |     1   (0)| 00:00:01 |
|*  6 |    INDEX UNIQUE SCAN          | LINEITEM_TABLE_MEMBERS |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------

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

   3 - access("SYS_NC00011$"='717951002372')
   4 - access("SYS_NC00011$"='717951002372')
   6 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")

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 existsNode, the parent row is located using the value of pseudocolumn NESTED_TABLE_ID. 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 might be acceptable if there are only a few hundred documents in the purchaseorder table, but it would be unacceptable if there are thousands or millions of documents in the table.

To improve the performance of this query, create an index that allows direct access to pseudocolumn NESTED_TABLE_ID, given the value of the Id attribute. Unfortunately,

Oracle XML DB does not 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 pseudocolumn `NESTED_TABLE_ID` that will allow direct access to the `purchaseorder` documents that contain `LineItem` elements that reference the required part.

Example 3–42 Creating an Index for Direct Access to a Nested Table

This example uses object-relational SQL to create the required index:

```
CREATE INDEX lineitem_part_index
  ON lineitem_table l(l.part.part_number, l.NESTED_TABLE_ID);

Index created.

EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference') "Reference"
    FROM purchaseorder
    WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder/LineItems/LineItem/Part[@Id="717951002372"]') = 1;

Explained.

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------
Plan hash value: 497281434

-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                         |    22 |  1012 |     4  (25)| 00:00:01 |
|   1 |  NESTED LOOPS                 |                         |    22 |  1012 |     4  (25)| 00:00:01 |
|   2 |   SORT UNIQUE                 |                         |    22 |   418 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           | LINEITEM_PART_INDEX     |    22 |   418 |     2   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| PURCHASEORDER           |     1 |    27 |     1   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN          | LINEITEM_TABLE_MEMBERS |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

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

   3 - access("SYS_NC00011$"='717951002372')
   5 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")

18 rows selected.
```

The `EXPLAIN PLAN` output shows that the same query as Example 3–42 will now make use of the newly created index. The query is resolved by using index `lineitem_part_index` to determine which documents in the `purchaseorder` table satisfy the condition in the XPath expression argument to function `existsNode`. This query is much more scalable with the indexes.

The query syntax has not changed. XPath rewrite has allowed the optimizer to analyze the query and this analysis determines that the new indexes `purchaseorder_user_index` and `lineitem_part_index` provide a more efficient way to resolve the queries.

EXPLAIN PLAN 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 PL/SQL procedure DBMS_XMLSCHEMA.register_schema contains a filter similar to the following:

```
3 - filter(SYS_CHECKACL("ACLOID","OWNERID",xmltype(''<privilege
              xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
              http://xmlns.oracle.com/xdb/acl.xsd
              DAV:http://xmlns.oracle.com/xdb/dav.xsd">
                        <read-properties/><read-contents/></privilege>''))=1)
```

This shows that ACL-based security is implemented for this table. In this example, the filter checks that the user performing the SQL query has read-contents privilege on each of the documents to be accessed.

Oracle XML DB Repository uses an ACL-based security mechanism that allows control of access to XML content document by document, rather than table by table. When XML content is accessed using a SQL statement, the SYS_CHECKACL predicate is automatically added to the WHERE clause to ensure that the security defined is enforced at the SQL level.

Enforcing ACL-based security adds overhead to the SQL query. If ACL-based security is *not* required, use procedure disable_hierarchy in package DBMS_XDBZ to turn *off* ACL checking. After calling this procedure, the SYS_CHECKACL filter no longer appears in the output generated by EXPLAIN PLAN.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information on procedure DBMS_XDBZ.disable_hierarchy

Example 3–43 EXPLAIN PLAN Generated When XPath Rewrite Does Not Occur

This example shows the kind of EXPLAIN PLAN output generated when Oracle XML DB cannot perform XPath rewrite. Function existsNode appears in the EXPLAIN output (line 3), indicating that the query will not be rewritten.

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("NESTED_TABLE_ID"=:B1)
   2 - access("NESTED_TABLE_ID"=:B1)
   3 - filter(EXISTSNODE(SYS_MAKEXML('C0A5497E8DCF110BE034080020E5CF39',
                                     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 result set based on any other conditions specified in the query WHERE clause. It then filters the rows in this potential result set to determine which rows belong in the actual result set. The filtering is performed by *constructing a DOM on each document* and performing a *functional evaluation* (using the methods defined by the DOM API) to determine whether or not each document is a member of the actual result set.

Performance can be poor when there are many documents in the potential result set. However, when the use of additional predicates in the WHERE clause leads to a small number of documents in the potential result set, this may be not be a problem.

XMLType and XPath abstractions make it possible for you to develop applications that are independent 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 evolving **SQL/XML standard**. This ANSI standard defines a set of SQL functions that allow XML to be generated directly from a SELECT statement. Using these functions, a query can generate one or more XML documents, rather than a traditional tabular result set. The SQL/XML standard functions are allow almost any shape of XML data to be 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 16, "Generating XML Data from the Database"

Example 3–44 Using SQL/XML Functions to Generate XML

This query generates an XML document that contains information from the tables departments, locations, countries, employees, and jobs:

```
SELECT XMLElement(
         "Department",
         XMLAttributes(d.Department_id AS "DepartmentId"),
         XMLForest(d.department_name AS "Name"),
         XMLElement(
           "Location",
           XMLForest(street_address AS "Address",
                     city AS "City",
                     state_province AS "State",
                     postal_code AS "Zip",
                     country_name AS "Country")),
           XMLElement(
             "EmployeeList",
             (SELECT XMLAgg(
                       XMLElement(
                         "Employee",
                         XMLAttributes(e.employee_id AS "employeeNumber"),
                         XMLForest(
                           e.first_name AS "FirstName",
                           e.last_name AS "LastName",
                           e.email AS "EmailAddress",
                           e.phone_number AS "PHONE_NUMBER",
                           e.hire_date AS "StartDate",
                           j.job_title AS "JobTitle",
                           e.salary AS "Salary",
                           m.first_name || ' ' || m.last_name AS "Manager"),
                         XMLElement("Commission", e.commission_pct)))
                FROM hr.employees e, hr.employees m, hr.jobs j
                WHERE e.department_id = d.department_id
                  AND j.job_id = e.job_id
                  AND m.employee_id = e.manager_id)))
```

```
  AS XML
  FROM hr.departments d, hr.countries c, hr.locations l
  WHERE department_name = 'Executive'
    AND d.location_id = l.location_id
    AND l.country_id  = c.country_id;
```

The query returns the following XML:

```
XML
--------------------------------------------------------------------------------
<Department DepartmentId="90"><Name>Executive</Name><Location><Address>2004
 Charade Rd</Address><City>Seattle</City><State>Washingto
n</State><Zip>98199</Zip><Country>United States of
 America</Country></Location><EmployeeList><Employee
 employeeNumber="101"><FirstNa
me>Neena</FirstName><LastName>Kochhar</LastName><EmailAddress>NKOCHHAR</EmailAdd
ess><PHONE_NUMBER>515.123.4568</PHONE_NUMBER><Start
Date>1989-09-21</StartDate><JobTitle>Administration Vice
 President</JobTitle><Salary>17000</Salary><Manager>Steven King</Manager><Com
mission></Commission></Employee><Employee
 employeeNumber="102"><FirstName>Lex</FirstName><LastName>De
 Haan</LastName><EmailAddress>L
DEHAAN</EmailAddress><PHONE_NUMBER>515.123.4569</PHONE
NUMBER><StartDate>1993-01-13</StartDate><JobTitle>Administration Vice Presiden
t</JobTitle><Salary>17000</Salary><Manager>Steven
 King</Manager><Commission></Commission></Employee></EmployeeList></Department>
```

This query generates element `Department` for each row in the `departments` table.

- Each `Department` element contains attribute `DepartmentID`. The value of `DepartmentID` comes from the `department_id` column. The `Department` element contains sub-elements `Name`, `Location`, and `EmployeeList`.
- The text node associated with the `Name` element 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 of named elements from columns in the `locations` and `countries` tables. The values in the columns become the text node for the named element.
- The `Employeelist` element will contain an aggregation of `Employee` Elements. The content of the `EmployeeList` element is created by a subquery that returns the set of rows in the `employees` table that correspond to the current department. Each `Employee` element will contain information about the employee. The contents of the elements and attributes for each `Employee` element is taken from tables `employees` and `jobs`.

The output generated by the SQL/XML functions is *not* pretty-printed. This allows these functions to avoid creating a full DOM when generating the required output, and reduce the size of the generated document.

This lack of pretty-printing by SQL/XML functions will not matter to most applications. However, it makes verifying the generated output manually more difficult. When pretty-printing is required, invoke `XMLType` method `extract()` on the generated document to force construction of a DOM and pretty-print the 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 3–45 Forcing Pretty-Printing by Invoking Method extract() on the Result

This example shows how to force pretty-printing by invoking `XMLType` method `extract()` on the result generated by SQL function `XMLElement`.

```
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>1989-09-21</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
```

```
      <Commission/>
    </Employee>
    <Employee employeeNumber="102">
      <FirstName>Lex</FirstName>
      <LastName>De Haan</LastName>
      <EmailAddress>LDEHAAN</EmailAddress>
      <PHONE_NUMBER>515.123.4569</PHONE_NUMBER>
      <StartDate>1993-01-13</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 `XMLType` values. This means that you can use them to create `XMLType` *views* over conventional relational tables. Example 3–46 illustrates this. `XMLType` views are object views, so each row in the view must be identified by an object id. The object id must be specified in the `CREATE VIEW` statement.

Example 3–46 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, as shown in Example 3–47.

Example 3–47 Querying XMLType Views

This example shows a simple query against an XMLType view. The XPath expression passed to SQL function existsNode restricts the result set to the node that contains the Executive department information.

```
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>1989-09-21</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
    <Employee employeeNumber="102">
      <FirstName>Lex</FirstName>
      <LastName>De Haan</LastName>
      <EmailAddress>LDEHAAN</EmailAddress>
      <PHONE_NUMBER>515.123.4569</PHONE_NUMBER>
      <StartDate>1993-01-13</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
  </EmployeeList>
</Department>

1 row selected.
```

As can be seen from the following EXPLAIN PLAN output, Oracle XML DB is able to correctly rewrite the XPath in the existsNode expression 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.

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
Plan hash value: 1218413855

------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                   |    27 |  2160 |    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                      |                   |    27 |  2160 |     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 |     0   (0)| 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",LTRIM(TO_CHAR("E"."HIRE_DATE",'SYYYY-MM-DD'))),XMLELEMENT("JobTitle
              ","J"."JOB_TITLE"),CASE  WHEN "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
              ="Executive"]')=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)
```

```
59 rows selected.
```

> **Note:** XPath rewrite on XML expressions that operate on `XMLType` views is only supported when nodes referenced in the XPath expression are *not* descendants of an element created using SQL function `XMLAgg`.

Generating XML From Relational Tables Using DBURIType

Another way to generate XML from relational data is with SQL function `DBURIType`. Function `DBURIType` exposes one or more rows in a given table or view as a single XML document. The name of the root element is derived from the name of the table or view. The root element contains a set of `ROW` elements. There is one `ROW` element for each row in the table or view. The children of each `ROW` element are derived from the columns in the table or view. Each child element contains a text node with the value of the column for the given row.

Example 3–48 Accessing DEPARTMENTS Table XML Content Using DBURIType and getXML()

This example shows how to use SQL function `DBURIType` to access the contents of the `deptartments` table in schema `hr`. The example uses method `getXML()` to return the resulting document as an `XMLType` instance.

```
SELECT DBURIType('/HR/DEPARTMENTS').getXML() FROM DUAL;

DBURITYPE('/HR/DEPARTMENTS').GETXML()
-----------------------------------------------------
<?xml version="1.0"?>
<DEPARTMENTS>
 <ROW>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Administration</DEPARTMENT_NAME>
  <MANAGER_ID>200</MANAGER_ID>
  <LOCATION_ID>1700</LOCATION_ID>
 </ROW>
...
 <ROW>
  <DEPARTMENT_ID>20</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Marketing</DEPARTMENT_NAME>
  <MANAGER_ID>201</MANAGER_ID>
  <LOCATION_ID>1800</LOCATION_ID>
 </ROW>
</DEPARTMENTS>
```

SQL function `DBURIType` allows XPath notations to be used to control how much of the data in the table or view is returned when the table or view is accessed using `DBURIType`. Predicates in the XPath expression allow control over which of the rows in the table are included in the generated document.

Example 3–49 Using a Predicate in the XPath Expression to Restrict Which Rows Are Included

This example demonstrates how to use a predicate in an XPath expression to restrict the rows that are included in the generated XML document. Here, the XPath expression restricts the XML document to `DEPARTMENT_ID` columns with value `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, SQL function `DBURIType` provides a simple way to expose some or all rows in a relational table as one or more XML documents. The URL passed to function `DBURIType` can be extended to return a single column from the view or table, but in that case the URL must also include predicates that identify a single row in the target table or view. For example, the following URI would return just the value of the `department_name` column for the `departments` row where the `department_id` column has 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.
```

SQL function `DBURIType` does not provide the flexibility of the SQL/XML SQL functions: `DBURIType` provides no way to control the shape of the generated document. The data can only come from a single table or view. The generated document consists of one or more `ROW` elements. Each `ROW` element contains a child for each column in the target table. The names of the child elements are derived from the column names.

To control the names of the XML elements, to include columns from more than one table, or to control which columns from a table appear in the generated document, create a relational view that exposes the desired set of columns as a single row, and then use function `DBURIType` to generate an XML document from the contents of that view.

XSL Transformation and Oracle XML DB

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

> **See Also:** Appendix C, "XSLT Primer" for an introduction to the W3C XSL and XSLT recommendations

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 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.

Oracle XML DB includes an XSLT processor that allows XSL transformations to be performed *inside the database*. In this way, Oracle XML DB can provide XML-specific memory optimizations that significantly reduce the memory required to perform the transformation. It can also eliminate overhead associated with parsing the documents. These optimizations are only available when the source for the transformation is a *schema-based* XML document, however.

Oracle XML provides three options for invoking the XSL processor.

- SQL function `XMLtransform`
- `XMLType` method `transform()`
- PL/SQL package `DBMS_XSLPROCESSOR`

Each of these options expects the source document and XSL style sheet to be provided as `XMLType` objects. 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*, which is valid XML *and* valid HTML

Example 3–50 XSLT Style Sheet Example: PurchaseOrder.xsl

This example shows *part* of an XSLT style sheet, `PurchaseOrder.xsl`. The complete style sheet is given in "XSL Style Sheet Example, PurchaseOrder.xsl" on page C-4.

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xdb="http://xmlns.oracle.com/xdb"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsl:template match="/">
    <html>
      <head/>
      <body bgcolor="#003333" text="#FFFFCC" link="#FFCC00" vlink="#66CC99" alink="#669999">
        <FONT FACE="Arial, Helvetica, sans-serif">
          <xsl:for-each select="PurchaseOrder"/>
          <xsl:for-each select="PurchaseOrder">
            <center>
              <span style="font-family:Arial; font-weight:bold">
                <FONT COLOR="#FF0000">
                  <B>PurchaseOrder </B>
                </FONT>
              </span>
            </center>
            <br/>
            <center>
              <xsl:for-each select="Reference">
                <span style="font-family:Arial; font-weight:bold">
                  <xsl:apply-templates/>
                </span>
              </xsl:for-each>
            </center>
          </xsl:for-each>
          <P>
            <xsl:for-each select="PurchaseOrder">
              <br/>
            </xsl:for-each>
            <P/>
            <P>
              <xsl:for-each select="PurchaseOrder">
```

```
      <br/>
    </xsl:for-each>
  </P>
</P>
<xsl:for-each select="PurchaseOrder"/>
<xsl:for-each select="PurchaseOrder">
  <table border="0" width="100%" BGCOLOR="#000000">
    <tbody>
      <tr>
        <td WIDTH="296">
          <P>
            <B>
              <FONT SIZE="+1" COLOR="#FF0000" FACE="Arial, Helvetica, sans-serif">Internal</FONT>
            </B>
          </P>

          ...

        </td>
        <td width="93"/>
        <td valign="top" WIDTH="340">
          <B>
            <FONT COLOR="#FF0000">
              <FONT SIZE="+1">Ship To</FONT>
            </FONT>
          </B>
          <xsl:for-each select="ShippingInstructions">
            <xsl:if test="position()=1"/>
          </xsl:for-each>
          <xsl:for-each select="ShippingInstructions">
          </xsl:for-each>

            ...
```

These is nothing Oracle XML DB-specific about this 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 transformation operates on a *small subset* of the nodes in the source document.

In traditional XSL processors, the entire source document must be parsed and loaded into memory before XSL processing can begin. This process requires significant amounts of memory and processor. When only a small part of the document is processed this is inefficient.

When Oracle XML DB performs XSL transformations on a *schema-based* XML document there is no need to parse the document before processing can begin. The lazily loaded virtual DOM eliminates the need to parse the document, by loading content directly from disk as the nodes are accessed. The lazy load also reduces the amount of memory required to perform the transformation, because only the parts of the document that are processed are loaded into memory.

Example 3–51 Applying a Style Sheet Using TRANSFORM

This example shows how to use SQL function `XMLtransform` to apply an XSL style sheet to a document stored in an `XMLType` table, producing HTML code. SQL function `XDBURIType` reads the XSL style sheet from Oracle XML DB Repository. Method `extract()` is called here on the result of `XMLtransform` merely to force pretty-printing, for clarity.

In the interest of brevity, only part of the result of the transformation is shown here; omitted parts are indicated with an ellipsis (. . .). Figure 3–8 shows what the transformed result looks like in a Web browser.

```
SELECT
  XMLtransform(
    OBJECT_VALUE,
    XDBURIType('/source/schemas/poSource/xsl/purchaseOrder.xsl').getXML()).extract('/*')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]')=1;

XMLTRANSFORM(OBJECT_VALUE,
             XDBURITYPE('/SOURCE/SCHEMAS/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">
                                                                  . . .
              </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">
                . . .
              </table>
            </td>
          </tr>
        </tbody>
      </table>
      <br/>
      <B>
```

```
        <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
      </B>
      <br/>
      <br/>
      <table border="0">
        . . .
      </table>
    </FONT>
  </body>
</html>

1 row selected.
```

> **See Also:** Chapter 9, "Transforming and Validating XMLType Data"

Using Oracle XML DB Repository

Oracle XML DB Repository makes it possible to organize XML content using a file - folder metaphor. This lets you use a URL to uniquely identify XML documents stored in the database. This approach appeals to XML developers used to using constructs such as URLs and XPath expressions to identify content.

Oracle XML DB Repository is modelled on the DAV standard. The DAV standard uses the term **resource** to describe any file or folder managed by a WebDAV server. A resource consists of a combination of metadata and content. The DAV specification defines the set of (system-defined) metadata properties that a WebDAV server is expected to maintain for each resource and the set of XML documents that a DAV server and DAV-enabled client uses to exchange metadata.

Although Oracle XML DB Repository can manage any kind of content, it provides specialized capabilities and optimizations related to managing resources where the content is XML.

Installing and Uninstalling Oracle XML DB Repository

All of the metadata and content managed by Oracle XML DB Repository is stored using a set of tables in the database schema owned by database 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 Oracle XML DB Repository and invalidate any `XMLType` tables or columns associated with registered XML schemas.

Oracle XML DB Provides Name-Level Locking

When using a relational database to maintain hierarchical folder structures, ensuring a high degree of concurrency when adding and removing items in a folder is a challenge. In conventional file system there is no concept of a transaction. Each operation (add a file, create a subfolder, rename a file, delete a file, and so on) is treated as an atomic transaction. Once the operation has completed the change is immediately available to all other users of the file system.

Note: As a consequence of transactional semantics enforced by the database, folders created using SQL statements will *not* be visible to other database users until the transaction is committed. *Concurrent* access to Oracle XML DB Repository is controlled by the same mechanism used to control concurrency in Oracle Database. The integration of the repository with Oracle Database provides *strong management options for XML content*.

One key advantage of Oracle XML DB Repository is the ability to use SQL for repository operations in the context of a logical transaction. Applications can create long-running transactions that include updates to one or more folders. In this situation a conventional locking strategy that takes an exclusive lock on each updated folder or directory tree would quickly result in significant concurrency problems.

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(S), 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.

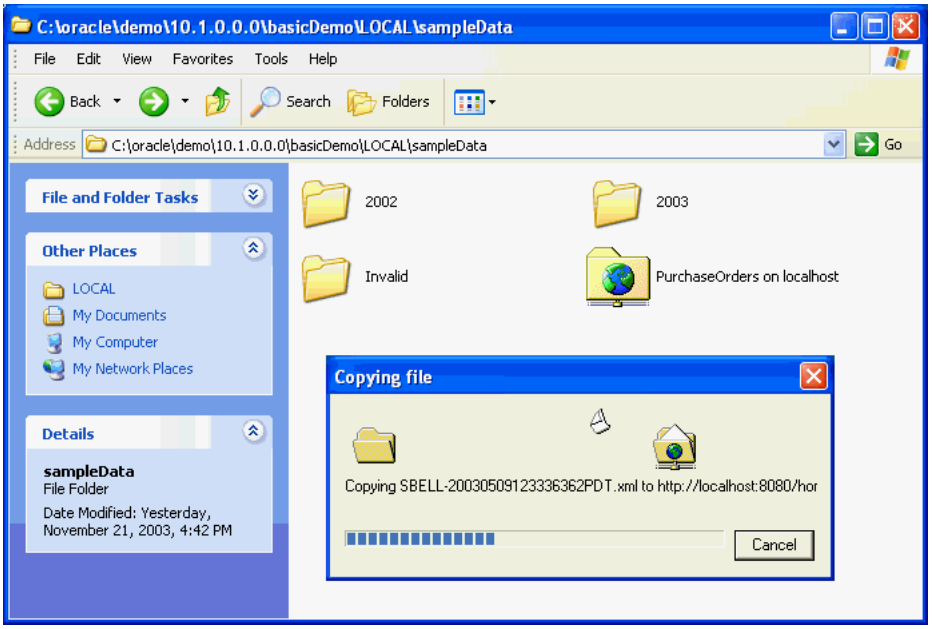
Using Standard Protocols to Store and Retrieve Content

Oracle XML DB supports industry-standard internet protocols such as HTTP(S), WebDav, and FTP. The combination of protocol support and URL-based access makes it possible to insert, retrieve, update, and delete content stored in Oracle Database from standard desktop applications such as Windows Explorer, Microsoft Word, and XMLSpy.

Figure 3–4 shows Windows Explorer used to insert a folder from the local hard drive into Oracle Database. Windows Explorer includes support for the WebDAV protocol. WebDAV extends the HTTP standard, adding additional verbs that allow an HTTP server to act as a file server.

When a Windows Explorer copy operation or FTP input command is used to transfer a number of documents into Oracle XML DB Repository, each `put` or `post` command is treated as a separate atomic operation. This ensures that the client does not get confused if one of the file transfers fails. It also means that changes made to a document through a protocol are visible to other users as soon as the request has been processed.

Figure 3–4 Copying Files into Oracle XML DB Repository



Uploading Content 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 3–52 Uploading Content into the Repository Using FTP

```
$ ftp mdrake-sun 2100
Connected to mdrake-sun.
220 mdrake-sun FTP Server (Oracle XML DB/Oracle Database 10g Enterprise Edition
Release 10.1.0.1.0 - Beta) ready.
Name (mdrake-sun:oracle10): QUINE
331 pass required for QUINE
Password:
230 QUINE logged in
ftp> cd /source/schemas
250 CWD Command successful
ftp> mkdir PurchaseOrders
257 MKD Command successful
ftp> cd PurchaseOrders
250 CWD Command successful
ftp> mkdir 2002
257 MKD Command successful
ftp> cd 2002
250 CWD Command successful
ftp> mkdir "Apr"
257 MKD Command successful
ftp> put "Apr/AMCEWEN-20021009123336171PDT.xml"
```

```
"Apr/AMCEWEN-20021009123336171PDT.xml"
200 PORT Command successful
150 ASCII Data Connection
226 ASCII Transfer Complete
local: Apr/AMCEWEN-20021009123336171PDT.xml remote:
Apr/AMCEWEN-20021009123336171PDT.xml
4718 bytes sent in 0.0017 seconds (2683.41 Kbytes/s)
ftp> put "Apr/AMCEWEN-20021009123336271PDT.xml"
"Apr/AMCEWEN-20021009123336271PDT.xml"
200 PORT Command successful
150 ASCII Data Connection
226 ASCII Transfer Complete
local: Apr/AMCEWEN-20021009123336271PDT.xml remote:
Apr/AMCEWEN-20021009123336271PDT.xml
4800 bytes sent in 0.0014 seconds (3357.81 Kbytes/s)
.....
ftp> cd "Apr"
250 CWD Command successful
ftp> ls -l
200 PORT Command successful
150 ASCII Data Connection
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 AMCEWEN-20021009123336171PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 AMCEWEN-20021009123336271PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 EABEL-20021009123336251PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 PTUCKER-20021009123336191PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 PTUCKER-20021009123336291PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SBELL-20021009123336231PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SBELL-20021009123336331PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SKING-20021009123336321PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SMCCAIN-20021009123336151PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SMCCAIN-20021009123336341PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 VJONES-20021009123336301PDT.xml
226 ASCII Transfer Complete
remote: -l
959 bytes received in 0.0027 seconds (349.45 Kbytes/s)
ftp> cd ".."
250 CWD Command successful
....
ftp> quit
221 QUIT Goodbye.
$
```

The key point demonstrated by both of these examples is that neither Windows Explorer nor the FTP tool is aware that it is working with Oracle XML DB. Since the tools and Oracle XML DB both support open Internet protocols they simply work with each other out of the box.

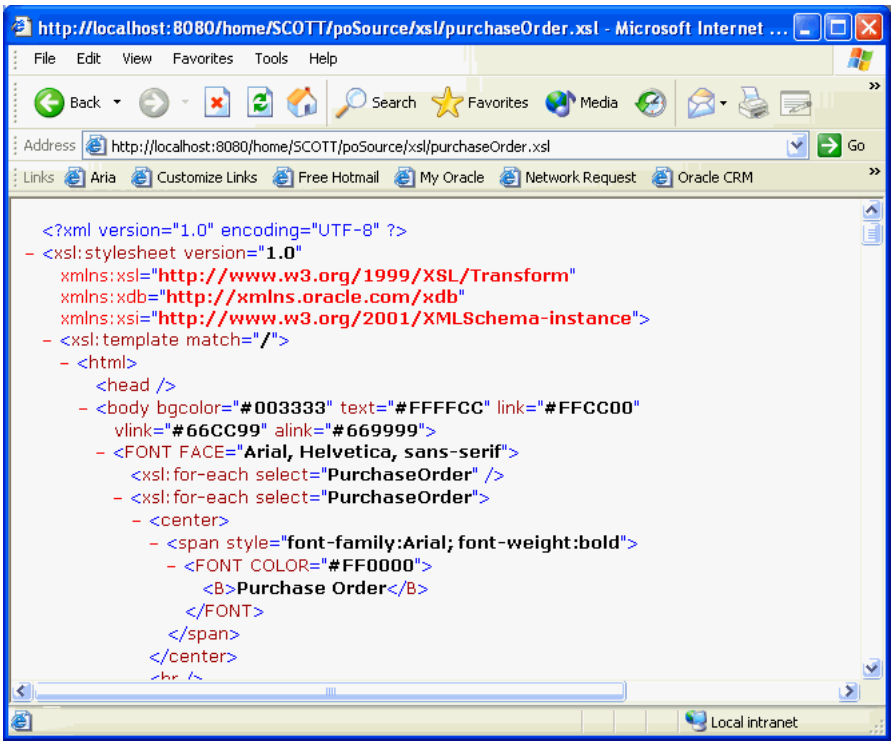
Any tool that understands the WebDAV or FTP protocol can be used to create content managed by Oracle XML DB Repository. No additional software has to installed on the client or the mid-tier.

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 zero (0) bytes. This was designed as such for two reasons:

- It is not clear what the size of the document should be. Is it the size of the `CLOB` instance generated by printing the document, or the number of bytes required to store the objects used to persist the document inside the database?
- Regardless of which definition is chosen, calculating and maintaining this information is costly.

Figure 3–5 shows Internet Explorer using a URL and the HTTP protocol to view an XML document stored in the database.

Figure 3–5 Path-Based Access Using HTTP and a URL



Accessing Oracle XML DB Repository Programmatically

Oracle XML DB Repository can be accessed and updated directly from SQL. 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 the repository. Oracle XML DB includes PL/SQL package `DBMS_XDB`, which provides methods that allow resources to be created, modified, and deleted programmatically.

Example 3–53 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/QUINE/NurseryRhyme.txt',
                                 bfilename('XMLDIR', 'tdadxdb-03-01.txt'),
                                 nls_charset_id('AL32UTF8'));
END;
/
```

Accessing and Updating XML Content in the Repository

This section describes features for accessing and updating Oracle XML DB Repository content.

Access XML Documents Using SQL

Content stored in the repository can be accessed and updated from SQL and PL/SQL. You can interrogate the structure of the repository in complex ways. For example, you can query to determine how many files with extension `.xsl` are under a location other than `/home/mystylesheetdir`.

You can also mix path-based repository access with content-based access. You can, for example, ask "How many documents not under `/home/purchaseOrders` have a node identified by the XPath `/PurchaseOrder/User/text()` with a value of `KING`?"

All of the *metadata* for managing the repository is stored in a database schema owned by 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 resource (file or folder) in the repository. Documents in this table are referred to as **resource documents** . The XML schema that defines the structure of an Oracle XML DB resource document is registered under URL, `"http://xmlns.oralce.com/xdb/XDBResource.xsd`.

Repository Content is Exposed Through RESOURCE_VIEW and PATH_VIEW

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

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

Use EXISTS_PATH and UNDER_PATH to Include Path-Based Predicates in the WHERE Clause

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

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

Documents Other Than XML Can Be Stored In the Repository

It is also possible to store other kinds of documents in the repository. When a document that is not XML or is not schema-based XML is stored in the repository, the document *content* is stored in a LOB along with the metadata about the document.

PL/SQL Packages to Create, Delete, Rename, Move, ... Folders and Documents

Since Oracle XML DB repository can be accessed and updated using SQL, any application capable of calling a PL/SQL procedure can use the 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 the repository Access Control Lists (ACLs).

With supplied PL/SQL packages `DBMS_XDB`, `DBMS_XDBZ`, and `DBMS_XDB_VERSION`, you can create, delete, and rename documents and folders, move a file or folder within

the folder hierarchy, set and change the access permissions on a file or folder, and initiate and manage versioning.

Example 3–54 Using PL/SQL Package DBMS_XDB To Create Folders

This example shows PL/SQL package `DBMS_XDB` used to create a set of subfolders beneath folder `/public`.

```
DECLARE
  RESULT BOOLEAN;
BEGIN
  IF (NOT DBMS_XDB.existsResource('/public/mysource')) THEN
     result := DBMS_XDB.createFolder('/public/mysource');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/mysource/schemas')) THEN
     result := DBMS_XDB.createFolder('/public/mysource/schemas');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/mysource/schemas/poSource')) THEN
     result := DBMS_XDB.createFolder('/public/mysource/schemas/poSource');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/mysource/schemas/poSource/xsd')) THEN
     result := DBMS_XDB.createFolder('/public/mysource/schemas/poSource/xsd');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/mysource/schemas/poSource/xsl')) THEN
     result := DBMS_XDB.createFolder('/public/mysource/schemas/poSource/xsl');
  END IF;
END;
/
```

Accessing the Content of Documents Using SQL

You can access the content of documents stored in Oracle XML DB Repository in several ways. The easiest way is to use `XDBURIType`. `XDBURIType` uses a URL to specify which resource to access. The URL passed to the `XDBURIType` is assumed to start at the root of the repository. Datatype `XDBURIType` provides methods `getBLOB()`, `getCLOB()`, and `getXML()` to access the different kinds of content that can be associated with a resource.

Example 3–55 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/QUINE/NurseryRhyme.txt').getClob()
  FROM DUAL;

XDBURITYPE('/HOME/QUINE/NURSERYRHYME.TXT').GETCLOB()
--------------------------------------------------
Mary had a little lamb
Its fleece was white as snow
and everywhere that Mary went
that lamb was sure to go

1 row selected.
```

Example 3–56 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/QUINE/NurseryRhyme.txt') = 1;

DBMS_XMLGEN.CONVERT(EXTRACT(RES,'/RESOURCE/CONTENTS/TEXT/TEXT()','XMLNS="HTTP://
--------------------------------------------------------------------------------
Mary had a little lamb
Its fleece was white as snow
and everywhere that Mary went
that lamb was sure to go

1 row selected.
```

SQL function `extract`, rather than `extractValue`, is used to access the text node. This returns the content of the text node as an `XMLType` instance, which makes it possible to access the content of the node using `XMLType` method `getCLOBVal()`. Hence, you can access the content of documents larger than 4K. Here, `DBMS_XMLGEN.convert` removes any entity escaping from the text.

Example 3–57 Accessing XML Documents Using 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(des), '/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"'))) des
  WHERE
    equals_path(RES, '/home/QUINE/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 3–58 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 function ref locates the target row in the default table based on value of the XMLRef element contained in the resource document.

```
SELECT extractValue(value(des), '/Description')
  FROM RESOURCE_VIEW r,
       purchaseorder p,
       table(XMLSequence(extract(OBJECT_VALUE,
                                 '/PurchaseOrder/LineItems/LineItem/Description'))) des
 WHERE equals_path(res, '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.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 3–59 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/QUINE/PurchaseOrders/2002/Mar and contain orders for part number 715515009058.

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference')
  FROM RESOURCE_VIEW r, purchaseorder p
  WHERE under_path(res, '/home/QUINE/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-2002100912333661PDT
SBELL-2002100912333601PDT
```

```
3 rows selected.
```

In general, when accessing the content of schema-based XML documents, joining `RESOURCE_VIEW` or `PATH_VIEW` with the default table is more efficient than using 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 the Repository

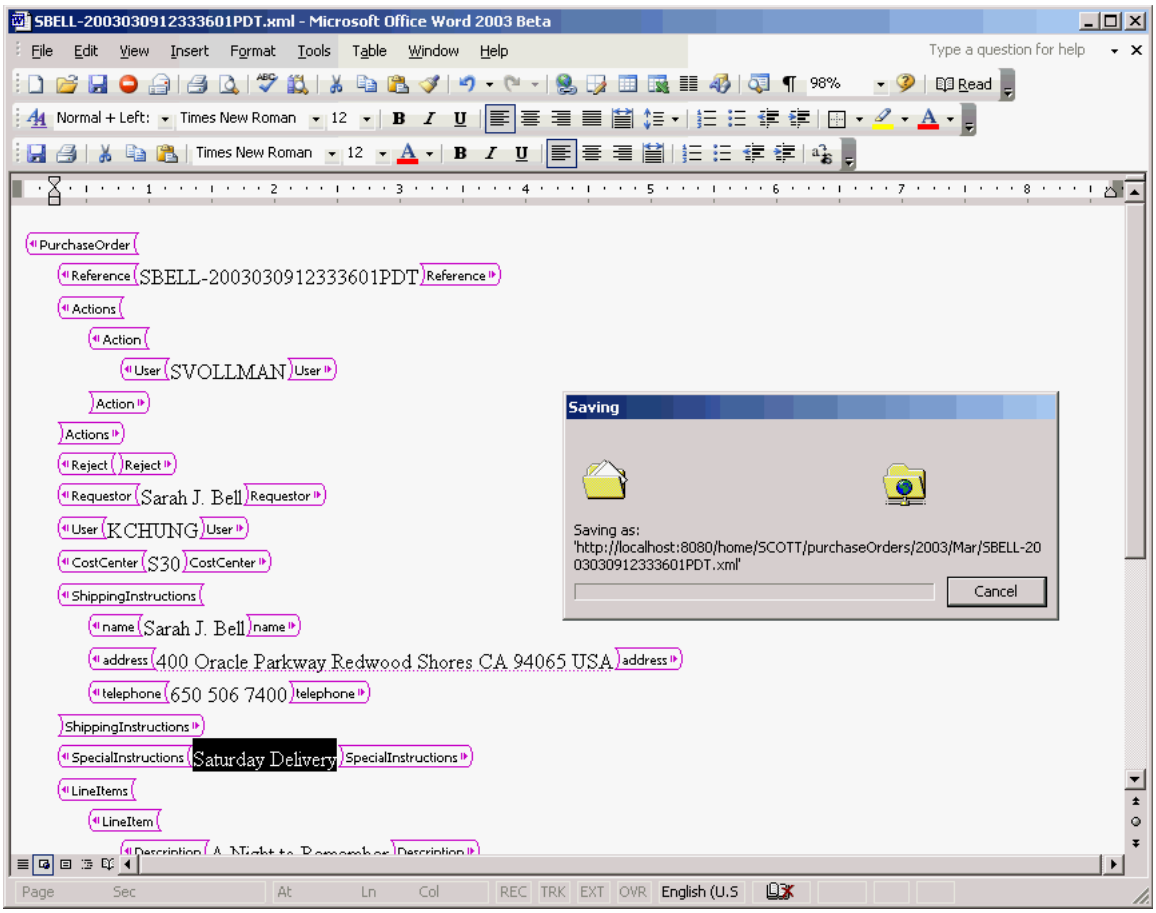
You can also update the content of documents stored in Oracle XML DB Repository using protocols or SQL.

Updating Repository Content Using Protocols

The most popular content authoring tools now support HTTP, FTP, and WebDAV protocols. These tools can use a URL and the HTTP verb `get` to access the content of a document, and the HTTP verb `put` to save the contents of a document. Hence, given the appropriate access permissions, a simple URL is all you need to access and edit content stored in Oracle XML DB Repository.

Figure 3–6 shows how, with the WebDAV support included in Microsoft Word, you can use Microsoft Word to update and edit a document stored in Oracle XML DB Repository.

Figure 3–6 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

SQL functions such as `updateXML` can be used to update the content of any document stored in Oracle XML DB Repository. The content of the document can be modified by updating the resource document, or, in the case of schema-based XML documents, by updating the default table that contains the content of the document.

Example 3–60 Updating a 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 SQL function `updateXML` on the resource document. An XPath expression is passed to `updateXML` as the target of the update operation, identifying the text node belonging to element `/Resource/Contents/text`.

```
DECLARE
  file         BFILE;
  contents     CLOB;
  dest_offset  NUMBER := 1;
```

```
  src_offset   NUMBER := 1;
  lang_context NUMBER := 0;
  conv_warning NUMBER := 0;
BEGIN
  file := bfilename('XMLDIR', 'tdadxdb-03-02.txt');
  DBMS_LOB.createTemporary(contents, true, DBMS_LOB.SESSION);
  DBMS_LOB.fileopen(file, DBMS_LOB.file_readonly);
  DBMS_LOB.loadClobfromFile(contents,
                            file,
                            DBMS_LOB.getLength(file),
                            dest_offset,
                            src_offset,
                            nls_charset_id('AL32UTF8'),
                            lang_context,
                            conv_warning);
  DBMS_LOB.fileclose(file);
  UPDATE RESOURCE_VIEW
    SET res = updateXML(res,
                        '/Resource/Contents/text/text()',
                        contents,
                        'xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"')
      WHERE equals_path(res, '/home/QUINE/NurseryRhyme.txt') = 1;
  DBMS_LOB.freeTemporary(contents);
END;
/
```

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 Oracle XML DB Repository.

Example 3–61 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, SQL function `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/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
        = 1;

1 row updated.

SELECT extractValue(res,
                    '/r:Resource/r:Contents/PurchaseOrder/User/text()',
                    'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"')
  FROM RESOURCE_VIEW
  WHERE equals_path(
          res,
          '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
        = 1;

EXTRACTVALUE(RES, '/R:RESOURCE/R:CONTENTS/PURCHASEORDER/USER/TEXT()',
             'XMLNS:R="HTTP://XMLNS.ORACLE.COM/XDB/XDBRESOURCE.XSD"')
--------------------------------------------------------------------
```

```
SKING

1 row selected.
```

Updating XML Schema-Based Documents in the Repository

You can update XML schema-based XML documents by performing the update operation directly on the default table 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 3–62 Updating XML Schema-Based Documents in the Repository

In this example, SQL function `updateXML` operates on the default table with the target row identified by a path. The row to be updated is identified by a `ref`. The value of the `ref` is obtained from the resource document identified by SQL function `equals_path`. 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/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
               = 1);

1 row updated.

SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/User/text()')
  FROM purchaseorder p, RESOURCE_VIEW
  WHERE ref(p) = extractValue(res, '/Resource/XMLRef')
    AND equals_path(res, '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
        = 1;

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

1 row selected.
```

Controlling Access to Repository Data

You can control access to the resources in Oracle XML DB Repository by using Access Control Lists (ACLs). An ACL is a list of access control entries, each of which grants or denies a set of privileges to a specific principal. The principal can be a database user, a database role, an LDAP user, an LDAP group or the special principal `dav:owner`, which refers to the owner of the resource. Each resource in the repository is protected by an ACL. The ACL determines what privileges, such as `read-properties` and

update, a user has on the resource. Each repository operation includes a check of the ACL to determine if the current user is allowed to perform the operation.

By default, a new resource inherits the ACL of its parent folder. But you can set the ACL of a resource using procedure DBMS_XDB.setACL(). For more details on Oracle XML DB resource security, see Chapter 24, "Repository Resource Security".

In the following example, the current user is QUINE. The query gives the number of resources in the folder /public. Assume that there are only two resources in this folder: f1 and f2. Also assume that the ACL on f1 grants the read-properties privilege to QUINE while the ACL on f2 does not grant QUINE any privileges. A user needs the read-properties privilege on a resource for it to be visible to the user. The result of the query is 1, because only f1 is visible to QUINE.

```
SELECT count(*) FROM RESOURCE_VIEW r
  WHERE under_path(r.res, '/public') = 1;

  COUNT(*)
----------
         1
```

Oracle XML DB Transactional Semantics

When working from SQL, normal transactional behavior is enforced. Multiple calls to SQL functions such as updateXML can be used within a single logical unit of work. Changes made through functions like updateXML are not visible to other database users until the transaction is committed. At any point, ROLLBACK can be used to back out the set of changes made since the last commit.

Querying Metadata and the Folder Hierarchy

In Oracle XML DB, the system-defined metadata for each resource is preserved as an XML document. The structure of these resource documents is defined by 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 and PATH_VIEW

RESOURCE_VIEW contains one entry for each file or folder stored in Oracle XML DB Repository. The view has two columns. Column RES contains the resource – an XML document that manages the metadata properties associated with the resource content. Column ANY_PATH contains a valid URL that the current user can pass to XDBURIType to access the resource content. If this content is not binary data, then the resource itself also contains the content.

Oracle XML DB supports the concept of **linking**. Linking makes it possible to define multiple paths to a given document. A separate XML document, called the **link-properties document**, maintains metadata properties that are specific to the path, rather than to the resource. Whenever a resource is created, an initial link is also created.

PATH_VIEW exposes the link-properties documents. There is one entry in PATH_VIEW for each possible path to a document. PATH_VIEW has three columns. Column RES contains the resource document pointed to by this link. Column PATH contains the

path that the link allows to be used to access the resource. Column LINK contains the link-properties document (metadata) for this PATH.

Example 3–63 Viewing RESOURCE_VIEW and PATH_VIEW Structures

The following example shows the description of public views RESOURCE_VIEW and PATH_VIEW:

```
DESCRIBE RESOURCE_VIEW

Name      Null?    Type
----------------------------------------------------------
RES                SYS.XMLTYPE(XMLSchema
                               "http://xmlns.oracle.com/xdb/XDBResource.xsd"
                               Element
                               "Resource")
ANY_PATH           VARCHAR2(4000)
RESID              RAW(16)


DESCRIBE PATH_VIEW

Name      Null?    Type
----------------------------------------------------------
PATH               VARCHAR2(1024)
RES                SYS.XMLTYPE(XMLSchema
                               "http://xmlns.oracle.com/xdb/XDBResource.xsd"
                               Element
                               "Resource")
LINK               SYS.XMLTYPE
RESID              RAW(16)
```

See Also: Chapter 22, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"

Querying Resources in RESOURCE_VIEW and PATH_VIEW

Oracle XML DB provides two SQL functions, equals_path and under_path, that can be used to perform **folder-restricted queries**. Such queries limit SQL statements that operate on the RESOURCE_VIEW or PATH_VIEW to documents that are at a particular location in Oracle XML DB folder hierarchy. Function equals_path restricts the statement to a single document identified by the specified path. Function under_path restricts the statement to those documents that exist beneath a certain point in the hierarchy.

The following examples demonstrate simple folder-restricted queries against resource documents stored in RESOURCE_VIEW and PATH_VIEW.

Example 3–64 Accessing Resources Using EQUALS_PATH and RESOURCE_VIEW

The following query uses SQL function equals_path and RESOURCE_VIEW to access the resource created in Example 3–63.

```
SELECT r.RES.getClobVal()
  FROM RESOURCE_VIEW r
  WHERE equals_path(res, '/home/QUINE/NurseryRhyme.txt') = 1;

R.RES.GETCLOBVAL()
--------------------------------------------------------------------------------
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
```

```
          Hidden="false"
          Invalid="false"
          Container="false"
          CustomRslv="false"
          VersionHistory="false"
          StickyRef="true">
  <CreationDate>2004-08-06T12:12:48.022251</CreationDate>
  <ModificationDate>2004-08-06T12:12:53.215519</ModificationDate>
  <DisplayName>NurseryRhyme.txt</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>text/plain</ContentType>
  <RefCount>1</RefCount>
  <ACL>
    <acl description=
           "Private:All privileges to OWNER only and not accessible to others"
         xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
         http://xmlns.oracle.com/xdb/acl.xsd">
      <ace>
        <principal>dav:owner</principal>
        <grant>true</grant>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>QUINE</Owner>
  <Creator>QUINE</Creator>
  <LastModifier>QUINE</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#text</SchemaElement>
  <Contents>
    <text>Hickory Dickory Dock
The Mouse ran up the clock
The clock struck one
The Mouse ran down
Hickory Dickory Dock
    </text>
  </Contents>
</Resource>

1 row selected.
```

As Example 3–64 shows, a resource document is an XML document that captures the set of metadata defined by the DAV standard. The metadata includes information such as `CreationDate`, `Creator`, `Owner`, `ModificationDate`, and `DisplayName`. The content of the resource document can be queried and updated just like any other XML document, using SQL functions such as `extract`, `extractValue`, `existsNode`, and `updateXML`.

Example 3–65 Determining the Path to XSL Style Sheets Stored in the Repository

The first query finds a path to each of the XSL style sheets stored in Oracle XML DB Repository. It performs a search based on the `DisplayName` ending in `.xsl`.

```
SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE extractValue(RES, '/Resource/DisplayName') LIKE '%.xsl';
```

```
ANY_PATH
----------------------------------------
/source/schemas/poSource/xsl/empdept.xsl
/source/schemas/poSource/xsl/purchaseOrder.xsl

2 rows selected.
```

Example 3–66 Counting Resources Under a Path

This example counts the number of resources (files and folders) under the path `/home/QUINE/PurchaseOrders`. Using `RESOURCE_VIEW` rather than `PATH_VIEW` ensures that any resources that are the target of multiple links are only counted once. SQL function `under_path` restricts the result set to documents that can be accessed using a path that starts from `/home/QUINE/PurchaseOrders`.

```
SELECT count(*)
   FROM RESOURCE_VIEW
   WHERE under_path(RES, '/home/QUINE/PurchaseOrders') = 1;

  COUNT(*)
----------
       145

1 row selected.
```

Example 3–67 Listing the Folder Contents in a Path

This query lists the contents of the folder identified by path `/home/QUINE/PurchaseOrders/2002/Apr`. This is effectively a directory listing of the folder.

```
SELECT PATH
  FROM PATH_VIEW
  WHERE under_path(RES, '/home/QUINE/PurchaseOrders/2002/Apr') = 1;

PATH
--------------------------------------------------------------------------------
/home/QUINE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/PTUCKER-20021009123336191PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/PTUCKER-20021009123336291PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336231PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336331PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SKING-20021009123336321PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336151PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/VJONES-20021009123336301PDT.xml

11 rows selected.
```

Example 3–68 Listing the Links Contained in a Folder

This query lists the set of links contained in the folder identified by the path `/home/QUINE/PurchaseOrders/2002/Apr` where the `DisplayName` element in the associated resource starts with an `S`.

```
SELECT PATH
  FROM PATH_VIEW
  WHERE extractValue(RES, '/Resource/DisplayName') like 'S%'
    AND under_path(RES, '/home/QUINE/PurchaseOrders/2002/Apr') = 1;
```

```
PATH
--------------------------------------------------------------------------------
/home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336231PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336331PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SKING-20021009123336321PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336151PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml

5 rows selected.
```

Example 3–69 Finding Paths to Resources that Contain Purchase-Order XML Documents

This query finds a path to each resource in Oracle XML DB Repository that contains a `PurchaseOrder` document. The documents are identified based on the metadata property `SchemaElement` that identifies the XML schema URL and global element for schema-based XML data stored in the repository.

```
SELECT ANY_PATH
  FROM RESOURCE_VIEW
  WHERE existsNode(RES,
                   '/Resource[SchemaElement=
                      "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd#PurchaseOrder"]')
        = 1;
```

This returns the following paths, each of which contains a `PurchaseOrder` document:

```
ANY_PATH
--------------------------------------------------------------------------------
/home/QUINE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/PTUCKER-20021009123336191PDT.xml

...

132 rows selected.
```

Oracle XML DB Hierarchical 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, so path-based access and folder-restricted queries would become inefficient as the number of documents and depth of the folder hierarchy increase.

To address this issue, Oracle XML DB introduces a new index type, the **hierarchical index**. A 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 3–70 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/QUINE/PurchaseOrders/2002/Apr') = 1;

Explained.

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------
Plan hash value: 2568289845

-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |   127 | 22606 |    35   (9)| 00:00:01 |
|   1 |  NESTED LOOPS                       |               |   127 | 22606 |    35   (9)| 00:00:01 |
|   2 |   NESTED LOOPS                      |               |   127 | 20447 |    34   (6)| 00:00:01 |
|   3 |    NESTED LOOPS                     |               |   127 | 16891 |    34   (6)| 00:00:01 |
|*  4 |     TABLE ACCESS BY INDEX ROWID     | XDB$RESOURCE  |     1 |   131 |     3   (0)| 00:00:01 |
|*  5 |      DOMAIN INDEX                   | XDBHI_IDX     |       |       |            |          |
|   6 |     COLLECTION ITERATOR PICKLER FETCH|               |       |       |            |          |
|*  7 |    INDEX UNIQUE SCAN                | XDB_PK_H_LINK |     1 |    28 |     0   (0)| 00:00:01 |
|*  8 |   INDEX UNIQUE SCAN                 | SYS_C003013   |     1 |    17 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

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

   4 - filter("P"."SYS_NC00011$" LIKE 'S%')
   5 - access("XDB"."UNDER_PATH"(SYS_MAKEXML('8758D485E6004793E034080020B242C6',734,"XMLEXTRA"
              ,"XMLDATA"),'/home/QUINE/PurchaseOrders/2002/Apr',9999)=1)
   7 - access("H"."PARENT_OID"=SYS_OP_ATG(VALUE(KOKBF$),3,4,2) AND
              "H"."NAME"=SYS_OP_ATG(VALUE(KOKBF$),2,3,2))
   8 - access("R2"."SYS_NC_OID$"=SYS_OP_ATG(VALUE(KOKBF$),3,4,2))

25 rows selected.
```

How Documents are Stored in the Repository

Oracle XML DB provides special handling for XML documents. The rules for storing the contents of schema-based XML document are defined by the XML schema. The content of the document is stored in the default table associated with the global element definition.

Oracle XML DB Repository also stores files that do not contain XML data, such as JPEG images or Word documents. The XML schema for each resource defines which elements are allowed, and specifies whether the content of these files is to be stored as `BLOB` or `CLOB` instances. The content of a non-schema-based XML document is stored as a `CLOB` instance in the repository.

There is one resource and one link-properties document for every file or folder in the 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 the repository.

When an XML file is loaded into the 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 the 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 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(S) 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: "DBUriServlet" on page 19-25

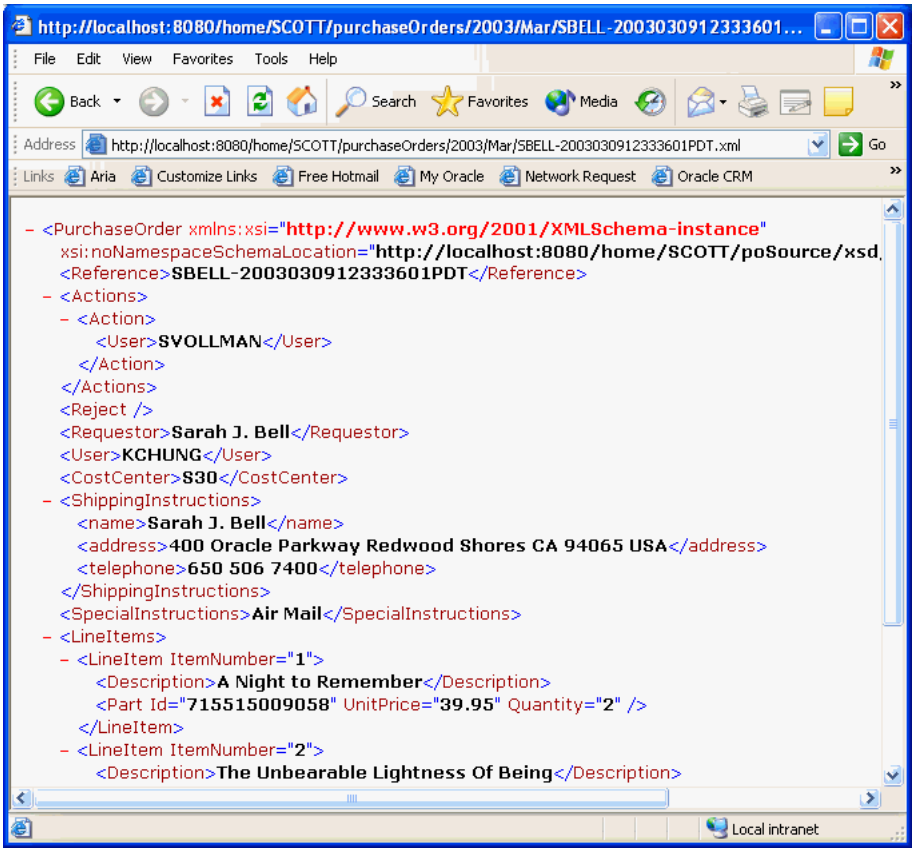
Style sheet processing is initiated by specifying a transform parameter as part of the URL passed to `DBUri servlet`. The style sheet is specified using a URI that

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 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.

Figure 3–7 shows how `DBUri` can access a row in the `purchaseorder` table.

Figure 3–7 Using DBUri Servlet to Access XML Content



The root of the URL is `/oradb`, so the URL is passed to the DBUri servlet that accesses the `purchaseorder` table in the `SCOTT` database schema, rather than as a resource in Oracle XML DB Repository. The URL includes an XPath expression that restricts the result set to those documents where node `/PurchaseOrder/Reference/text()` contains the value specified in the predicate. The `contenttype` parameter sets the MIME type of the generated document to `text/xml`.

XSL Transformation Using DBUri Servlet

Figure 3–8 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 SQL functino `XMLtransform` to apply the style sheet `/home/SCOTT/xsl/purchaseOrder.xsl`

to the `PurchaseOrder` document identified by the main URL, before returning the document to the browser. This style sheet transforms the XML document to a more user-friendly HTML page. The URL also uses `contentType` parameter to specify that the MIME-type of the final document will be `text/html`.

Figure 3–8 Database XSL Transformation of a PurchaseOrder Using DBUri Servlet

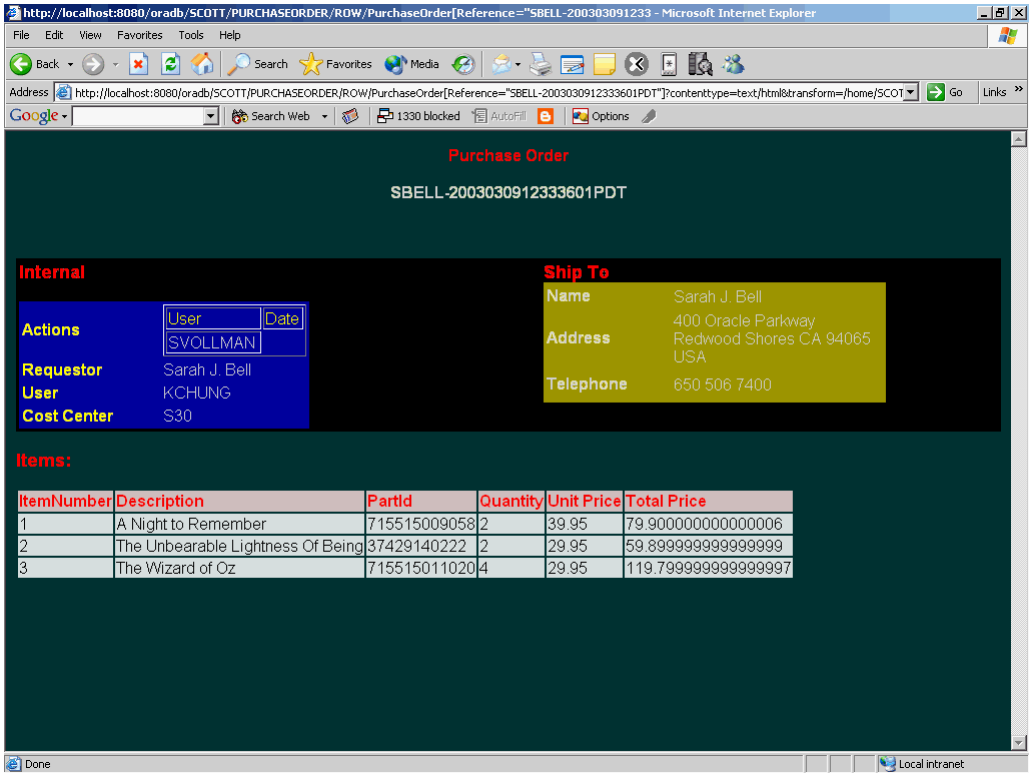
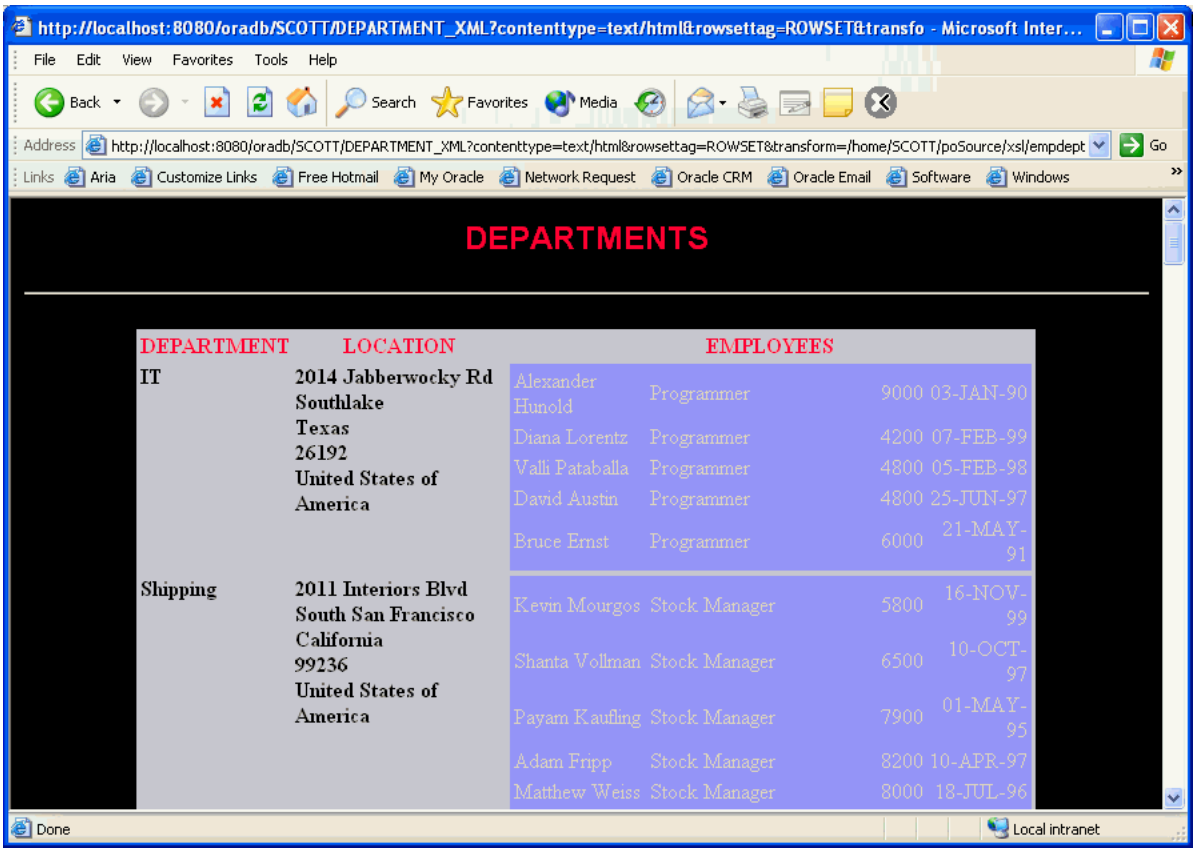


Figure 3–9 shows the `departments` table displayed as an HTML document. You need no code to achieve this, you only need an `XMLType` view, based on SQL/XML functions, an industry-standard XSL style sheet, and `DBUri` servlet.

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 applications (schema-based and non-schema-based). It includes guidelines for creating, manipulating, updating, querying, and indexing `XMLType` columns and tables.

This chapter contains these topics:

- Selecting and Querying XML Data
- Updating XML Instances and XML Data in Tables
- Indexing XMLType Columns

Note:

- ***Non-schema-based XML:*** `XMLType` tables and columns described in this chapter are not based on the 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 XML:*** Appendix A, "XML Schema Primer" and Chapter 5, "XML Schema Storage and Query: Basic" describe how to work with XML schema-based `XMLType` tables and columns.

Selecting and Querying XML Data

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

- Select `XMLType` columns in SQL, PL/SQL, or Java.
- Query `XMLType` columns directly or using `XMLType` methods `extract()` and `existsNode()`.
- Use Oracle Text operators to query the XML content. See "Indexing XMLType Columns" on page 4-32 and Chapter 10, "Full-Text Search Over XML".
- Use the XQuery language. See "Using XQuery with XMLType Data" on page 17-19

Pretty-Printing of Results

Pretty-printing of results has a performance cost in result size and processing time, because it requires building a full DOM and retaining or generating appropriate whitespace formatting information. For this reason, it is *not* the default behavior.

If you need pretty-printed output, invoke `XMLType` method `extract()` on the results. Avoid doing this, however, when working with large documents.

The following rules govern pretty-printing of results:

- SQL functions *never* pretty-print.
- `XMLType` methods (member functions) `extract()` and `transform()` *always* pretty-print.
- All other `XMLType` methods and all PL/SQL functions in packages `DBMS_XMLDOM` and `DBMS_XSLPROCESSOR` pretty-print if the data is stored *object-relationally*; otherwise (CLOB storage), they do *not* pretty-print.

Note: As mentioned in "Conventions" on page xlii, many examples in this book show results in pretty-printed form to promote readability, even when the results of the operation would not be pretty-printed in reality.

Searching XML Documents with XPath Expressions

The XPath language is a W3C Recommendation for navigating XML documents. XPath models an XML document as a tree of nodes. It provides a rich set of operations that walk this tree and apply predicates and node-test functions. Applying an XPath expression to an XML document can result in a set of nodes. For example, the expression `/PO/PONO` selects all `PONO` child elements under the `PO` root element of the document.

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

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

See Also: Appendix B, "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` selects the shipping address element of all purchase orders whose purchase-order number is 20 and whose purchase-order name is `PO_2`. Brackets are also used to denote 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`, Oracle XML DB additionally supports the function `ora:contains()`. This functions behaves 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 10, "Full-Text Search Over XML"

Selecting XML Data Using XMLType Methods

You can select `XMLType` data using PL/SQL, C, or Java. You can also use the `XMLType` methods `getClobVal()`, `getStringVal()`, `getNumberVal()`, and `getBlobVal(csid)` to retrieve XML data as a `CLOB`, `VARCHAR`, `NUMBER`, and `BLOB` value, respectively.

Example 4–1 Selecting XMLType Columns Using Method getClobVal()

This example shows how to select an `XMLType` column using method `getClobVal()`:

```
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/source/schemas/poSource/xsd/purchaseOr
der.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
...

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/source/schemas/poSource/xsd/purchaseOr
der.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
...
```

```
1 row selected.
```

> **Note:** In some circumstances, `XMLType` method `getClobVal()` returns a *temporary* `CLOB` value. If you call `getClobVal()` programmatically, you must explicitly *free* such a temporary `CLOB` value when finished with it. You can do this by calling PL/SQL method `DBMS_LOB.freeTemporary()` or its equivalent in Java or C (OCI). You can use method `DBMS_LOB.isTemporary()` to test whether a `CLOB` value is temporary.

Querying XMLType Data with SQL Functions

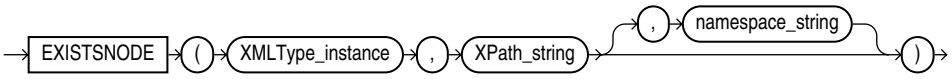
You can query `XMLType` data and extract portions of it using SQL functions `existsNode`, `extract`, and `extractValue`. These functions use a subset of the W3C XPath recommendation to navigate the document.

- EXISTSNODE SQL Function
- EXTRACT SQL Function
- EXTRACTVALUE SQL Function

EXISTSNODE SQL Function

Figure 4–1 describes the syntax for SQL function `existsNode`.

Figure 4–1 EXISTSNODE Syntax



SQL function `existsNode` checks whether the given XPath path references at least one XML element node or text node. If so, the function returns `1`; otherwise, it returns `0`. Optional parameter *`namespace_string`* is used to map the namespace prefixes specified in parameter *`XPath_string`* to the corresponding namespaces.

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.

If `existsNode` is called with an XPath expression that locates no nodes, the function returns `0`.

Function `existsNode` can be used in queries, and it can be used to create function-based indexes to speed up evaluation of queries.

> **Note:** When using SQL function `existsNode` in a query, always use it in the `WHERE` clause, never in the `SELECT` list.

Example 4–2 Using EXISTSNODE to Find a node

This example uses SQL function `existsNode` to select rows with `SpecialInstructions` set to `Expedite`.

```
SELECT OBJECT_VALUE
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[SpecialInstructions="Expedite"]')
```

```
        = 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 SQL function `existsNode` to speed up the execution. You can also create a `CTXXPATH` index to help speed up arbitrary XPath searching.

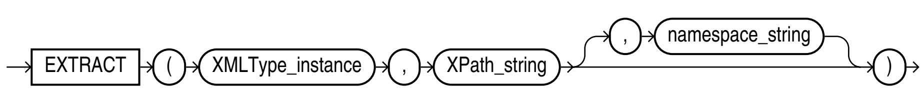
> **See Also:** "Creating CTXXPATH Indexes" on page 4-38

EXTRACT SQL Function

SQL function `extract` is similar to `existsNode`. It accepts a `VARCHAR2` XPath string that targets a node set and an optional `namespace` parameter. It returns an `XMLType` instance containing an XML fragment. The syntax is described in Figure 4–2:

```
extract(XMLType_instance IN XMLType,
        XPath_string IN VARCHAR2,
        namespace_string In VARCHAR2 := NULL) RETURN XMLType;
```

Figure 4–2 EXTRACT Syntax



Applying `extract` to an `XMLType` value extracts the node or a set of nodes from the document identified by the XPath expression. The XPath argument must target a node set. So, for example, XPath expression `/a/b/c[count('//d')=4]` can be used, but `count('//d')` cannot, because it returns a scalar value (number).

The extracted nodes can be element, attribute, or text nodes. If multiple text nodes are referenced in the XPath expression, the text nodes are collapsed into a single text node value. `Namespace` can be used to supply namespace information for prefixes in the XPath expression.

The `XMLType` instance returned from `extract` need not be a well-formed XML document. It can contain a set of nodes or simple scalar data. You can use `XMLType` methods `getStringVal()` and `getNumberVal()` to extract the scalar data.

For example, the XPath expression `/PurchaseOrder/Reference` identifies the `Reference` element inside the XML document shown previously. The expression

/PurchaseOrder/Reference/text(), on the other hand, refers to the text node of this Reference element.

> **Note:** A text node is considered an instance of XMLType. In other words, the following expression returns an XMLtype instance even though the instance may contain only text:
>
> ```
> extract(OBJECT_VALUE, '/PurchaseOrder/Reference/text()')
> ```
>
> You can use method getStringVal() to retrieve the text from the XMLType instance as a VARCHAR2 value.

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

For example:

- XPath /PurchaseOrder/Reference identifies the fragment <Reference> ... </Reference>
- XPath /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. If you have an XML document such as that in Example 4–3, then you can use:

- XPath expression //LineItem[1] to identify the first LineItem element.
- XPath expression //LineItem[2] to identify the second LineItem element.

Example 4–3 Purchase-Order XML Document

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

The result of SQL function `extract` is always an `XMLType` instance. If applying the XPath path produces an empty set, then `extract` returns a `NULL` value.

SQL function `extract` can be used in a number of ways. You can 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

Example 4–4 Using EXTRACT to Extract the Value of a Node

This example uses SQL function `extract` to retrieve the `Reference` children of `PurchaseOrder` nodes whose `SpecialInstructions` attribute has value `Expedite`.

```
SELECT extract(OBJECT_VALUE, '/PurchaseOrder/Reference') "REFERENCE"
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[SpecialInstructions="Expedite"]')
        = 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: SQL function `extractValue` and `XMLType` method `getStringVal()` differ in their treatment of entity encoding. Function `extractValue` *unescapes* any encoded entities; method `getStringVal()` returns the data with entity encoding intact.

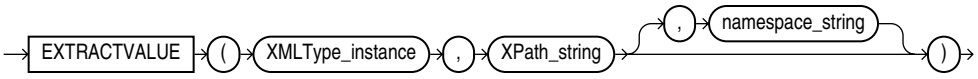
EXTRACTVALUE SQL Function

SQL function `extractValue` takes as parameters an `XMLType` instance and an XPath expression that targets a node set. It returns a scalar value corresponding to the result of the XPath evaluation on the `XMLType` instance.

- **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 query containing `extractValue` can be rewritten, 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`.

Figure 4–3 describes the `extractValue` syntax.

Figure 4–3 EXTRACTVALUE Syntax



SQL function `extractValue` attempts to determine the proper return type from the XML schema associated with 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. `CLOB` values are returned directly.

If a specific datatype is desired, a conversion function such as `to_char` or `to_date` can be applied to the result of `extractValue` or `extract.getStringVal()`. This can help maintain consistency between different queries regardless of whether the queries can be rewritten.

Use EXTRACTVALUE for Convenience

SQL function `extractValue` lets you extract the desired value more easily than `extract`; it is a convenience function. You can use it in place of `extract().getStringVal()` or `extract().getnumberval()`.

For example, you can replace `extract(x, 'path/text()').getStringVal()` with `extractValue(x, 'path/text()')`. If the node at `path` has only one child and that child is a text node, then you can leave the `text()` test off of the XPath argument: `extractValue(x, 'path')`. If not, an error is raised if you leave off `text()`.

SQL function `extractValue` has the same syntax as function `extract`.

EXTRACTVALUE Characteristics

SQL function `extractValue` has the following characteristics:

- It returns only a scalar value (`NUMBER`, `VARCHAR2`, and so on). It cannot return XML nodes or mixed content. An error is raised if extractValue cannot return a scalar value.
- By default, it returns a `VARCHAR2` value. If the length is greater than 4K, a run-time error is raised.
- If XML schema information is available at query compile time, then the datatype of the returned value is based on the XML schema information. For instance, if the XML schema information for the XPath

`/PurchaseOrder/LineItems/LineItem[1]/Part/@Quantity` indicates a number, then `extractValue` returns a `NUMBER`.

- If `extractValue` is applied to a SQL/XML view and the datatype of the column can be determined from the view definition at compile time, the appropriate type is returned.
- If the XPath argument identifies a node, then the node must have exactly one text child (or an errror is raised). The text child is returned. For example, this expression extracts the text child of the `Reference` node:

```
extractValue(xmlinstance, '/PurchaseOrder/Reference')
```

- The XPath argument must target a node set. So, for example, XPath expression `/a/b/c[count('//d')=4]` can be used, but `count('//d')` cannot, because it returns a scalar value (number).

Example 4–5 Extracting the Scalar Value of an XML Fragment Using extractValue

This query extracts the scalar value of the `Reference` node. This is in contrast to Example 4–4 where function `extract` is used to retrieve the `<Reference>` node itself.

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference') "REFERENCE"
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[SpecialInstructions="Expedite"]')
        = 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:** Function `extractValue` and `XMLType` method `getStringVal()` differ in their treatment of entity encoding. Function `extractValue` *unescapes* any encoded entities; method `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 Querying XMLType Using EXTRACTVALUE and EXISTSNODE

This example inserts two rows into the `purchaseorder` table, then queries data in those rows using `extractValue`.

```
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.

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 Querying Transient XMLType Data

This example uses a PL/SQL cursor to query XML data. A local `XMLType` instance is used to store transient 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 Extracting XML Data with EXTRACT, and Inserting It into a Table

This example extracts data from an XML purchase-order document, and inserts it into a SQL relational table using `extract`.

```
CREATE TABLE purchaseorder_table (reference           VARCHAR2(28) PRIMARY KEY,
                                  requestor           VARCHAR2(48),
                                  actions             XMLType,
                                  userid              VARCHAR2(32),
                                  costcenter          VARCHAR2(3),
                                  shiptoname          VARCHAR2(48),
                                  address             VARCHAR2(512),
                                  phone               VARCHAR2(32),
                                  rejectedby          VARCHAR2(32),
                                  daterejected        DATE,
                                  comments            VARCHAR2(2048),
                                  specialinstructions VARCHAR2(2048));

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, requestor, 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(li).extract('/LineItem/@ItemNumber').getNumberVal(),
         value(li).extract('/LineItem/Part/@Id').getNumberVal(),
         value(li).extract('/LineItem/Description/text()').getStringVal(),
         value(li).extract('/LineItem/Part/@Quantity').getNumberVal(),
         value(li).extract('/LineItem/Part/@UnitPrice').getNumberVal()
    FROM purchaseorder x, table(XMLSequence(OBJECT_VALUE.extract('/PurchaseOrder/LineItems/LineItem'))) li
    WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[Reference="EABEL-20021009123336251PDT"]') = 1;

3 rows created.

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.
```

Example 4–9 Extracting XML Data with EXTRACTVALUE, and Inserting It into a Table

This example extracts data from an XML purchase-order document, and inserts it into a relational table using SQL function `extractValue`.

```
CREATE OR REPLACE PROCEDURE insertPurchaseOrder(PurchaseOrder XMLType) AS reference VARCHAR2(28);
BEGIN
  INSERT INTO purchaseorder_table (reference, requestor, actions, userid, costcenter, shiptoname, address,
                                   phone, rejectedby, daterejected, comments, specialinstructions)
    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(li), '/LineItem/@ItemNumber'),
            extractValue(value(li), '/LineItem/Part/@Id'),
            extractValue(value(li), '/LineItem/Description'),
            extractValue(value(li), '/LineItem/Part/@Quantity'),
            extractValue(value(li), '/LineItem/Part/@UnitPrice')
      FROM table(XMLSequence(extract(PurchaseOrder, '/PurchaseOrder/LineItems/LineItem'))) li;
END;
/
Procedure created.

CALL insertPurchaseOrder(XMLType(bfilename('XMLDIR', 'purchaseOrder.xml'), nls_charset_id('AL32UTF8')));
```

```
Call completed.

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 Searching XML Data with XMLType Methods extract() and existsNode()

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. It uses `XMLType` methods `extract()` and `existsNode()` instead of SQL functions `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 Searching XML Data with EXTRACTVALUE

This example shows the query of Example 4–10 rewritten to use SQL function `extractValue`.

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Requestor') NAME, count(*)
  FROM purchaseorder
  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 SQL function `extract` to extract nodes identified by an XPath expression. An `XMLType` instance containing the XML fragment is returned by `extract`. 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()` method on the `XMLType` instance.

> **Note:** You cannot insert fragments into `XMLType` columns. You can use SQL function `sys_XMLGen` to convert a fragment into a well-formed document by adding an enclosing tag. See "Generating XML Using SQL Function SYS_XMLGEN" on page 16-48. You can, however, query further on the fragment using the various `XMLType` functions.

Example 4–12 Extracting Fragments From an XMLType Instance Using EXTRACT

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference') REFERENCE, count(*)
  FROM purchaseorder, table(XMLSequence(extract(OBJECT_VALUE,'//LineItem[Part/@Id="37429148327"]')))
  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-20021009123335771PDT                1
SKING-20021009123335560PDT                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 covers updating transient XML instances and XML data stored in tables. It details use of SQL functions `updateXML`, `insertChildXML`, `insertXMLbefore`, `appendChildXML`, and `deleteXML`.

Updating an Entire XML Document

For `CLOB`-based storage, an update effectively replaces the entire document. To update an entire XML document, use the SQL `UPDATE` statement. The right side of the `UPDATE` statement `SET` clause must be an `XMLType` instance. This can be created in any of the following ways:

- Use SQL functions or XML constructors that return an XML instance.

- Use the PL/SQL DOM APIs for `XMLType` that change and bind an existing XML instance.
- Use the Java DOM API that changes and binds an existing XML instance.

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

Example 4–13 Updating XMLType Using the UPDATE SQL Statement

This example updates an `XMLType` instance using a SQL `UPDATE` statement.

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference') REFERENCE,
       extractValue(value(li), '/LineItem/@ItemNumber') LINENO,
       extractValue(value(li), '/LineItem/Description') DESCRIPTION
  FROM purchaseorder, table(XMLSequence(extract(OBJECT_VALUE, '//LineItem'))) li
  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(li), '/LineItem/@ItemNumber') LINENO,
       extractValue(value(li), '/LineItem/Description') DESCRIPTION
  FROM purchaseorder, table(XMLSequence(extract(OBJECT_VALUE, '//LineItem'))) li
  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.
```

SQL Functions to Update XML Data

There are several SQL functions that you can use to update XML data incrementally—that is, to replace, insert, or delete XML data without replacing the entire surrounding XML document. This is also called **partial updating**. These SQL functions are described in the following sections:

- `updateXML` – Replace XML nodes of any kind. See "UPDATEXML SQL Function" on page 4-17.
- `insertChildXML` – Insert XML element or attribute nodes as children of a given element node. See "INSERTCHILDXML SQL Function" on page 4-25.

- `insertXMLbefore` – Insert XML nodes of any kind immediately before a given node (other than an attribute node). See "INSERTXMLBEFORE SQL Function" on page 4-28.
- `appendChildXML` – Insert XML nodes of any kind as the last child nodes of a given element node. See "APPENDCHILDXML SQL Function" on page 4-30.
- `deleteXML` – Delete XML nodes of any kind. See "DELETEXML SQL Function" on page 4-31.

Use functions `insertChildXML`, `insertXMLbefore`, and `appendChildXML` to insert XML data; use `deleteXML` to delete XML data; use `updateXML` to replace XML data. In particular, do *not* use function `updateXML` to insert or delete XML data by replacing a parent node in its entirety; this will work, but it is less efficient than using one of the other functions, which perform more localized updates.

These are all pure functions, without side effects. Each of these functions applies an XPath-expression argument to input XML data and returns a modified *copy* of the input XML data. You can then use that returned data with SQL operation `UPDATE` to modify database data.

Each of these functions can be used on XML documents that are either schema-based or non-schema-based. In the case of schema-based XML data, these SQL functions perform partial validation on the result, and, where appropriate, argument values are also checked for compatibility with the XML schema.

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

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

See Also: "Partial Validation" on page 3-31 for more information on partial validation against an XML schema

UPDATEXML SQL Function

SQL function `updateXML` replaces XML nodes of any kind. The XML document that is the target of the update can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned; the original data is unaffected. You can use that returned data with SQL operation `UPDATE` to modify database data.

Function `updateXML` has the following parameters (in order):

- ***`target-data`*** (`XMLType`) – The XML data containing the target node to replace.
- One or more *pairs* of *`xpath`* and *`replacement`* parameters:

  - ***xpath*** (VARCHAR2) – An XPath 1.0 expression that locates the nodes within *target-data* to replace; *each* targeted node is replaced by *replacement*. These can be nodes of any kind. If *xpath* matches an empty sequence of nodes, then no replacement is done; *target-data* is returned unchanged (and no error is raised).
  - ***replacement*** (XMLType or VARCHAR2) – The XML data that replaces the data targeted by *xpath*. The datatype of *replacement* must correspond to the data to be replaced. If *xpath* targets an element node for replacement, then the datatype must be XMLType; if *xpath* targets an attribute node or a text node, then it must be VARCHAR2. In the case of an attribute node, *replacement* is only the replacement *value* of the attribute (for example, 23), not the complete attribute node including the name (for example, my_attribute="23").

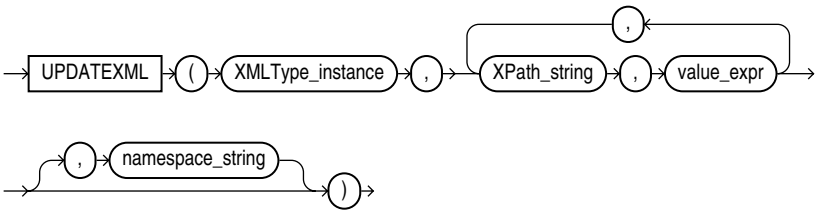
- ***namespace*** (VARCHAR2, *optional*) – The XML namespace for parameter *xpath*.

SQL function updateXML can be used to *replace* existing elements, attributes, and other nodes with new values. It is *not* an efficient way to insert new nodes or delete existing ones; you can only perform insertions and deletions with updateXML by using it to replace the entire node that is parent of the node to be inserted or deleted.

Function updateXML updates only the transient XML instance in memory. Use a SQL UPDATE statement to update data stored in tables.

Figure 4–4 illustrates the syntax.

Figure 4–4 UPDATEXML Syntax



Example 4–14 Updating XMLType Using UPDATE and UPDATEXML

This example uses updateXML on the right side of an UPDATE statement to update the XML document in a table instead of creating a new document. The entire document is updated, not just the part that is selected.

```
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-2002100912333601PDT"]') = 1;

ACTION
--------------------------------
<Action>
  <User>SKING</User>
</Action>

1 row selected.
```

Example 4–15 Updating Multiple Text Nodes and Attribute Values Using UPDATEXML

This example updates multiple nodes using SQL function `updateXML`.

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Requestor') NAME,
       extract(OBJECT_VALUE, '/PurchaseOrder/LineItems') LINEITEMS
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 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-2002100912333601PDT"]') = 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-2002100912333601PDT"]') = 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 Updating Selected Nodes Within a Collection Using UPDATEXML

This example uses SQL function `updateXML` to update selected nodes within a collection.

```
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/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`, the attributes and children of the element are removed, and the element becomes empty. The type and namespace properties of the element are retained. See Example 4–17.

If you update an *attribute* value to `NULL`, the value appears as the empty string. See Example 4–17.

If you update the *text* node of an element to `NULL`, the content (text) of the element is removed; the element itself remains, but is empty. See Example 4–18.

Example 4–17 NULL Updates With UPDATEXML – Element and Attribute

This example updates all of the following to `NULL`:

- The `Description` element and the `Quantity` attribute of the `LineItem` element whose `Part` element has attribute `Id` value 715515009058.
- The `LineItem` element whose `Description` element has the content (text) "The Unbearable Lightness Of Being".

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Requestor') NAME,
       extract(OBJECT_VALUE, '/PurchaseOrder/LineItems') LINEITEMS
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]') = 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-2002100912333601PDT"]') = 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.
```

Example 4–18 updates the text node of a `Part` element whose `Description` attribute has value "`A Night to Remember`" to `NULL`.

Example 4–18 NULL Updates With UPDATEXML – Text Node

The XML data for this example corresponds to a different, revised purchase-order XML schema – see "Revised Purchase-Order XML Schema" on page 8-4. In that XML schema, `Description` is an *attribute* of the `Part` element, not a sibling element.

```
SELECT extractValue(OBJECT_VALUE,
                    '/PurchaseOrder/LineItems/LineItem/Part[@Description="A Night to Remember"]') PART
  FROM purchaseorder
  WHERE existsNode(object_value,'/PurchaseOrder[@Reference="SBELL-20030309123336O1PDT"]') = 1;

PART
----
<Part Description="A Night to Remember" UnitCost="39.95">715515009058</Part>

UPDATE purchaseorder
  SET OBJECT_VALUE =
        updateXML(OBJECT_VALUE,
                  '/PurchaseOrder/LineItems/LineItem/Part[@Description="A Night to Remember"]/text()', NULL)
  WHERE existsNode(object_value,'/PurchaseOrder[@Reference="SBELL-20030309123336O1PDT"]') = 1;


SELECT extractValue(OBJECT_VALUE,
                    '/PurchaseOrder/LineItems/LineItem/Part[@Description="A Night to Remember"]') PART
  FROM purchaseorder
  WHERE existsNode(object_value,'/PurchaseOrder[@Reference="SBELL-20030309123336O1PDT"]') = 1;


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

See Also: Example 6–2, Example 6–3, Example 3–34, and Example 3–34 for examples of rewriting `updateXML` expressions

Updating the Same XML Node More Than Once

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

Preserving DOM Fidelity When Using UPDATEXML

Here are some guidelines for preserving DOM fidelity when using SQL function `updateXML`:

When DOM Fidelity is Preserved When you update an element to `NULL`, you make that element appear *empty* in its parent, such as in `<myElem/>`.

When you update a text node inside an element to `NULL`, you *remove* that text node from the element.

When you update an attribute node to `NULL`, you make the value of the attribute become the *empty* string, for example, `myAttr=""`.

When DOM Fidelity is Not Preserved When you update a `complexType` element to `NULL`, you make the element appear *empty* in its parent, for example, `<myElem/>`.

When you update a SQL-inlined `simpleType` element to `NULL`, you make the element *disappear* from its parent.

When you update a text node to `NULL`, you are doing the same thing as setting the parent `simpleType` element to `NULL`. Furthermore, text nodes can appear only inside `simpleType` elements when DOM fidelity is not preserved, since there is no positional descriptor with which to store mixed content.

When you update an attribute node to `NULL`, you *remove* the attribute from the element.

Determining Whether DOM Fidelity is Preserved You can determine whether or not DOM fidelity is preserved for particular parts of a given `XMLType` in a given XML schema by querying the schema metadata for attribute `maintainDOM`.

See Also:

- "Querying a Registered XML Schema to Obtain Annotations" on page 5-25 for an example of querying a schema to retrieve DOM fidelity values
- "DOM Fidelity" on page 5-14

Optimization of SQL Functions that Modify XML

In most cases, the SQL functions that modify XML data (`updateXML`, `insertChildXML`, `insertXMLbefore`, `appendChildXML`, and `deleteXML`) materialize a copy of the entire input XML document in memory, then update the copy. However, functions `updateXML`, `insertChildXML`, and `deleteXML` are optimized for SQL `UPDATE` operations on XML schema-based, object-relationally stored `XMLType` tables and columns. If particular conditions are met, then the function call is rewritten to update the object-relational columns directly with the values.

See Also: Chapter 3, "Using Oracle XML DB" and Chapter 6, "XPath Rewrite" for information on the conditions for XPath rewrite.

For example, the XPath argument to `updateXML` in Example 4–19 is processed by Oracle XML DB and rewritten into the equivalent object relational SQL statement shown in Example 4–20.

Example 4–19 XPath Expressions in UPDATEXML Expression

```
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 4–20 Object Relational Equivalent of UPDATEXML Expression

```
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 with SQL Functions that Modify XML

You can use the SQL functions that modify XML data (`updateXML`, `insertChildXML`, `insertXMLbefore`, `appendChildXML`, and `deleteXML`) to create new views of XML data.

Example 4–21 Creating Views Using UPDATEXML

This example creates a view of the `purchaseorder` table using SQL function `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/source/schemas/poSource/xsd/purchaseOrder.xsd">
  <Reference>DAUSTIN-20021009123335811PDT</Reference>
  <Actions/>
  <Reject/>
  <Requestor>David L. Austin</Requestor>
  <User>DAUSTIN</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions/>
  <SpecialInstructions>Courier</SpecialInstructions>
  <LineItems/>
</PurchaseOrder>

1 row selected.
```

INSERTCHILDXML SQL Function

SQL function `insertChildXML` inserts new children (one or more elements of the same type or a single attribute) under parent XML elements. The XML document that is the target of the insertion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned; the original data is unaffected. You can use that returned data with SQL operation `UPDATE` to modify database data.

Function `insertChildXML` has the following parameters (in order):

- ***target-data*** (`XMLType`) – The XML data containing the target parent element.
- ***parent-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates the parent elements within *target-data*; *child-data* is inserted under *each* parent element.

  If *parent-xpath* matches an empty sequence of element nodes, then no insertion is done; *target-data* is returned unchanged (and no error is raised). If *parent-xpath* does not match a sequence of element nodes (in particular, if *parent-xpath* matches one or more *attribute* or *text* nodes), then an error is raised.

- ***child-name*** (VARCHAR2) – The name of the child elements or attribute to insert. An attribute name is distinguished from an element name by having an at-sign (@) prefix as part of *child-name*, for example, `@my_attribute` versus `my_element`. (The at-sign is not part of the actual attribute name, but serves in the argument to indicate that *child-name* refers to an attribute.)
- ***child-data*** (XMLType or VARCHAR2) – The child XML data to insert:
  - If one or more *elements* are being inserted, then this is of datatype XMLType, and it contains element nodes. *Each* of the top-level element nodes in *child-data* must have the same name (tag) as *child-name* (or else an error is raised).
  - If an *attribute* is being inserted, then this is of datatype VARCHAR2, and it represents the (scalar) attribute value. If an attribute of the same name already exists at the insertion location, then an error is raised.
- ***namespace*** (VARCHAR2, *optional*) – The XML namespace for parameters *parent-xpath* and *child-data*.

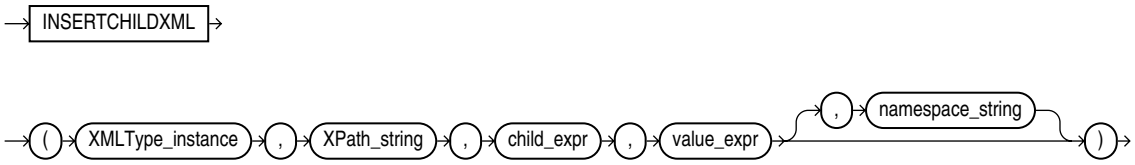
XML data *child-data* is inserted as one or more child elements, or a single child attribute, under *each* of the parent elements located at *parent-xpath*; the result is returned.

In order of decreasing precedence, function insertChildXML has the following behavior for NULL arguments:

- If *child-name* is NULL, then an error is raised.
- If *target-data* or *parent-xpath* is NULL, then NULL is returned.
- If *child-data* is NULL, then:
  - If *child-name* names an element, then no insertion is done; *target-data* is returned unchanged.
  - If *child-name* names an attribute, then an empty attribute value is inserted, for example, `my_attribute = ""`.

Figure 4–5 shows the syntax.

Figure 4–5 INSERTCHILDXML Syntax



If *target-data* is XML *schema-based*, then the schema is consulted to determine the insertion positions. For example, if the schema constrains child elements named *child-name* to be the first child elements of a *parent-xpath*, then the insertion takes this into account. Similarly, if the *child-name* or *child-data* argument is inappropriate for an associated schema, then an error is raised.

If the parent element does *not* yet have a child corresponding in name and kind to *child-name* (and if such a child is permitted by the associated XML schema, if any), then *child-data* is inserted as new child elements, or a new attribute value, named *child-name*.

If the parent element already has a child *attribute* named *child-name* (without the at-sign), then an error is raised. If the parent element already has a child *element* named

child-name (and if more than one child element is permitted by the associated XML schema, if any), then *child-data* is inserted so that its elements become the *last* child elements named *child-name*.

Example 4–22 Inserting a LineItem Element into a LineItems Element

```
SELECT extract(OBJECT_VALUE,
               '/PurchaseOrder/LineItems/LineItem[@ItemNumber="222"]')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[@ITEMNUMBER="222"]')
--------------------------------------------------------------------------------

1 row selected.

UPDATE purchaseorder
  SET OBJECT_VALUE =
        insertChildXML(OBJECT_VALUE,
                       '/PurchaseOrder/LineItems',
                       'LineItem',
                       XMLType('<LineItem ItemNumber="222">
                                  <Description>The Harder They Come</Description>
                                  <Part Id="953562951413"
                                        UnitPrice="22.95"
                                        Quantity="1"/>
                                </LineItem>'))
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

1 row updated.

SELECT extract(OBJECT_VALUE,
               '/PurchaseOrder/LineItems/LineItem[@ItemNumber="222"]')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[@ITEMNUMBER="222"]')
--------------------------------------------------------------------------------
<LineItem ItemNumber="222">
  <Description>The Harder They Come</Description>
  <Part Id="953562951413" UnitPrice="22.95" Quantity="1"/>
</LineItem>

1 row selected.
```

If XML data to be updated is XML schema-based and it refers to a namespace, then the data to be inserted must also refer to the same namespace; otherwise, an error will be raised because the inserted data does not conform to the XML schema. For example, if the data in Example 4–22 used the namespace `films.xsd`, then the `UPDATE` statement would need to be as shown in Example 4–23.

Example 4–23 Inserting an Element that Uses a Namespace

This example is the same as Example 4–22, except that the `LineItem` element to be inserted refers to a namespace. This assumes that the XML schema requires a namespace for this element.

Note that this use of namespaces is different from the use of a namespace *argument* to function `insertChildXML` – namespaces supplied in that optional argument apply only to the XPath argument, not to the content to be inserted.

```
UPDATE purchaseorder
  SET OBJECT_VALUE =
        insertChildXML(OBJECT_VALUE,
                       '/PurchaseOrder/LineItems',
                       'LineItem',
                       XMLType('<LineItem xmlns="films.xsd" ItemNumber="222">
                                  <Description>The Harder They Come</Description>
                                  <Part Id="953562951413"
                                        UnitPrice="22.95"
                                        Quantity="1"/>
                                </LineItem>'))
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

1 row updated.
```

INSERTXMLBEFORE SQL Function

SQL function `insertXMLbefore` inserts one or more nodes of any kind immediately before a target node that is not an attribute node. The XML document that is the target of the insertion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned; the original data is unaffected. You can use that returned data with SQL operation `UPDATE` to modify database data.

Function `insertXMLbefore` has the following parameters (in order):

- ***target-data*** (`XMLType`) – The XML data that is the target of the insertion.
- ***successor-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates zero or more nodes in *`target-data`* of any kind *except* attribute nodes. *`XML-data`* is inserted immediately before *each* of these nodes; that is, the nodes in *`XML-data`* become preceding siblings of each of the*`successor-xpath`* nodes.

  If *`successor-xpath`* matches an empty sequence of nodes, then no insertion is done; *`target-data`* is returned unchanged (and no error is raised). If *`successor-xpath`* does not match a sequence of nodes that are not attribute nodes, then an error is raised.
- ***XML-data*** (`XMLType`) – The XML data to be inserted: one or more nodes of *any kind*. The order of the nodes is preserved after the insertion.
- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameter *`successor-xpath`*.

The *`XML-data`* nodes are inserted immediately before *each* of the nonattribute nodes located at *`successor-xpath`*; the result is returned.

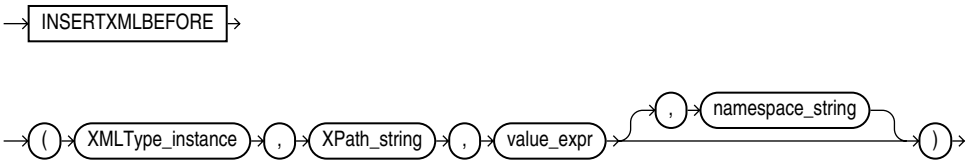
Function `insertXMLbefore` has the following behavior for `NULL` arguments:

- If *`target-data`* or *`parent-xpath`* is `NULL`, then `NULL` is returned.

- Otherwise, if *child-data* is NULL, then no insertion is done; *target-data* is returned unchanged.

Figure 4–6 shows the syntax.

Figure 4–6 INSERTXMLBEFORE Syntax



Example 4–24 Inserting a LineItem Element Before the First LineItem ELement

```
SELECT extract(OBJECT_VALUE, '/PurchaseOrder/LineItems/LineItem[1]')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[1]')
------------------------------------------------------------
<LineItem ItemNumber="1">
  <Description>Salesman</Description>
  <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
</LineItem>

1 row selected.

UPDATE purchaseorder
  SET OBJECT_VALUE =
      insertXMLbefore(OBJECT_VALUE,
                      '/PurchaseOrder/LineItems/LineItem[1]',
                      XMLType('<LineItem ItemNumber="314">
                                 <Description>Brazil</Description>
                                 <Part Id="314159265359"
                                       UnitPrice="69.95"
                                       Quantity="2"/>
                               </LineItem>'))
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

SELECT extract(OBJECT_VALUE, '/PurchaseOrder/LineItems/LineItem[position() <= 2]')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[POSITION()<=2]')
------------------------------------------------------------------------
<LineItem ItemNumber="314">
  <Description>Brazil</Description>
  <Part Id="314159265359" UnitPrice="69.95" Quantity="2"/>
</LineItem>
<LineItem ItemNumber="1">
  <Description>Salesman</Description>
  <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
```

```
</LineItem>

1 row selected.
```

APPENDCHILDXML SQL Function

SQL function `appendChildXML` inserts one or more nodes of any kind as the last children of a given element node. The XML document that is the target of the insertion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned; the original data is unaffected. You can use that returned data with SQL operation `UPDATE` to modify database data.

Function `appendChildXML` has the following parameters (in order):

- ***target-data*** (`XMLType`)– The XML data containing the target parent element.
- ***parent-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates zero or more *element* nodes in *`target-data`* that are the targets of the insertion operation; *`child-data`* is inserted as the last child or children of *each* of these parent elements.

  If *`parent-xpath`* matches an empty sequence of element nodes, then no insertion is done; *`target-data`* is returned unchanged (and no error is raised). If *`parent-xpath`* does not match a sequence of element nodes (in particular, if *`parent-xpath`* matches one or more *attribute* or *text* nodes), then an error is raised.
- ***child-data*** (`XMLType`) – Child data to be inserted: one or more nodes of *any kind*. The order of the nodes is preserved after the insertion.
- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameter *`parent-xpath`*.

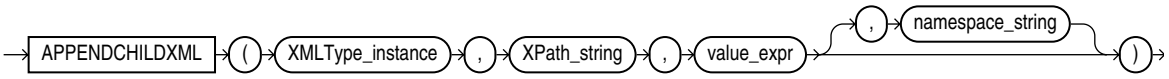
XML data *`child-data`* is inserted as the last child or children of *each* of the element nodes indicated by *`parent-xpath`*; the result is returned.

Function `appendChildXML` has the following behavior for `NULL` arguments:

- If *`target-data`* or *`parent-xpath`* is `NULL`, then `NULL` is returned.
- Otherwise, if *`child-data`* is `NULL`, then no insertion is done; *`target-data`* is returned unchanged.

Figure 4–6 shows the syntax.

Figure 4–7 APPENDCHILDXML Syntax



Example 4–25 Inserting a Date Element as the Last Child of an Action Element

```
SELECT extract(OBJECT_VALUE, '/PurchaseOrder/Actions/Action[1]')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/ACTIONS/ACTION[1]')
--------------------------------------------------------
```

```
<Action>
  <User>KPARTNER</User>
</Action>

1 row selected.

UPDATE purchaseorder
  SET OBJECT_VALUE =
      appendChildXML(OBJECT_VALUE,
                     '/PurchaseOrder/Actions/Action[1]',
                     XMLType('<Date>2002-11-04</Date>'))
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

SELECT extract(OBJECT_VALUE, '/PurchaseOrder/Actions/Action[1]')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/ACTIONS/ACTION[1]')
--------------------------------------------------------
<Action>
  <User>KPARTNER</User>
  <Date>2002-11-04</Date>
</Action>

1 row selected.
```

DELETEXML SQL Function

SQL function `deleteXML` deletes XML nodes of any kind. The XML document that is the target of the deletion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned; the original data is unaffected. You can use that returned data with SQL operation `UPDATE` to modify database data.

Function `deleteXML` has the following parameters (in order):

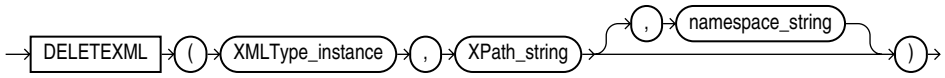
- ***target-data*** (`XMLType`) – The XML data containing the target nodes (to be deleted).
- ***xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates zero or more nodes in *`target-data`* that are the targets of the deletion operation; *each* of these nodes is deleted.

  If *`xpath`* matches an empty sequence of nodes, then no deletion is done; *`target-data`* is returned unchanged (and no error is raised). If *`xpath`* matches the top-level element node, then an error is raised.
- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameter *`xpath`*.

The XML nodes located at *`xpath`* are deleted from *`target-data`*; the result is returned. Function `deleteXML` returns `NULL` if *`target-data`* or *`xpath`* is `NULL`.

Figure 4–6 shows the syntax.

Figure 4–8 DELETEXML Syntax



Example 4–26 Deleting LineItem Element Number 222

```
SELECT extract(OBJECT_VALUE,
               '/PurchaseOrder/LineItems/LineItem[@ItemNumber="222"]')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;
EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[@ITEMNUMBER="222"]')
--------------------------------------------------------------------------------
<LineItem ItemNumber="222">
  <Description>The Harder They Come</Description>
  <Part Id="953562951413" UnitPrice="22.95" Quantity="1"/>
</LineItem>

1 row selected.

UPDATE purchaseorder
  SET OBJECT_VALUE =
      deleteXML(OBJECT_VALUE,
                '/PurchaseOrder/LineItems/LineItem[@ItemNumber="222"]')
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;

SELECT extract(OBJECT_VALUE,
               '/PurchaseOrder/LineItems/LineItem[@ItemNumber="222"]')
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]')
        = 1;
EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LINEITEMS/LINEITEM[@ITEMNUMBER="222"]')
--------------------------------------------------------------------------------

1 row selected.
```

Indexing XMLType Columns

Chapter 3 provides a basic introduction to creating indexes on XML documents that have been stored using the structured storage option. It demonstrates 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 the following:

- 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, Oracle XML DB attempts to rewrite the XPath expressions provided to SQL function `extractValue` into `CREATE INDEX` statements that operate directly on the underlying objects.

For instance, given an index created as shown in Example 4–27, XPath rewrite will rewrite the index, resulting in the create index statement shown in Example 4–28 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 *occurs only once* in the XML Document.

Example 4–27 Using EXTRACTVALUE to Create an Index on a Singleton Element or Attribute

```
CREATE INDEX ipurchaseorder_rejectedby
  ON purchaseorder (extractValue(OBJECT_VALUE, '/PurchaseOrder/Reject/User'));

Index created.
```

Example 4–28 XPath Rewrite of an Index on a Singleton Element or Attribute

```
CREATE INDEX ipurchaseorder_rejectedby
  ON purchaseorder p (p."XMLDATA"."rejection"."rejected_by");

Index created.
```

Creating B-Tree Indexes on the Contents of a Collection

You often need to create an index over a collection: nodes that occur more than once in the target document.

For instance, suppose that you want 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 4–29. However, when the element or attribute being indexed occurs multiple times in the document, the `CREATE INDEX` operation fails, because `extractValue()` is only allowed to return a single value for each row it processes.

Example 4–29 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 2:
ORA-19025: EXTRACTVALUE returns value of only one node
```

You can instead create an index by replacing `extractValue` with function `extract` and method `getStringVal()`, as shown in Example 4–30.

Example 4–30 Using 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 you might expect. The index is created by invoking SQL function

extract and XMLType method getStringVal() for each row in the table, and then indexing the result against the rowid of the row.

The problem with this technique is that function extract can only return multiple nodes. The result of function extract is a single XMLType XML fragment containing all the matching nodes. The result of invoking getStringVal() on an XMLType instance that contains a fragment is a *concatenation* of the nodes in the fragment:

```
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.
```

What is indexed for this row is the concatenation of the 3 UPC codes, not, as intended, each of the individual UPC codes. In general, care should be taken when creating an index using SQL function extract. It is unlikely that this index will be useful.

As is shown in Chapter 3 for schema-based XMLType values, the best way to resolve this issue is to 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 that generated by XPath rewrite.

Creating Function-Based Indexes on XMLType Tables and Columns

The index created in Example 4–30 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 the case of Example 4–30, the results of the functions were not useful, and consequently the index itself was not useful.

A function-based index can be useful when the XML content is not managed using structured storage. In this case, instead of the CREATE INDEX statement being rewritten, the index is created by invoking the function on the XML content and indexing the result.

Example 4–31 Creating a Function-Based Index on a CLOB-based XMLType()

The table created in this example uses CLOB storage rather than structured storage. The CREATE INDEX statement creates a function-based index on the value of the text node of the Reference element. This index enforces the uniqueness constraint on the text-node value.

```
CREATE TABLE purchaseorder_clob OF XMLType
  XMLTYPE STORE AS CLOB
  ELEMENT "http://localhost:8080/source/schemas/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 (QUINE.IPURCHASEORDER_REFERENCE) violated
```

The optimizer only considers 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 4–32, which find a `PurchaseOrder`-based value of the text node associated with the `Reference` element. The first query, which uses function `existsNode` to locate the document, does *not* use the index, while the second query, which uses function `extractValue`, does use the index. This is because the index was created using `extractValue`.

Example 4–32 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.

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.

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
-----
   - 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 rewrite 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 SQL function `existsNode`. Function `existsNode` 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 rewrite 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 function `existsNode`.

In Example 4–33, an index is created that can be used to speed up a query that searches for instances of a rejected purchase order by looking for the presence of a text node of element `/PurchaseOrder/Reject/User`.

Since the index is function-based, it can be used with structured and unstructured schema-based `XMLType` tables and columns, and with non-schema-based `XMLType` tables and columns.

Example 4–33 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-20020912130000000PDT

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.

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. This is a general-purpose XPath-based index that can be used to improve the performance of any search based on SQL function `existsNode`. A `CTXXPath` index has the following *advantages*:

- You do not need prior knowledge of the XPath expressions that will be searched on.
- You can use it with structured and unstructured schema-based `XMLType` tables and columns, and with non-schema-based `XMLType` tables and columns.
- You can use it to 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 rewrite the XPath expression supplied to SQL function `existsNode` into `HASPATH` and `INPATH` operators, which 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 Oracle 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.

> **See Also:** "EXISTSNODE SQL Function" on page 4-5 for more information on using `existsNode`.

CTXXPATH Indexing Features

CTXXPATH indexing has the following characteristics:

- It can be used only to speed up processing of function existsNode. It acts as a primary filter for function existsNode. In other words, it provides a superset of the results that existsNode provides.
- It works only for queries where the XPath expressions that identify the required documents are supplied using an existsNode expression that appears in the WHERE clause of the SQL statement being executed.
- It handles only a limited set of XPath expressions. See "Choosing the Right Plan: Using CTXXPATH Index in EXISTSNODE Processing" on page 4-41 for the list of XPath expressions not supported by the index.
- It supports only the STORAGE preference parameter. See "Creating CTXXPATH Storage Preferences With CTX_DDL. Statements" on page 4-39.
- It follows the transactional semantics of function existsNode, returning unindexed rows as part of its result set, in order to guarantee that it returns a superset of the valid results. This is despite the asynchronous nature of Data Manipulation Language (DML) operations such as updating and deleting. (You must use a special command to synchronize DML operations, in a fashion similar to that of the Oracle Text index.)

Creating CTXXPATH Indexes

You create CTXXPATH indexes the same way you create Oracle Text indexes, using the following 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 4–34 Using CTXXPATH Index and EXISTSNODE for XPath Searching

This example demonstrates how to create a CTXXPATH index for XPath searching. A CTXXPATH index is a *global* index – the EXPLAIN PLANs in this example show that the index is used for each of two very different queries.

```
CREATE INDEX purchaseorder_clob_xpath ON purchaseorder_clob (OBJECT_VALUE)
  INDEXTYPE IS CTXSYS.ctxxpath;

EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference') FROM purchaseorder_clob
    WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[SpecialInstructions="Air Mail"]') = 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('E26E03C077B81004E0340003BA0BF841',3626,"PURCHASEORDER_CLOB
              "."XMLDATA"),'/PurchaseOrder[SpecialInstructions="Air Mail"]')=1)
   2 - access("CTXSYS"."XPCONTAINS"(SYS_MAKEXML('E26E03C077B81004E0340003BA0BF841',3626,"XMLDATA
              "),'HASPATH(/2cc2504b[6b3fb29d="17a03105"]) ')>0)

Note
-----
   - dynamic sampling used for this statement

21 rows selected.

EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference') FROM purchaseorder_clob
    WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder/LineItems/LineItem[Description="The Rock"]') = 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('E26E03C077B81004E0340003BA0BF841',3626,"PURCHASEORDER_CLOB
              "."XMLDATA"),'/PurchaseOrder/LineItems/LineItem[Description="The Rock"]')=1)
   2 - access("CTXSYS"."XPCONTAINS"(SYS_MAKEXML('E26E03C077B81004E0340003BA0BF841',3626,"XMLDATA
              "),'HASPATH(/2cc2504b/52304c67/69182640[2ea8698d="0fb68600"]) ')>0)

Note
-----
   - dynamic sampling used for this statement

21 rows selected.
```

Creating CTXXPATH Storage Preferences With CTX_DDL. Statements

The only preference allowed in CTXXPATH indexing is the STORAGE preference. Create the STORAGE preference the same way you would for an Oracle Text index, as shown in Example 4–35.

> **Note:** You must be granted execute privileges on the CTXSYS.CTX_DLL package in order to create storage preferences.

Example 4–35 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 4–36 Synchronizing the CTXXPATH Index

This example shows how to synchronize DML operations using the `sync_index` procedure in the `CTX_DDL` package.

```
CALL CTX_DDL.sync_index('purchaseorder_clob_xpath');

Call completed.
```

Example 4–37 Optimizing the CTXXPATH Index

This example shows how to optimize the `CTXXPATH` index using the `optimize_index` procedure in the `CTX_DDL` package.

```
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 an asterisk (`*`), double slashes (`//`), or double periods (`..`); for example, `/A/*/@attr`, `/A//@attr`, or `/A//../@attr`
  - A period (`.`) or an asterisk (`*`) at the end of a path expression
  - A predicate following a period (`.`) or an asterisk (`*`)
  - String literal equalities are supported with the following restrictions:
    * The left side must be a path – period (`.`) by itself is not allowed; for example, `.="dog"` is not allowed
    * The right side must be a literal
  - Anything not expressible by abbreviated syntax is unsupported

For the cost-based optimizer to better estimate the costs and selectivities for function `existsNode`, 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 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 4–38, a CTXXPATH index is created on table purchaseorder. The existsNode expression specified in the WHERE clause is rewritten into an expression that checks if the underlying object-relational column is not NULL. This is in accordance with XPath rewrite rules. The optimizer hint /*+ NO_XML_QUERY_REWRITE */ causes XPath rewrite to be turned *off* for the query, so the existsNode expression is left unchanged.

Example 4–38 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.

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.

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 Whether 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 queries with `existsNode` expressions on non-schema-based `XMLType` tables and columns when the data is stored as a `CLOB` value. `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.

If the storage is `CLOB`-based:

- Check the query mix to see if a significant fraction involves the same set of XPath expressions. If so, then 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.

If the storage is object-relational:

- Check the query mix to see if a significant fraction involves XPath expressions that can be rewritten. Chapter 6, "XPath Rewrite" describes the XPath expressions that can potentially get rewritten. The set of XPath expressions that are actually rewritten depends on the type of XPath expression 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 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 queries that are not rewritten, 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: Use indexes for OLTP environments that are *seldom updated*. Maintaining `CTXXPATH` and function-based indexes when there are frequent updates adds 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, Oracle Database 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` SQL function for full-text search over XML.

To create an Oracle Text index, use `CREATE INDEX`, specifying the `INDEXTYPE`.

Example 4–39 Creating an Oracle Text Index

```
CREATE INDEX ipurchaseordertextindex ON purchaseorder (OBJECT_VALUE)
  INDEXTYPE IS CTXSYS.CONTEXT;

Index created.
```

You can also perform Oracle Text operations such as `contains` and `score` on `XMLType` columns.

Example 4–40 Searching XML Data Using CONTAINS

This example shows an Oracle Text search 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 10, "Full-Text Search Over XML" for more information on using Oracle Text operations with Oracle XML DB.

5

XML Schema Storage and Query: Basic

The XML Schema Recommendation was created by the World Wide Web Consortium (W3C) to describe the content and structure of XML documents in XML. It includes the full capabilities of Document Type Definitions (DTDs) so that existing DTDs can be converted to XML Schema. XML schemas have additional capabilities compared to DTDs.

This chapter provides basic information about using XML Schema with Oracle XML DB. It explains how to do all of the following:

- Register, update, and delete an XML schema
- Create storage structures for XML schema-based data
- Map XML `simpleType` and `complexType` to SQL storage types

See Also:

- Chapter 7, "XML Schema Storage and Query: Advanced" for more advanced information on using XML Schema with Oracle XML DB
- Chapter 6, "XPath Rewrite" for information on the optimization of XPath expressions in Oracle XML DB
- Appendix A, "XML Schema Primer" for an introduction to XML Schema

This chapter contains these topics:

Overview of XML Schema and Oracle XML DB

XML Schema is a schema definition language written in XML. It can be used to describe the structure and 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 instance definition as an **XML schema**.

Example 5–1 XML Schema Instance purchaseOrder.xsd

The following is an XML schema that declares a `complexType` called `purchaseOrderType` and a global element `PurchaseOrder` of this type. This is the same schema as Example 3–7, "Purchase-Order XML Schema, purchaseOrder.xsd", with the exception of the lines in **`bold`** here, which are additional. For brevity, part of the schema is omitted here (marked **`...`**).

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

    ...

    <xs:simpleType name="DescriptionType">
      <xs:restriction base="xs:string">
        <xs:minLength value="1"/>
        <xs:maxLength value="256"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="NotesType">
      <xs:restriction base="xs:string">
        <xs:minLength value="1"/>
        <xs:maxLength value="32767"/>
      </xs:restriction>
```

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

Example 5–2 purchaseOrder.XML: Document That Conforms to 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
...
tiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla
 pariatur?&quot;

1914 translation by H. Rackham

&quot;But I must explain to you how all this mistaken idea of denouncing
```

```
 pleasure and praising pain was born and I will give you a c
...
o avoids a pain that produces no resultant pleasure?&quot;

Section 1.10.33 of &quot;de Finibus Bonorum et Malorum&quot;, written by Cicero
in 45 BC

&quot;At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis
 praesentium voluptatum deleniti atque corrupti quos
...
 delectus, ut aut reiciendis voluptatibus maiores alias
consequatur aut perferendis doloribus asperiores repellat.&quot;

1914 translation by H. Rackham

&quot;On the other hand, we denounce with righteous indignation and dislike men
 who are so beguiled and demoralized by the charms of
...
secure other greater pleasures, or else he endures pains to avoid worse
 pains.&quot;
  </Notes>
</po:PurchaseOrder>
```

Note: The URL used here is simply a name that uniquely identifies the registered XML schema within the database: `http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd`. This 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 Schemas 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: `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 parameter `schemaurl` of PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema`.

Oracle XML DB provides XML Schema support for the following tasks:

- Registering any W3C-compliant XML schemas.
- Validating your XML documents against registered XML schema definitions.
- Registering local and global XML schemas.
- Generating XML schemas 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(S)/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 schemas 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 most is not 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 XML schemas 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 schemas in Oracle XML DB, when inserting and storing XML instances using protocols such as FTP or HTTP(S), 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 schemas can be useful in predicting optimum storage, fidelity, and access.

DTD Support in Oracle XML DB

A **DTD** is a set of rules that define the allowable structure of an XML document. DTDs are text files that derive their format from SGML and can be associated with an XML document either by using the `DOCTYPE` element or by using an external file through a `DOCTYPE` reference. In addition to supporting XML Schema, which provides a structured mapping to object- relational storage, Oracle XML DB also supports DTD specifications in XML instance documents. Though DTDs are not used to derive the mapping, XML processors can still access and interpret the DTDs.

Inline DTD Definitions

When an XML instance document has an inline DTD definition, it is used during document parsing. Any DTD validations and entity declaration handling is done at this point. However, once parsed, the entity references are replaced with actual values and the original entity reference is lost.

External DTD Definitions

Oracle XML DB also supports external DTD definitions if they are stored in Oracle XML DB Repository. Applications needing to process an XML document containing an external DTD definition such as `/public/flights.dtd`, must first ensure that the DTD document is stored in Oracle XML DB at path `/public/flights.xsd`.

> **See Also:** Chapter 20, "Accessing Oracle XML DB Repository Data"

Managing XML Schemas with DBMS_XMLSCHEMA

Before an XML schema can be used by Oracle XML DB, it must be registered with Oracle Database. You register an XML schema using the PL/SQL package `DBMS_XMLSCHEMA`.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference*

Some of the main `DBMS_XMLSCHEMA` procedures are these:

- `registerSchema` – register an XML schema with Oracle Database
- `deleteSchema` – delete a previously registered XML schema.

- `copyEvolve` – update a registered XML schema; see Chapter 8, "XML Schema Evolution".

Registering an XML Schema

The main arguments to procedure `DBMS_XMLSCHEMA.registerSchema` are these:

- `schemaURL` – the XML schema URL. This is a unique identifier for the XML schema within Oracle XML DB. It is conventionally in the form of a URL; 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 with DBMS_XMLSCHEMA.REGISTERSCHEMA

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;
/
```

Schema Registration Considerations

When you register an XML schema, keep in mind the following considerations:

- The act of registering a schema has *no effect* on the status of any instance documents already loaded into Oracle XML DB Repository that claim to be members of the class defined the schema.

  Because the schema they reference was not yet registered, such instance documents were non-schema-based when they were loaded. *They remain non-schema-based after the schema is registered*.

  You must *delete* such instance documents, and *reload* them after registering the schema, in order to obtain schema-based documents.

-

Storage and Access Infrastructure

As part of registering an XML schema, Oracle XML DB also performs several tasks that facilitate storing, accessing, and manipulating XML instances that conform to the XML schema. These steps include:

- ***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 XML-schema

annotations to control how these object types are named and generated. See "SQL Object Types" on page 5-9 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 XML-schema annotations to control the names of the tables and to provide column-level and table-level storage clauses and constraints for use during table creation.

  See Also:

  - "Creating Default Tables During XML Schema Registration" on page 5-10
  - "Oracle XML Schema Annotations" on page 5-18

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 follows:

- If registration succeeds, then the operation is auto-committed.
- If registration fails, then the database is rolled back to the state before registration began.

Because XML schema registration potentially involves creating object types and tables, error recovery involves dropping any such created types and tables. The entire XML schema registration process is guaranteed to be atomic: either it succeeds or the database is restored to its state before the start of registration.

Managing and Storing XML Schemas

XML schema documents are themselves stored in Oracle XML DB as `XMLType` instances. XML schema-related `XMLType` types and tables are created as part of the Oracle XML DB installation script, `catxdbs.sql`.

The XML schema for 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 28, "Administering 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_BASE/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", "Using XML Schema with Oracle XML DB" on page 3-17

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.

```
DESCRIBE "PurchaseOrderType1668_T"

"PurchaseOrderType1668_T" is NOT FINAL
Name                 Null?  Type
-------------------- ------ --------------------------------
SYS_XDBPD$                  XDB.XDB$RAW_LIST_T
Reference                   VARCHAR2(30 CHAR)
Actions                     ActionsType1661_T
Reject                      RejectionType1660_T
Requestor                   VARCHAR2(128 CHAR)
User                        VARCHAR2(10 CHAR)
CostCenter                  VARCHAR2(4 CHAR)
ShippingInstructions        ShippingInstructionsTyp1659_T
SpecialInstructions         VARCHAR2(2048 CHAR)
LineItems                   LineItemsType1666_T
Notes                       VARCHAR2(4000 CHAR)

DESCRIBE "LineItemsType1666_T"

"LineItemsType1666_T" is NOT FINAL
Name                 Null? Type
-------------------- ----- --------------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
LineItem                   LineItem1667_COLL

DESCRIBE "LineItem1667_COLL"

"LineItem1667_COLL" VARRAY(2147483647) OF LineItemType1665_T
"LineItemType1665_T" is NOT FINAL
Name                Null? Type
------------------- ----- --------------------------------
SYS_XDBPD$                XDB.XDB$RAW_LIST_T
ItemNumber                NUMBER(38)
Description               VARCHAR2(256 CHAR)
Part                      PartType1664_T
```

Note: By default, the names of the object types and attributes in the preceding example are system-generated.

- Developers can use XML-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 and protocols that do not have any table specification, such as FTP or HTTP(S). In such cases, the XML instance is inserted into the default table.

Example 5–5 Default Table for Global Element PurchaseOrder

```
DESCRIBE "purchaseorder1669_tab"

Name                          Null? Type
----------------------------- ----- -----------------------
TABLE of
  SYS.XMLTYPE(
    XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
    Element "PurchaseOrder")
  STORAGE Object-relational TYPE "PurchaseOrderType1668_T"
```

If you provide a value for attribute `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` attributes is appended to the generated `CREATE TABLE` statement.

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 with DBMS_XMLSCHEMA.REGISTERSCHEMA", 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 functions of package `DBMS_XMLGEN`. Such cursors 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 code in Example 5–6. You can also examine `USER_XML_SCHEMAS`, `ALL_XML_SCHEMAS`, `USER_XML_TABLES`, and `ALL_XML_TABLES`.

Example 5–6 Data Dictionary Table for Registered Schemas

```
DESCRIBE DBA_XML_SCHEMAS

Name          Null? Type
------------ ----- ----------------------
OWNER               VARCHAR2(30)
SCHEMA_URL          VARCHAR2(700)
LOCAL               VARCHAR2(3)
SCHEMA              XMLTYPE(XMLSchema "http://xmlns.oracle.com/xdb/XDBSchema.xsd"
                            Element "schema")
INT_OBJNAME         VARCHAR2(4000)
QUAL_SCHEMA_URL     VARCHAR2(767)

SELECT OWNER, LOCAL, SCHEMA_URL FROM DBA_XML_SCHEMAS;

OWNER   LOC   SCHEMA_URL
-----   ---   ---------------------
XDB     NO    http://xmlns.oracle.com/xdb/XDBSchema.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBResource.xsd
XDB     NO    http://xmlns.oracle.com/xdb/acl.xsd
XDB     NO    http://xmlns.oracle.com/xdb/dav.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBStandard.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/xdblog.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/ftplog.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/httplog.xsd
XDB     NO    http://www.w3.org/2001/xml.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBFolderListing.xsd
XDB     NO    http://xmlns.oracle.com/xdb/stats.xsd
XDB     NO    http://xmlns.oracle.com/xdb/xdbconfig.xsd
SCOTT   YES   http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd

13 rows selected.

DESCRIBE DBA_XML_TABLES

Name          Null? Type
------------ ----- ----------------------
OWNER               VARCHAR2(30)
TABLE_NAME          VARCHAR2(30)
XMLSCHEMA           VARCHAR2(700)
SCHEMA_OWNER        VARCHAR2(30)
ELEMENT_NAME        VARCHAR2(2000)
STORAGE_TYPE        VARCHAR2(17)

SELECT TABLE_NAME FROM DBA_XML_TABLES
  WHERE XMLSCHEMA = 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd';

TABLE_NAME
--------------------
PurchaseOrder1669_TAB

1 row selected.
```

Deleting an XML Schema with DBMS_XMLSCHEMA.DELETESCHEMA

You can delete a registered XML schema by using procedure `DBMS_XMLSCHEMA.deleteSchema`. 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 Database PL/SQL Packages and Types Reference*

Example 5–7 Deleting an XML Schema with DBMS_XMLSCHEMA.DELETESCHEMA

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 Methods Related to XML Schema

XMLType Method	Description
`isSchemaBased()`	Returns `TRUE` if the `XMLType` instance is based on an XML schema, `FALSE` otherwise.

`getSchemaURL()`
`getRootElement()`
`getNamespace()`
	Return the XML schema URL, name of root element, and the namespace for an XML schema-based `XMLType` instance, respectively.

`schemaValidate()`
`isSchemaValid()`
`isSchemaValidated()`
`setSchemaValidated()`
	An `XMLType` instance can be validated against a registered XML schema using these validation methods. See Chapter 9, "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 Oracle XML DB Repository. The XML schema URL determines the path name of the resource in the 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. 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 24, "Repository 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/documenta
tion/purchaseOrder.xsd
```

This extended URL can be used by privileged users to specify XML schemas 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 XML 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.
>
> Role `XDBAdmin` also provides write access to this directory, assuming that it is protected by the default protected Access Control Lists (ACL). See Chapter 24, "Repository Resource Security" for further information on privileges and for details on role `XDBAdmin`.

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

```
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 global 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 5-29 and "Overriding the SQLType Value in XML Schema When Declaring Elements" on page 5-29

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 XML DB *internal use only*. You should never directly access or manipulate this column.

The positional descriptor attribute stores all information that cannot be stored in any of the other attributes. PD information is used to ensure the DOM fidelity of all XML documents stored in Oracle XML DB. Examples of such information include: ordering information, comments, processing instructions, and namespace prefixes.

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, it is not a good idea to suppress the PD attribute, because the extra information, such as comments and processing instructions, could be lost if there is no PD column.

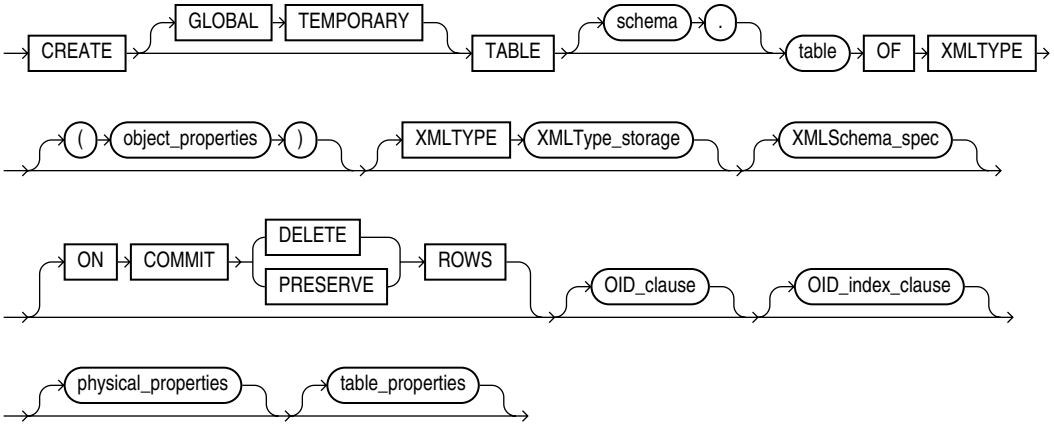
Creating XMLType Tables and Columns Based on XML Schema

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 a particular 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 a particular 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";
```

There are two ways to specify the XMLSchema and Element:

- as separator clauses
- using the Element clause with an XPointer notation

The data associated with an XMLType table or column that is constrained to an XML schema can be stored in 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";
```

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.

The collection and the LOB column must be identified using object-relational notation.

Example 5–12 Specifying Storage Options 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 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, SYS_NC_ARRAY_INDEX$))
                  ORGANIZATION INDEX OVERFLOW)
```

```
  VARRAY "XMLDATA"."LineItems"."LineItem"
    STORE AS TABLE lineitem_table1
                  ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_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, SYS_NC_ARRAY_INDEX$))
                  ORGANIZATION INDEX OVERFLOW)
  VARRAY xml_document."XMLDATA"."LineItems"."LineItem"
    STORE AS TABLE lineitem_table2
                  ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$))
                  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:** Use the *thick* JDBC driver with schema-based `XMLType` values stored object-relationally. (You can use either the thin or the thick driver with `CLOB` storage of `XMLType` 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.

It is not possible to define constraints for `XMLType` tables and columns that make use of unstructured storage.

> **See Also:** "Constraints on Repetitive Elements in Schema-Based XML" on page 7-26

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 to `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 to `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 identifiers when working directly with SQL.
- To allow XPath rewriting in the case of (document-correlated recursive) XPath queries, that is, for certain `extract`, `extractValue`, and `existsNode` applications whose XPath expression targets recursive XML data.

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 the empty string `""` 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 datatypes and SQL datatypes. 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. A nested table is created for each element that specifies `maxOccurs > 1`. The nested tables are created with system-generated names.

Note: Annotation `storeVarrayAsTable="true"` causes element collections to be persisted as rows in an index-organized table (IOT). Oracle Text does not support IOTs. Do *not* use this annotation if you will need to use Oracle Text indexes for text-based `ora:contains` searches over a collection of elements. See "ora:contains Searches Over a Collection of Elements" on page 10-23. To provide for searching with Oracle Text indexes:

1. Set `genTables="false"` during schema registration.
2. Create the necessary tables *manually*, without using the clause `ORGANIZATION INDEX OVERFLOW`, so the tables will be *heap-organized* instead of index-organized (IOT).

You do not need 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>
```

Note: As always:

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

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

The schema element includes the declaration of the `xdb` namespace. It also includes the annotation `xdb:storeVarrayAsTable="true"`. This causes 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;
/

SELECT table_name, xmlschema, element_name FROM user_xml_tables;

TABLE_NAME     XMLSCHEMA                              ELEMENT_NAME
-------------  -----------------------------------    -------------
PURCHASEORDER  http://xmlns.oracle.com/xdb/documen   PurchaseOrder
               tation/purchaseOrder.xsd

1 row selected.

DESCRIBE purchaseorder

Name                           Null? Type
-----------------------------  ----- -----------------
TABLE of SYS.XMLTYPE(XMLSchema
 "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
 ELEMENT "PurchaseOrder") STORAGE Object-relational TYPE "PURCHASEORDER_T"

DESCRIBE purchaseorder_t

PURCHASEORDER_T is NOT FINAL
Name                 Null? Type
-------------------- ----- --------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
REFERENCE                  VARCHAR2(30 CHAR)
ACTION_COLLECTION          ACTIONS_T
REJECT                     REJECTION_T
REQUESTOR                  VARCHAR2(128 CHAR)
EMAIL                      VARCHAR2(10 CHAR)
```

```
COSTCENTER                VARCHAR2(4 CHAR)
SHIPPINGINSTRUCTIONS      SHIPPING_INSTRUCTIONS_T
SPECIALINSTRUCTIONS       VARCHAR2(2048 CHAR)
LINEITEM_COLLECTION       LINEITEMS_T
Notes                     CLOB

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

DESCRIBE lineitem_v

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

DESCRIBE part_t

PART_T is NOT FINAL
Name                 Null? Type
-------------------- ----- --------------------------
ID                         VARCHAR2(14 CHAR)
QUANTITY                   NUMBER(12,2)
UNITPRICE                  NUMBER(8,4)

SELECT table_name, parent_table_column FROM user_nested_tables
  WHERE parent_table_name = 'purchaseorder';

TABLE_NAME                     PARENT_TABLE_COLUMN
----------                     -----------------------
SYS_NTNOHV+tfSTRaDTA9FETvBJw==   "XMLDATA"."LINEITEM_COLLECTION"."LINEITEM_VARRAY"
SYS_NTV4bNVqQ1S4WdCIvBK5qjZA==   "XMLDATA"."ACTION_COLLECTION"."ACTION_VARRAY"

2 rows selected.
```

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`.

Type `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 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 as 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, such as cyclic references, where SQL does 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 and protocols where table name is not specified, such as FTP and HTTP(S).

Table 5–3 Annotations 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 in XML Schema Declarations

Attribute	Values	Default	Description
mapUnboundedStringToLob	true | false	false	If true, unbounded strings are mapped to CLOB by default. Similarly, unbounded binary data gets mapped to a BLOB value, 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.

Note: Annotation storeVarrayAsTable="true" causes element collections to be persisted as rows in an index-organized table (IOT). Oracle Text does not support IOTs. Do *not* use this annotation if you will need to use Oracle Text indexes for text-based ora:contains searches over a collection of elements. See "ora:contains Searches Over a Collection of Elements" on page 10-23. To provide for searching with Oracle Text indexes:

1. Set genTables="false" during schema registration.
2. Create the necessary tables *manually*, without using the clause ORGANIZATION INDEX OVERFLOW, so the tables will be *heap-organized* instead of index-organized (IOT).

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 object types and DOM fidelity values.

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,
       extractValue(value(ct),
                    '/xs:complexType/@xdb:maintainDOM',
                    'xmlns:xs="http://www.w3.org/2001/XMLSchema"
                     xmlns:xdb="http://xmlns.oracle.com/xdb"')
       DOM_FIDELITY
  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            DOM_FIDELITY
-------------------------  -----------------------  ------------
PurchaseOrderType          PURCHASEORDER_T          true
LineItemsType              LINEITEMS_T              true
LineItemType               LINEITEM_T               true
PartType                   PART_T                   true
ActionsType                ACTIONS_T                true
RejectionType              REJECTION_T              true
ShippingInstructionsType   SHIPPING_INSTRUCTIONS_T  true

7 rows selected.
```

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 Types with DBMS_XMLSCHEMA" on page 5-29 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:

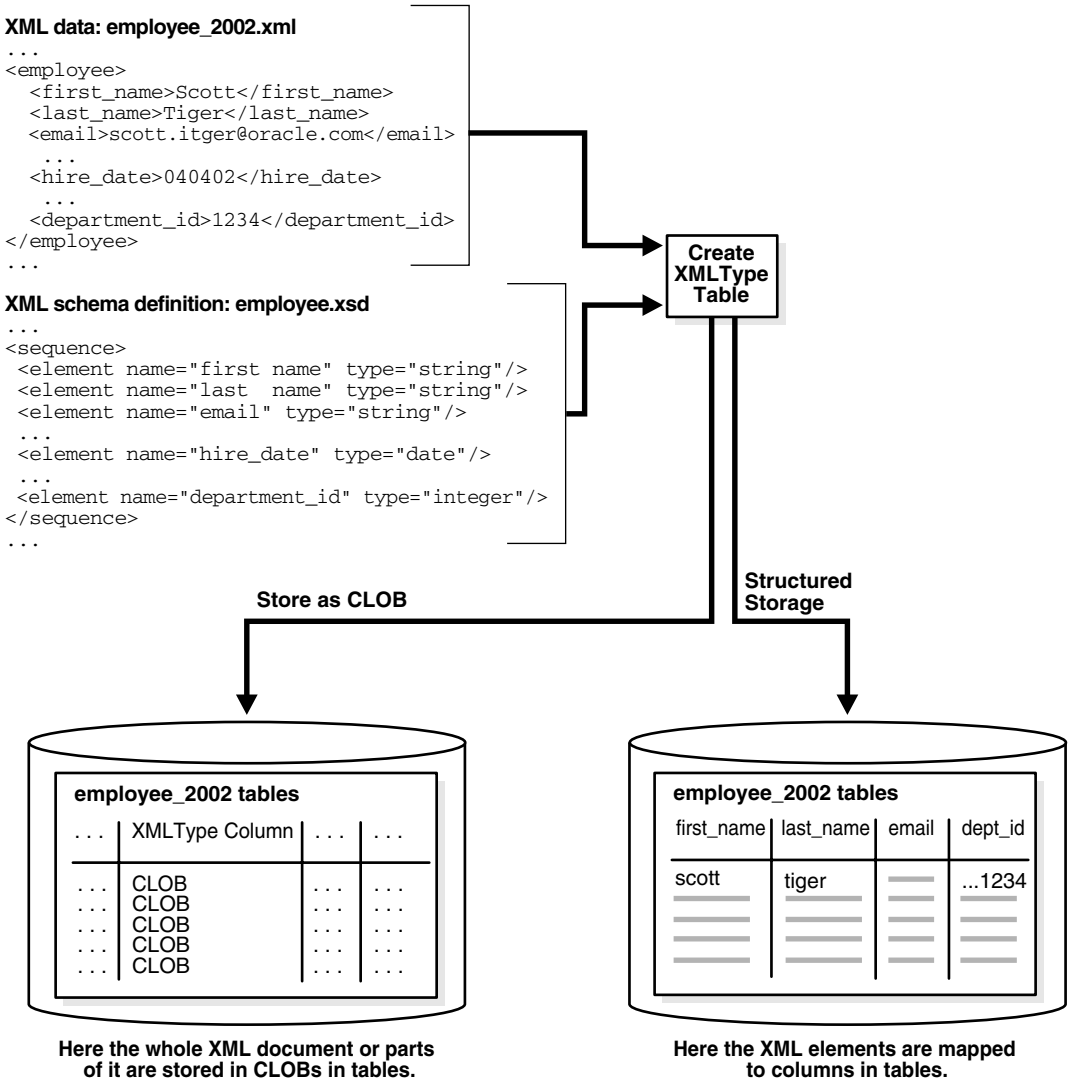
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
```

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

...

<xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION" xdb:SQLCollType="ACTION_V">
      <xs:complexType xdb:SQLType="ACTION_T">
        <xs:sequence>
          <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
          <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_ACTIONED"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
```

```
    </xs:all>
  </xs:complexType>
  <xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
    <xs:sequence>
      <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
      <xs:element name="address" type="AddressType" minOccurs="0" xdb:SQLName="SHIP_TO_ADDRESS"/>
      <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
    </xs:sequence>
  </xs:complexType>
  ...
</xs:schema>
```

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 value in one XMLType column, or stored object-relationally and spread out across several columns in the table.

Mapping Types with DBMS_XMLSCHEMA

Use PL/SQL package `DBMS_XMLSCHEMA` to map types 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 5-29.
- 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 when processing the complexType values that are used to declare elements and the elements declared within the complexType, the SQLInline attribute is forced to be false and the correct SQL mapping is set to REF XMLType.
- If maxOccurs > 1, a varray type 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 7, "XML Schema Storage and Query: Advanced" for more information about mapping simpleType values and complexType values to SQL.

Mapping simpleType to SQL

This section describes how XML schema definitions map XML Schema simpleType to SQL object types. Figure 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

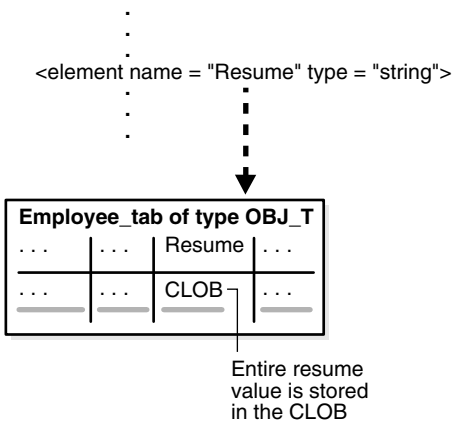


Table 5–5 Mapping XML String Datatypes to SQL

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

Table 5–7 (Cont.) Default Mapping of Numeric XML Primitive Types to SQL

XML Simple Type	Default Oracle DataType	totalDigits (m), fractionDigits(n) Specified	Compatible Datatypes
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	--

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.

simpleType: Mapping XML Strings to SQL VARCHAR2 Versus CLOBs

If the XML schema specifies the datatype to be a 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"/>
```

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.

For 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 complexType to SQL

Using XML Schema, a `complexType` is mapped to a SQL object type as follows:

- ***XML attributes*** declared within the `complexType` are mapped to ***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:

```
<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>
```

Note: As always:

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

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

6

XPath Rewrite

This chapter explains the fundamentals of XPath rewrite in Oracle XML DB and how to use it for XML schema-based structured storage. It details the rewriting of XPath-expression arguments to these important SQL functions: `existsNode`, `extract`, `extractValue`, `XMLSequence`, `updateXML`, `insertChildXML`, and `deleteXML`.

This chapter contains these topics:

See Also: "XPath Rewrite on XMLType Views" on page 18-20

Overview of XPath Rewrite

When `XMLType` data 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. The optimization process of rewriting XPath expressions is called **XPath rewrite**.

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 XPath-expression arguments to SQL functions such as `existsNode`, `extract`, `extractValue`, and `updateXML`. This enables the XPath expression to be evaluated against the XML document without constructing the XML document in memory.

The XPath expressions that are rewritten by Oracle XML DB are a proper subset of those that are supported by Oracle XML DB. Whenever you can do so without losing functionality, use XPath expressions that can be rewritten.

Example 6–1 XPath Rewrite

For example, a query such as the following tries to obtain the `Company` element and compare it with the literal '`Oracle`':

```
SELECT OBJECT_VALUE FROM mypurchaseorders p
  WHERE extractValue(OBJECT_VALUE, '/PurchaseOrder/Company') = 'Oracle';
```

Because table `mypurchaseorders` was created with XML schema-based structured storage, `extractValue` is rewritten to the underlying relational column that stores the company information for the `purchaseOrder`. The query is rewritten to the following:

```
SELECT VALUE(p) FROM mypurchaseorders p WHERE p.xmldata.Company = 'Oracle';
```

> **Note:** `XMLDATA` is a pseudo-attribute of datatype `XMLType` that enables direct access to the underlying object column. See Chapter 4, "XMLType Operations".

If there is a regular index created on the `Company` column, such as the following, then the preceding query uses the index for its evaluation.

```
CREATE INDEX company_index
  ON mypurchaseorders e (extractValue(OBJECT_VALUE, '/PurchaseOrder/Company'));
```

XPath rewrite happens for XML schema-based tables and both schema-based and non-schema-based views. In this chapter, we consider only examples related to schema-based tables.

Example 6–2 XPath Rewrite with UPDATEXML

The XPath argument to SQL function `updateXML` in this example is rewritten to the equivalent object relational SQL statement given in Example 6–3.

```
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 6–3 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.
```

See Also: Chapter 3, "Using Oracle XML DB", "Understanding and Optimizing XPath Rewrite" on page 3-57, for additional examples of rewrite over schema-based and non-schema-based views

Where Does XPath Rewrite Occur?

XPath rewrite happens for the following SQL functions:

- `extract`
- `existsNode`
- `extractValue`
- `updateXML`
- `insertChildXML`
- `deleteXML`
- `XMLSequence`

XPath rewrite can happen when these SQL functions are present in any expression in a query, DML, or DDL statement. For example, you can use function `extractValue` to create indexes on the underlying relational columns.

Example 6–4 SELECT Statement and XPath Rewrites

This example gets the existing purchase orders:

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/Company')
  FROM mypurchaseorders x
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder/Item[1]/Part') = 1;
```

Here are some examples of statements that get rewritten to use underlying columns:

Example 6–5 DML Statement and XPath Rewrites

This example deletes all `PurchaseOrders` where the `Company` is not `Oracle`:

```
DELETE FROM mypurchaseorders x
  WHERE extractValue(OBJECT_VALUE, '/PurchaseOrder/Company') = 'Oracle Corp';
```

Example 6–6 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 mypurchaseorders e (extractValue(OBJECT_VALUE,'/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.

Which XPath Expressions Are Rewritten?

An XPath expression can generally be rewritten if all of the following are *true*:

- The XML function or method is rewritable.

  SQL functions `extract`, `existsNode`, `extractValue`, `updateXML`, `insertChildXML`, `deleteXML`, and `XMLSequence` are rewritten. Except method `existsNode()`, *none* of the corresponding `XMLType` *methods* are rewritten.

- The XPath expression uses only the descendent axis.

  Expressions involving axes (such as parent and sibling) other than descendent are *not* rewritten. Expressions that select attributes, elements, or text nodes can be rewritten. XPath predicates are rewritten to SQL predicates.

- The XML Schema constructs for the XPath expression are rewritable.

  XML Schema constructs such as complex types, enumerated values, lists, inherited (derived) types, and substitution groups are rewritten. Constructs such as recursive type definitions are *not* rewritten.

- The storage structure chosen during XML-schema registration is rewritable.

  Storage using the object-relational mechanism is rewritten. Storage of complex types using `CLOB`s are *not* rewritten

Table 6–1 lists the kinds of XPath expressions that can be translated into underlying SQL queries.

Table 6–1 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 function is used during a `CREATE INDEX` operation.
Predicates: `[Company="Oracle"]`	Predicates in the XPath are rewritten into SQL predicates.
List index (positional predicate): `lineitem[1]`	Indexes are rewritten to access the nth item in a collection. These are not rewritten for `updateXML`, `insertChildXML`, and `deleteXML`.

Table 6–1 (Cont.) Sample List of XPath Expressions for Translation to Underlying SQL constructs

XPath Expression for Translation	Description
Wildcard traversals: `/PurchaseOrder/*/Part`	If the wildcard can be translated to a unique XPath (for example, `/PurchaseOrder/Item/Part`), then it is rewritten, unless it is the last entry in the path expression.
Descendent axis: `/PurchaseOrder//Part`	Similar to a 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.

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 rewritten if they occur between the concatenation (`||`) operators and are inside the double-quotes.
Unnest operations using XMLSequence `table(XMLSequence(extract(...)))`	When used in a `table` function call, `XMLSequence` combined with `extract` 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 axis (XML schema-based data only): Rewrites over the 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 XML schema indicates that multiple descendants of the children potentially match, and there is no unique path that 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 XML schema indicates that multiple grandchildren potentially match, and there is no unique path that 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 are *not* rewritten:

- XPath functions other than those listed earlier. The listed functions are rewritten only if the input is an element with scalar content.
- XPath variable references.
- All axes other than the child and attribute axes.
- Recursive type definitions with descendent axes.
- UNION operations.

Common XMLSchema Constructs Supported in XPath Rewrite

In addition to standard XML Schema constructs such as complexTypes and sequences, 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. The `any` attributes are handled in a similar way.
- Datatype operations that cannot be coerced, such as addition of a Boolean value and 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 as nested tables.

- XML functions over schema-based and non-schema-based `XMLType` views and SQL/XML views also get rewritten.

  See Also: Chapter 18, "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

XPath Rewrite Can Change Comparison Semantics

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 nonnumeric entities is different*.

In XPath 1.0, the comparison operators, >, <, >=, and <=, use only numeric comparison. The two operands are converted to numeric values before comparison. If either of them fails to be converted to a numeric value, the comparison returns `false`.

For instance, if I have an XML-schema element definition such as the following, then an XPath predicate such as `[ShipDate < '2003-02-01']` will always evaluate to false with functional evaluation.

```
<element name="ShipDate" type="xs:date" xdb:SQLType="DATE"/>
```

This is because the string value `'2003-02-01'` cannot be converted to a numeric quantity. With XPath rewrite, however, this predicate gets translated to a SQL *date* comparison, and will evaluate to true or false, depending on the value of `ShipDate`.

Similarly if a collection value is compared with another collection value, the XPath 1.0 semantics dictate that the values must be converted to strings and then compared. With XPath rewrite, the comparison uses the rules for comparing SQL values.

To suppress this behavior, you can turn off rewrite either using query hints or session level events.

How Are XPath Expressions Rewritten?

This section uses the same purchase-order XML schema introduced earlier in this chapter.

Example 6–7 Creating XML Schema-Based Purchase-Order Data

```
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.

```
CREATE TABLE mypurchaseorders OF XMLType
  XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder"
  VARRAY xmldata."Item" STORE AS TABLE item_nested;

Table created
```

Now, we insert a purchase order into this table.

```
INSERT INTO mypurchaseorders
  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 the SYS_XDBPD$ (PD) 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 or not the element was present in the input document). This is needed for simpleType elements, because they map to simple relational columns. In this case, both empty and missing simpleType elements map to NULL values in the column, and the SYS_XDBPD$ attribute can be used to distinguish the two cases. The XPath rewrite mechanism takes into account the presence or absence of the SYS_XDBPD$ attribute, and rewrites queries appropriately.

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 path expressions is 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 – see Table 6–2.

Table 6–2 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"

Schema-Based: Mapping for simpleType Elements

An XPath expression can contain a text() node test, which targets the text node (content) of an element. When rewriting, this maps directly to the underlying relational columns. For example, the XPath expression "/PurchaseOrder/PONum/text()" maps directly to the SQL column XMLDATA."PONum".

A NULL in the PONum column implies that the text value is not available: either the text() node test is not present in the input document or the element itself is missing. If the column is NULL, there is no need to check for the existence of the element in the SYS_XBDPD$ attribute.

The XPath "/PurchaseOrder/PONum" also maps to the SQL attribute XMLDATA."PONum". However, in this case, XPath rewrite must check for the existence of the element itself, using attribute SYS_XDBPD$ in column XMLDATA.

Schema-Based: Mapping of Predicates

Predicates are mapped to SQL predicate expressions. Since the predicates are rewritten to SQL, the comparison rules of SQL are used instead of the XPath 1.0 semantics.

Example 6–8 Mapping Predicates

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")
```

The following query is rewritten to the structured (object-relational) equivalent, and will not require functional evaluation of the XPath.

```
SELECT extract(OBJECT_VALUE, '/PurchaseOrder/Item').getClobval()
  FROM mypurchaseorders p
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[PONum=1001 AND Company = "Oracle Corp"]') = 1;
```

Schema-Based: Mapping of Collection Predicates XPath expressions can involve relational collection expressions. In Xpath 1.0, these are treated as existential checks: if at least one member of the collection satisfies the expression, then the expression is true.

Example 6–9 Mapping Collection Predicates

The collection predicate in this XPath expression involves the relational greater-than operator (>):

```
/PurchaseOrder[Items/Price > 200]
```

This maps to the following SQL collection expression:

```
exists(SELECT NULL FROM table(XMLDATA."Item") x WHERE x."Price" > 200)
```

In this example, a collection is related to a scalar value. More complicated rewrites occur with a relation between two collections. For example, in the following XPath expression, both `LineItems` and `ShippedItems` are collections.

```
/PurchaseOrder[LineItems = ShippedItems]
```

In this case, if *any* combination of nodes from these two collections satisfies the equality, then the predicate is considered satisfied.

Example 6–10 Mapping Collection Predicates, Using EXISTSNODE

Consider a fictitious XPath that checks if a `Purchaseorder` has `Items` whose `Price` and `Part` number are the same:

```
/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"))
```

The following query is rewritten to the structured equivalent:

```
SELECT extract(OBJECT_VALUE, '/PurchaseOrder/Item').getClobval()
  FROM  mypurchaseorders p
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[Item/Price = Item/Part]') = 1;
```

Schema-Based: Document Ordering with Collection Traversals

Most of the rewrite preserves the original document ordering. However, because SQL does not guarantee ordering on the results of subqueries when selecting elements from a collection using SQL function `extract`, the resultant nodes may not be in document order.

Example 6–11 Document Ordering with Collection Traversals

For example:

```
SELECT extract(OBJECT_VALUE, '/PurchaseOrder/Item[Price>2100]/Part')
  FROM mypurchaseorders p;
```

This query is rewritten to use a subquery:

```
SELECT (SELECT XMLAgg(XMLForest(x."Part" AS "Part"))
          FROM table(XMLDATA."Item") x WHERE  x."Price" > 2100)
  FROM mypurchaseorders p;
```

In most cases, the result of the aggregation is in the same order as the collection elements, but this is not guaranteed. So, the results may not be in document order.

Schema-Based: Collection Position

An XPath expression can also access an element at a particular position of a collection. For example, `"/PurchaseOrder/Item[1]/Part"` is rewritten to extract out the first `Item` element of the collection, and access the `Part` attribute within that.

If the collection is stored as a varray, then this operation retrieves the nodes in the same order as in the original document. If the collection is stored as a nested table, then the order is indeterminate.

Schema-Based: XPath Expressions That Cannot Be Satisfied

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 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 function-based evaluation. For schema-based documents, if the function (such as `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 6–12 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 mypurchaseorders p
  WHERE  existsNode(OBJECT_VALUE, '/PurchaseOrder/PONum') = 1;
```

is equivalent to the query:

```
SELECT *
  FROM mypurchaseorders p
  WHERE existsNode(
          OBJECT_VALUE,
          '/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 `existsNode` expression should be rewritten as follows:

```
existsNode(
  OBJECT_VALUE,
  '/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 `existsNode` expression 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 or not the function is rewritten.

Schema-Based: Date Format Conversions

Date datatypes such as `DATE`, `gMONTH`, and `gDATE` have different format in XML Schema and SQL. If an expression has a string value for columns of such datatypes, then the rewrite automatically provides the XML format string to convert the string value correctly. Thus, the string value specified for a `DATE` column must match the XML date format, not the SQL `DATE` format.

Example 6–13 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 functions such as `extract`), XML format strings are added to convert them to the same date format as XML.

Existential Checks for Attributes and Elements with Scalar Values

SQL function `existsNode` checks for the existence of a node addressed by an XPath; function `extract` returns a node addressed by an XPath. Oracle XML DB needs to perform special checks for `simpleType` elements and for attributes used in `existsNode` expressions. This is because the SQL column value alone cannot distinguish whether an attribute or a `simpleType` element is missing or is empty; a `NULL` SQL column can represent either. These special checks are not required for intermediate elements, because the value of the user-defined SQL datatype indicates the absence or emptiness of the element.

Consider, for example, this expression:

```
existsNode(OBJECT_VALUE, '/PurchaseOrder/PONum/text()') = 1;
```

Because the query is only interested in the text value of the node, this is rewritten to:

```
(p.XMLDATA."PONum" IS NOT NULL)
```

Consider this expression, without the `text()` node test:

```
existsNode(OBJECT_VALUE, '/PurchaseOrder/PONum') = 1;
```

In this case, Oracle XML DB must check the `SYS_XDBPD$` attribute in the parent node to determine whether the element is empty or is missing. This check is done *internally*. It can be represented in *pseudocode* as follows:

```
node_exists(p.XMLDATA."SYS_XDBPD$", "PONum")
```

The pseudofunction ***`node_exists`*** is used for illustration only. It represents an Oracle XML DB implementation that uses its first argument, the positional-descriptor (PD) column (`SYS_XDBPD$`), to determine whether or not its second argument (element or attribute) node exists. It returns true if so, and false if not.

In the case of `extract` expressions, this check needs to be done for both attributes and elements. An expression of the form `extract(OBJECT_VALUE, '/PurchaseOrder/PONum')` maps to pseudocode such as the following:

```
CASE WHEN node_exists(p.XMLDATA.SYS_XDBPD$", "PONum")
       THEN XMLElement("PONum", p.XMLDATA."PONum")
       ELSE NULL END;
```

> **Note:** Be aware of this overhead when writing `existsNode` and `extract` expressions. You can avoid this overhead by using a `text()` node test in the XPath expression; using `extractValue` to obtain only the node value; or by turning off DOM fidelity for the parent node. DOM fidelity can be turned off by setting the value of the attribute `maintainDOM` in the element definition to be `false`. When turned off, empty elements and attributes are treated as missing.

Diagnosing XPath Rewrite

To determine if your XPath expressions are getting rewritten, you can use one of the following techniques:

Using EXPLAIN PLAN with XPath Rewrite

This section shows how you can use `EXPLAIN PLAN` to examine the query plans after rewrite. See "Understanding and Optimizing XPath Rewrite" on page 3-57 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 table `mypurchaseorders` we can see the use of `EXPLAIN PLAN`. We create an index on the `Company` element of `PurchaseOrder` to show how the plans differ.

```
CREATE INDEX company_index ON mypurchaseorders
       (extractValue(OBJECT_VALUE,'/PurchaseOrder/Company'));

Index created.

EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/PONum')
    FROM mypurchaseorders
    WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[Company="Oracle"]') = 1;

Explained.

SELECT PLAN_TABLE_OUTPUT
  FROM table(DBMS_XPLAN.display('plan_table', NULL, 'serial'))
/

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
| Id  | Operation                    | Name             | Rows | Bytes | Cost |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                  |      |       |      |
|   1 |  TABLE ACCESS BY INDEX ROWID | MYPURCHASEORDERS |      |       |      |
|*  2 |   INDEX RANGE SCAN           | COMPANY_INDEX    |      |       |      |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("MYPURCHASEORDERS"."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.

```
EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/PONum')
    FROM mypurchaseorders
    WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[Company+PONum="Oracle"]') = 1;

Explained.
```

```
SELECT PLAN_TABLE_OUTPUT
  FROM table(DBMS_XPLAN.display('plan_table', NULL, 'serial'))
/

PLAN_TABLE_OUTPUT
-------------------------------------------------------------
| Id  | Operation          | Name
-------------------------------------------------------------
|   0 | SELECT STATEMENT   |
|*  1 |  FILTER            |
|   2 |   TABLE ACCESS FULL| MYPURCHASEORDERS
|*  3 |   TABLE ACCESS FULL| ITEM_NESTED
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(EXISTSNODE(SYS_MAKEXML('C6DB2B4A1A3B0
              6CDE034080020E5CF39',2300,"MYPURCHASEORDERS"."XMLEXTRA",
              "MYPURCHASEORDERS"."XMLDATA"),
      '/PurchaseOrder[Company+PONum="Oracle"]')=1)
   3 - filter("NESTED_TABLE_ID"=:B1)
```

Using Events with XPath Rewrite

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 XPath 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 is not rewritten and is instead evaluated functionally. 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 6–3 lists the various levels and the corresponding behavior.

Table 6–3 Event Levels and Behaviors

Event	Turn *off* functional evaluation of . . .
Level 0x1	all XML functions
Level 0x2	`extract`
Level 0x4	`existsNode`
Level 0x8	`transform`
Level 0x10	`extractValue`
Level 0x20	`updateXML`
Level 0x40	`insertXMLbefore`
Level 0x80	`appendChildXML`
Level 0x100	`deleteXML`
Level 0x200	`XMLSequence`
Level 0x4000	`insertChildXML`
Level 0x8000	`XMLQuery`

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.

```
SELECT OBJECT_VALUE FROM mypurchaseorders
  WHERE existsNode(OBJECT_VALUE, '/PurchaseOrder[Company+PONum="Oracle"]')=1 ;

ERROR:
ORA-19022: XML XPath functions are disabled
```

Tracing Reasons that Rewrite Does Not Occur

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`:

```
ALTER SESSION SET EVENTS '19027 TRACE NAME CONTEXT FOREVER, LEVEL 8192';

Session altered.

EXPLAIN PLAN FOR
  SELECT OBJECT_VALUE FROM mypurchaseorders
    WHERE existsNode(OBJECT_VALUE, '/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 are inputs to an arithmetic function that are not numeric.

```
NO REWRITE
        XPath ==> /PurchaseOrder[Company+PONum = "Oracle"]
        Reason ==> non numeric inputs to arith{2}{4}
```

XPath Rewrite of SQL Functions

This section details XPath rewrite for SQL functions `existsNode`, `extractValue`, `extract`, `XMLSequence`, `updateXML`, `insertChildXML`, and `deleteXML`. It explains the overhead involved in certain types of operations using `existsNode` or `extract`, and how to avoid it.

An *update* using one of these SQL functions normally involves updating a copy of the XML document and then replacing the entire document with the newly modified document.

When `XMLType` data is stored in an object-relational manner using XML-schema mapping, updates are optimized to directly modify pieces of the document in place. For example, an update of the `PONum` element can be rewritten to directly update the `XMLDATA.PONum` column, instead of materializing the whole document in memory and then performing the update.

Each of the functions `updateXML`, `insertChildXML`, and `deleteXML` must satisfy different conditions for it to use such rewrite optimization during update. If all of the conditions are satisfied, then the functional expression is rewritten into a simple relational update. For example:

```
UPDATE purchaseorder_table
```

```
  SET OBJECT_VALUE =
      updateXML(OBJECT_VALUE,
                '/PurchaseOrder/@PurchaseDate', '2002-01-02',
                '/PurchaseOrder/PONum/text()', 2200);
```

This update operation is rewritten as something like the following:

```
UPDATE purchaseorder_table p
  SET p.XMLDATA."PurchaseDate" = TO_DATE('2002-01-02', 'SYYYY-MM-DD'),
      p.XMLDATA."PONum" = 2100;
```

XPath Rewrite for EXISTSNODE

SQL function `existsNode` returns one (`1`) if the XPath argument targets a nonempty sequence of nodes (text, element, or attribute); otherwise, it returns zero (`0`). The value is determined differently, depending on the kind of node targeted by the XPath argument:

- If the XPath argument targets a text node (using node test `text()`) or a `complexType` element node, Oracle XML DB simply checks whether the database representation of the element content is `NULL`.
- Otherwise, the XPath argument targets a `simpleType` element node or an attribute node. Oracle XML DB checks for the existence of the node using the positional-descriptor attribute `SYS_XDBPD$`. If `SYS_XDBPD$` is absent, then the existence of the node is determined by checking whether or not the column is `NULL`.

EXISTSNODE Mapping with Document Order Preserved

Table 6–4 shows the mapping of various XPaths in the case of SQL function `existsNode` when document ordering is preserved; that is, when `SYS_XDBPD$` exists and `maintainDOM="true"` is present in the schema document.

Table 6–4 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` *`node_exists`*[1]`(XMLDATA.SYS_XDBPD$, 'PurchaseDate') THEN 1 ELSE 0 END`
`/PurchaseOrder/PONum`	`CASE WHEN` *`node_exists`*[1]`(XMLDATA.SYS_XDBPD$, 'PONum') 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` *`node_exists`*[1]`(XMLDATA.SYS_XDBPD$, 'PurchaseDate') THEN 1 ELSE 0 END`
`/PurchaseOrder/PONum/text()`	`CASE WHEN XMLDATA."PONum" IS NOT NULL THEN 1 ELSE 0`

Table 6–4 (Cont.) XPath Mapping for EXISTSNODE with Document Ordering Preserved

XPath Expression	Maps to
/PurchaseOrder/Item	CASE WHEN exists(SELECT NULL FROM table(XMLDATA."Item") x WHERE value(x) IS NOT NULL) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part	CASE WHEN exists(SELECT NULL FROM table(XMLDATA."Item") x WHERE *node_exists*[1](x.SYS_XDBPD$, 'Part')) 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

[1] Pseudofunction *node_exists* is used for illustration only. It represents an Oracle XML DB implementation that uses its first argument, the PD column, to determine whether or not its second argument node exists. It returns true if so, and false if not.

Example 6–14 EXISTSNODE Mapping with Document Order Preserved

Using the preceding mapping, this query checks whether purchase order 1001 contains a part with price greater than 2000:

```
SELECT count(*)
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
                   '/PurchaseOrder[PONum=1001 and Item/Price > 2000]') = 1;
```

This is rewritten as something like the following:

```
SELECT count(*)
  FROM  purchaseorder 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;
```

This `CASE` expression is further optimized due to the constant relational equality expressions. The query becomes:

```
SELECT count(*)
  FROM purchaseorder p
  WHERE p.XMLDATA."PONum"=1001
    AND exists(SELECT NULL FROM table(p.XMLDATA."Item") x
               WHERE  x."Price" > 2000);
```

This uses relational indexes for its evaluation, if present on the `Part` and `PONum` columns.

EXISTSNODE Mapping Without Document Order Preserved

If the positional-descriptor attribute `SYS_XDBPD$` does not exist (that is, if the XML schema specifies `maintainDOM="false"`) then `NULL` scalar columns map to `simpleType` elements that do not exist. In that case, you do not need to check for node existence using attribute `SYS_XDBPD$`. Table 6–5 shows the mapping of `existsNode` in the absence of the `SYS_XDBPD$` attribute.

Table 6–5 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

XPath Rewrite for EXTRACTVALUE

SQL function `extractValue` is a shortcut for extracting text nodes and attributes using function `extract` and then using method `getStringVal()` or `getNumberVal()` to obtain the scalar content. Function `extractValue` returns the values of attribute nodes or the text nodes of elements with scalar values. Function `extractValue` cannot handle XPath expressions that return multiple values or `complexType` elements.

Table 6–6 shows the mappings of various XPath expressions for function `extractValue`. If an XPath expression targets an element, then `extractValue` retrieves the text node of the element. For example, `/PurchaseOrder/PONum` and `/PurchaseOrder/PONum/text()` are handled identically by `extractValue`: both retrieve the scalar content of `PONum`.

Table 6–6 XPath Mapping for EXTRACTVALUE

XPath Expression	Maps to
/PurchaseOrder	Not supported. Function `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. Function `extractValue` can only retrieve values for scalar elements and attributes.

Table 6–6 (Cont.) XPath Mapping for EXTRACTVALUE

XPath Expression	Maps to
`/PurchaseOrder/Item/Part`	Not supported. Function `extractValue` cannot retrieve multiple scalar values.
`/PurchaseOrder/Item/Part/text()`	Not supported. Function `extractValue` cannot retrieve multiple scalar values.

Example 6–15 Rewriting EXTRACTVALUE

Consider this SQL query:

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/PONum') FROM purchaseorder
  WHERE extractValue(OBJECT_VALUE, '/PurchaseOrder/PONum') = 1001;
```

This query would be rewritten as something like the following:

```
SELECT p.XMLDATA."PONum" FROM purchaseorder p WHERE p.XMLDATA."PONum" = 1001;
```

Because it gets rewritten to simple scalar columns, any indexes on attribute `PONum` can be used to satisfy the query.

Creating Indexes with EXTRACTVALUE

Function `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 6–16 Creating Indexes with EXTRACTVALUE

```
CREATE INDEX my_po_index ON purchaseorder
  (extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference);
```

This would would get rewritten into something like the following:

```
CREATE INDEX my_po_index ON purchaseorder x (x.XMLDATA."Reference");
```

This produces a regular B*Tree index. Unlike a function-based index, the same index can now satisfy queries that target the column, such as the following:

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

XPath Rewrite for EXTRACT

SQL function `extract` retrieves XPath results as XML. For Xpath expressions involving text nodes, `extract` is rewritten similarly to `extractValue`.

EXTRACT Mapping with Document Order Maintained

Table 6–7 shows the mapping of various XPath expressions inside `extract` expressions when document order is preserved (that is, when `SYS_XDBPD$` exists and `maintainDOM="true"` in the XML schema document).

Table 6–7 XPath Mapping for EXTRACT with Document Ordering Preserved

XPath	Maps to
/PurchaseOrder	XMLForest(XMLDATA AS "PurchaseOrder")
/PurchaseOrder/@PurchaseDate	CASE WHEN *node_exists*[1](XMLDATA.SYS_XDBPD$, 'PurchaseDate') THEN XMLElement("", XMLDATA."PurchaseDate") ELSE NULL END;
/PurchaseOrder/PONum	CASE WHEN *node_exists*[1](XMLDATA.SYS_XDBPD$, 'PONum') THEN XMLElement("PONum", XMLDATA."PONum") ELSE NULL END
/PurchaseOrder[PONum = 2100]	SELECT XMLForest(XMLDATA as "PurchaseOrder") FROM DUAL WHERE XMLDATA."PONum" = 2100
/PurchaseOrder [PONum = 2100]/@PurchaseDate	SELECT CASE WHEN *node_exists*[1](XMLDATA.SYS_XDBPD$, 'PurchaseDate') THEN XMLElement("", XMLDATA."PurchaseDate") ELSE NULL END FROM DUAL WHERE XMLDATA."PONum" = 2100
/PurchaseOrder/PONum/text()	XMLElement("", XMLDATA."PONum")
/PurchaseOrder/Item	SELECT XMLAgg(XMLForest(value(it) AS "Item")) FROM table(XMLDATA."Item") it
/PurchaseOrder/Item/Part	SELECT XMLAgg(CASE WHEN *node_exists*[1](p.SYS_XDBPD$, 'Part') 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

[1] Pseudofunction *node_exists* is used for illustration only. It represents an Oracle XML DB implementation that uses its first argument, the PD column, to determine whether or not its second argument node exists. It returns true if so, and false if not.

Example 6–17 XPath Mapping for EXTRACT with Document Ordering Preserved

Using the mapping in Table 6–7, consider this query that extracts the PONum element, where the purchase order contains a part with price greater than 2000:

```
SELECT extract(OBJECT_VALUE, '/PurchaseOrder[Item/Part > 2000]/PONum')
  FROM purchaseorder_table;
```

This query would become something like the following:

```
SELECT (SELECT CASE WHEN node_exists(p.XMLDATA.SYS_XDBPD$, 'PONum')
                    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 Maintaining Document Order

If attribute SYS_XDBPD$ does not exist (that is, if the XML schema specifies maintainDOM="false"), then NULL scalar columns map to simpleType elements that do not exist. Hence you do not need to check for the node existence using attribute SYS_XDBPD$. Table 6–8 shows the mapping for function existsNode in the absence of SYS_XDBPD$.

Table 6–8 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
/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

XPath Rewrite for XMLSEQUENCE

You can use SQL function `XMLSequence` in conjunction with SQL functions `extract` and `table` to unnest XML collection values. When used with schema-based storage, these functions also get rewritten to access the underlying relational collection storage.

For example, this query obtains the price and part numbers of all items in a relational form:

```
SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/PONum') AS ponum,
       extractValue(value(it), '/Item/Part') AS part,
       extractValue(value(it), '/Item/Price') AS price
  FROM purchaseorder,
       table(XMLSequence(extract(OBJECT_VALUE, '/PurchaseOrder/Item'))) it;

PONUM PART                 PRICE
----- -------------------- ---------
      1001 9i Doc Set           2550
      1001 8i Doc Set            350
```

In this example, SQL function `extract` returns a fragment containing the list of `Item` elements. Function `XMLSequence` converts the fragment into a collection of `XMLType` values one for each `Item` element. Function `table` converts the elements of the collection into rows of `XMLType`. The XML data returned from `table` is used to extract the `Part` and the `Price` elements.

The applications of functions `extract` and `XMLSequence` are rewritten to a simple `SELECT` operation from the `item_nested` nested table.

```
EXPLAIN PLAN
  FOR SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/PONum') AS ponum,
             extractValue(value(it) , '/Item/Part') AS part,
             extractValue(value(it), '/Item/Price') AS price
        FROM purchaseorder,
             table(XMLSequence(extract(OBJECT_VALUE, '/PurchaseOrder/Item'))) it;

Explained
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
| Id  | Operation                     | Name             |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                  |
|   1 |  NESTED LOOPS                 |                  |
|   2 |   TABLE ACCESS FULL           | ITEM_NESTED      |
|   3 |   TABLE ACCESS BY INDEX ROWID | PURCHASEORDER    |
|*  4 |    INDEX UNIQUE SCAN          | SYS_C002973      |
------------------------------------------------------------------------------

Predicate Information (identified by operation id)
---------------------------------------------------
   4 - access("NESTED_TABLE_ID"="SYS_ALIAS_1"."SYS_NC0001100012$")
```

The `EXPLAIN PLAN` output shows that the optimizer is able to use a simple nested-loops join between nested table `item_nested` and table `purchaseorder`. You can also query the `Item` values further and create appropriate indexes on the nested table to speed up such queries.

For example, to search on the price to get all the expensive items, we could create an index on the `Price` column of the nested table. The following `EXPLAIN PLAN` uses a price index to obtain the list of items and then joins with table `purchaseorder` to obtain the `PONum` value.

```
CREATE INDEX price_index ON item_nested ("Price");

Index created.

EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE, '/PurchaseOrder/PONum') AS ponum,
         extractValue(value(it), '/Item/Part') AS part,
         extractValue(value(it), '/Item/Price') AS price
    FROM  purchaseorder,
          table(XMLSequence(extract(OBJECT_VALUE, '/PurchaseOrder/Item'))) it
    WHERE extractValue(value(it), '/Item/Price') > 2000;

Explained.

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 | PURCHASEORDER    |
|*  5 |    INDEX UNIQUE SCAN          | SYS_C002973      |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("ITEM_NESTED"."Price">2000)
   5 - access("NESTED_TABLE_ID"="SYS_ALIAS_1"."SYS_NC0001100012$")
```

XPath Rewrite for UPDATEXML

SQL function `updateXML` must satisfy the following conditions for it to use rewrite optimization:

- The `XMLType` argument must be based on a registered XML schema.
- The `XMLType` argument must also be the target of the `UPDATE` operation. For example:

  ```
  UPDATE purchaseorder_table SET OBJECT_VALUE = updateXML(OBJECT_VALUE,...);
  ```

- XPath arguments must all be different (no duplicates).
- XPath arguments must otherwise be rewritable, as described in "Which XPath Expressions Are Rewritten?" on page 6-4.
- XPath arguments must target only text nodes or attribute nodes.
- XPath arguments cannot target nodes that have default values (as defined in the XML schema).
- XPath arguments must not have a positional predicate (for example, `foo[2]`).
- If an XPath argument has a predicate, the predicate must not come before a collection.

  For example, `/PurchaseOrder/LineItems[@MyAtt="3"]/LineItem` will not be rewritten, because the predicate occurs before the `LineItem` collection. (This assumes an XML schema where `LineItems` has an attribute `MyAtt`.)
- If an XPath argument references a collection, the collection must be stored as a separate table (varray or nested table), not out of line (`REF` storage) or in line.
- If an XPath argument references a collection, the collection must not be scalar (`simpleType` with `maxOccurs > 1`).

> **See Also:** Example 6–2, Example 6–3, Example 3–34, and Example 3–34 for examples of rewriting `updateXML` expressions

XPath Rewrite for INSERTCHILDXML and DELETEXML

SQL function `deleteXML` must satisfy the following conditions for it to use rewrite optimization:

- The `XMLType` argument must be based on a registered XML schema.
- The `XMLType` argument must also be the target of the `UPDATE` operation. For example:

  ```
  UPDATE purchaseorder_table SET OBJECT_VALUE = updateXML(OBJECT_VALUE,...);
  ```

- XPath arguments must otherwise be rewritable, as described in "Which XPath Expressions Are Rewritten?" on page 6-4.
- The XPath argument must not have a positional predicate (for example, `foo[2]`).
- If the XPath argument has a predicate, the predicate must not come before a collection.

  For example, `/PurchaseOrder/LineItems[@MyAtt="3"]/LineItem` will not be rewritten, because the predicate occurs before the `LineItem` collection. (This assumes an XML schema where `LineItems` has an attribute `MyAtt`.)

- The XPath argument must target an unbounded collection (element with `maxOccurs = "unbounded"`).
- The XPath argument must not target a `choice` of collections, as defined in the XML schema.
- The parent of the targeted collection must be defined in the XML schema with annotation `maintainDOM = "false"`.
- If an XPath argument references a collection, the collection must be stored as a separate table (varray or nested table), not out of line (`REF` storage) or in line.
- If an XPath argument references a collection, the collection must not be scalar (`simpleType` with `maxOccurs > 1`).

7

XML Schema Storage and Query: Advanced

This chapter describes advanced techniques for storing structured XML schema-based `XMLType` objects.

See Also:

- Chapter 5, "XML Schema Storage and Query: Basic" for basic information on using XML Schema with Oracle XML DB
- Chapter 6, "XPath Rewrite" for information on the optimization of XPath expressions in Oracle XML DB
- Chapter 8, "XML Schema Evolution" for information on updating an XML schema after you have registered it with Oracle XML DB
- Appendix A, "XML Schema Primer" for an introduction to XML Schema

This chapter contains these topics:

Generating XML Schemas with DBMS_XMLSCHEMA.GENERATESCHEMA

An XML schema can be generated from an object-relational type automatically using a default mapping. PL/SQL functions `generateSchema` and `generateSchemas` in

package DBMS_XMLSCHEMA take in a string that has the object type name and another that has the Oracle XML DB XML schema.

- Function generateSchema returns an XMLType containing an XML schema. It can optionally generate XML schema for all types referenced by the given object type or restricted only to the top-level types.
- Function generateSchemas is similar, except that it returns an XMLSequenceType value. This is a varray of XMLType instances, each of which is an XML schema that corresponds to a different namespace. It also takes an additional optional argument, 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 5-15

Example 7–1 Generating an XML Schema with Function GENERATESCHEMA

For example, given the object type:

```
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 annotations 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>
```

Note: As always:

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

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

Adding Unique Constraints to the Parent Element of an Attribute

After creating an `XMLType` table based on an XML schema, how can you add a unique constraint to the parent element of an attribute? You may, for example, want to create a unique key based on an attribute of an element that repeats itself (a collection).

To create constraints on elements that can occur more than once, store the varray as a table. This is also known as **Ordered Collections in Tables** (OCT). You can then create constraints on the OCT. Example 7–2 shows how the attribute `No` of element `<PhoneNumber>` can appear more than once, and how a unique constraint can be added to ensure that the same phone 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 7–2 Adding a Unique Constraint to the Parent Element of an Attribute

```
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:

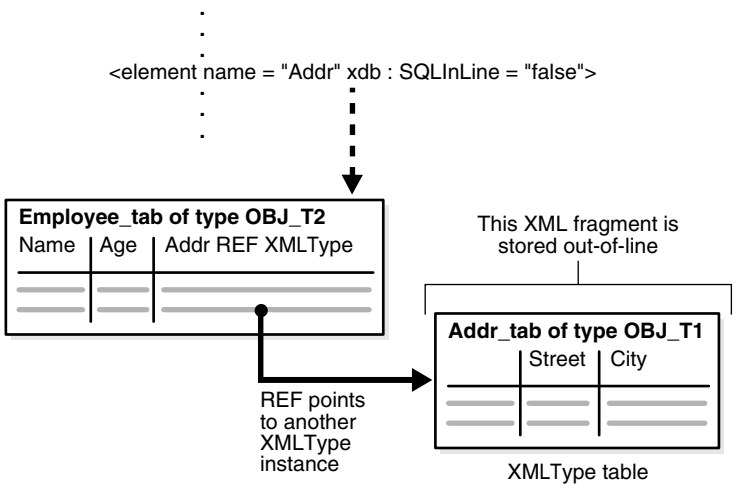
```
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C002136) violated
```

Setting Attribute SQLInline to false for Out-of-Line Storage

By default, a child 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 7–1 illustrates the mapping of a `complexType` to SQL for out-of-line storage.

Figure 7–1 Mapping complexType to SQL for Out-of-Line Storage



Example 7–3 complexType Mapping - Setting SQLInline 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;
/
```

When registering this XML schema, Oracle XML DB generates the following types:

```
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 of the employees, and contains an object reference that points to the address values stored in table addr_tab.

The advantage of this model is that it lets you query the out-of-line table (addr_tab) directly, to look up the address information. For example, if you want to obtain the distinct city information for all employees, you can query 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>'));

-- Table 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 model is that in order to obtain the entire Employee element you need to look up an additional table for the address.

XPath Rewrite for Out-Of-Line Tables

XPath expressions that involve elements stored out of line are rewritten. The query involves a join with the out-of-line table. For example, the following EXPLAIN PLAN shows how a query involving elements Employee and Addr is handled.

```
EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE,
                      '/x:Employee/Name',
                      'xmlns:x="http://www.oracle.com/emp.xsd"')
    FROM emp_tab
    WHERE existsNode(OBJECT_VALUE,
                     '/x:Employee/Addr[City="San Francisco"]',
                     'xmlns:x="http://www.oracle.com/emp.xsd"') = 1;

SELECT PLAN_TABLE_OUTPUT
  FROM table(DBMS_XPLAN.display('plan_table', NULL, 'serial'))
/

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 of 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 many entries in the `addr_tab`, then you can make this query more efficient by creating an index on column `City`.

```
CREATE INDEX addr_city_idx
  ON addr_tab (extractValue(OBJECT_VALUE, '/Addr/City'));
```

The `EXPLAIN PLAN` for the previous query now uses the `addr_city_idx` index.

```
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')
```

See Also: Chapter 6, "XPath Rewrite"

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 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;
/
```

When registering this XML schema, Oracle XML DB generates the following:

```
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;
```

Note: As always:

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

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

The employee type (`emp_t2`) contains a varray of `REF` values to address instead of a single `REF` attribute as in the previous XML schema. By default, this varray of `REF` values is stored in line in the employee table (`emp_tab2`). 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 from the employee table, as there is no way to efficiently join back from the address table. The following example shows the explain 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(OBJECT_VALUE,
                      '/x:Employee/Name',
                      'xmlns:x="http://www.oracle.com/emp.xsd"') AS name,
         extractValue(value(ad), '/Addr/Street') AS street
    FROM
      emp_tab2,
      table(XMLSequence(extract(OBJECT_VALUE,
                                '/x:Employee/Addr',
                                'xmlns:x="http://www.oracle.com/emp.xsd"'))) ad
    WHERE extractValue(value(ad), '/Addr/City') = 'San Francisco';

Explained.

SELECT PLAN_TABLE_OUTPUT
  FROM table(DBMS_XPLAN.display('plan_table', NULL, 'serial'));

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.

Out-of-Line Storage: Using an Intermediate Table to Store the List of References

In cases where the number of employees is large, a full table scan of the `emp_tab2` table can be too expensive. The correct approach in this case 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 (out of line) and creating an index on the `REF` values in that table. This lets Oracle XML DB query the address table, obtain an object reference (`REF`) to the relevant row, join it with the intermediate table storing the list of `REF` values, and join that table back with the employee table.

The intermediate table can be created by setting `xdb:storeVarrayAsTable="true"` in the XMLSchema definition. This forces the schema registration to store all varray values as separate tables.

Note: Annotation `storeVarrayAsTable="true"` causes element collections to be persisted as rows in an index-organized table (IOT). Oracle Text does not support IOTs. Do *not* use this annotation if you will need to use Oracle Text indexes for text-based `ora:contains` searches over a collection of elements. See "ora:contains Searches Over a Collection of Elements" on page 10-23. To provide for searching with Oracle Text indexes:

1. Set `genTables="false"` during schema registration.
2. Create the necessary tables *manually*, without using the clause `ORGANIZATION INDEX OVERFLOW`, so that the tables will be *heap-organized* instead of index-organized (IOT).

```
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;
/
```

Note: As always:

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

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

In addition to creating types `addr_t3` and `emp_t3` and 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==

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 table `emp_tab3_reflist` will only point to objects in table `addr_tab3`, 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);

CREATE INDEX city_idx ON addr_tab3 p (extractValue(OBJECT_VALUE, '/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(OBJECT_VALUE,
                      '/x:Employee/Name',
                      'xmlns:x="http://www.oracle.com/emp.xsd"') AS name,
         extractValue(value(ad), '/Addr/Street') AS street
    FROM emp_tab3,
         table(XMLSequence(extract(OBJECT_VALUE,
                                   '/x:Employee/Addr',
                                   'xmlns:x="http://www.oracle.com/emp.xsd"'))) ad
    WHERE extractValue(value(ad), '/Addr/City') = 'San Francisco';

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"="SYS_ALIAS_1"."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 either, then Oracle XML DB raises an error.

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:

`http://xmlns.oracle.com/xdb/schemas/`***<database-user>***`/<schemaURL-minus-protocol>`

Example 7–4 Using a 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`) value or a Binary Large Object (`BLOB`) value, as shown in Figure 7–2. 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 7–5 Oracle XML DB XML Schema: Mapping complexType XML Fragments to LOBs

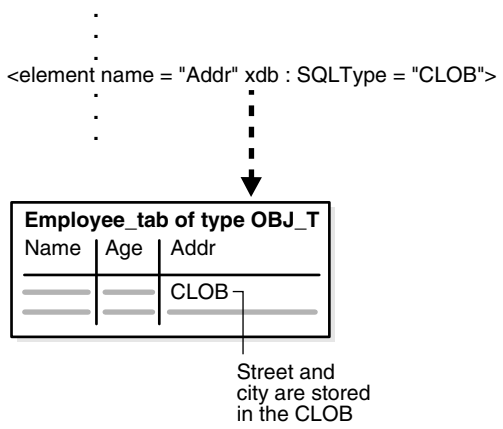
In the following example, the XML schema specifies that the XML fragment element `Addr` uses the attribute `SQLType="CLOB"`:

```
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_T">
         <sequence>
           <element name="Name" type="string"/>
           <element name="Age" type="decimal"/>
           <element name="Addr" xdb:SQLType="CLOB">
             <complexType >
               <sequence>
                 <element name="Street" type="string"/>
                 <element name="City" type="string"/>
               </sequence>
             </complexType>
           </element>
         </sequence>
       </complexType>
     </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema('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 7–2 Mapping complexType XML Fragments to Character Large Objects (CLOBs)



complexType Extensions and Restrictions in Oracle XML DB

In XML Schema, `complexType` values are declared based on `complexContent` and `simpleContent`.

- `simpleContent` is declared as an extension of `simpleType`.
- `complexContent` is declared as one of the following:
  - Base type
  - `complexType` extension
  - `complexType` restriction

This section describes the Oracle XML DB extensions and restrictions to `complexType`.

complexType Declarations in XML Schema: Handling Inheritance

For `complexType`, Oracle XML DB handles inheritance in the XML schema as follows:

- ***For 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 7–6 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, so all SQL object types are created as types that are *not* final.

```
CREATE TYPE addr_t AS OBJECT(SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                             "street" VARCHAR2(4000),
                             "city" VARCHAR2(4000)) NOT FINAL;
CREATE TYPE usaddr_t UNDER addr_t ("zip" VARCHAR2(4000)) NOT FINAL;
CREATE TYPE intladdr_t UNDER addr_t ("country" VARCHAR2(4000)) FINAL;
```

Example 7–7 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 7–8 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 7–9 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));
```

Oracle XPath Extension Functions to Examine Type Information

Oracle XML DB supports XML schema-based data, where elements and attributes have XML Schema datatype information associated with them. However, XPath 1.0 is not aware of datatype information. Oracle XML DB extends XPath 1.0 with the following Oracle extension functions to support examining datatype information:

- instanceof
- instanceof-only

These XPath functions are in namespace http://xmlns.oracle.com/xdb, which has the predefined prefix ora.

An element is an **instance** of a specified XML Schema datatype 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 is a type that extends or restricts *T*, or extends or restricts another subtype of *T*.

For XPath expressions involving XML schema-based data, you can use Oracle XPath function ora:instanceof-only to restrict the result set to nodes of a certain datatype, and ora:instanceof to restrict the result set to nodes of a certain datatype or its subtypes. For *non*-schema-based XML data, elements and attributes do not have datatype information, so these functions return *false* for non-schema-based data.

ora:instanceof-only XPath Function

Syntax

```
ora:instanceof-only(nodeset-expr, typename [, schema-url])
```

On XML schema-based data, `ora:instanceof-only` evaluates XPath expression *`nodeset-expr`* and determines the XML Schema datatype for each of the resultant nodes. Expression *`nodeset-expr`* is typically a *relative* XPath expression. If the datatype of *any* of the nodes exactly matches datatype *`typename`* (a string), which can be qualified with a namespace prefix, then `instanceof-only` returns true; otherwise, it returns false. It returns false for non-schema-based data.

Optional parameter *`schema-url`* (a string) indicates the schema location URL for the datatype to be matched. If specified, then the *`schema-url`* parameter must specify the location of the the XML schema that defines the node datatype. If *`schema-url`* is not specified, the schema location of the node is not checked.

Example 7–10 Using ora:instanceof-only

The following query selects the `Name` attributes of `AE` children of element `Person` that are of datatype `PersonType` (subtypes of `PersonType` are not matched).

```
SELECT extract(OBJECT_VALUE,
               '/p9:Person[ora:instanceof-only(AE,"p9:PersonType")]/AE/Name',
               'xmlns:p9="person9.xsd" xmlns:ora="http://xmlns.oracle.com/xdb"')
  FROM po_table;
```

ora:instanceof XPath Function

Syntax

```
ora:instanceof(nodeset-expr, typename [, schema-url])
```

Oracle XPath function `ora:instanceof` is similar to `ora:instanceof-only`, but it also returns true if the datatype of any of the matching nodes exactly matches a *subtype* of datatype *`typename`*.

Example 7–11 Using ora:instanceof

The following query selects the `Name` attributes of `AE` children of element `Person` that are of datatype `PersonType` or of one of its subtypes.

```
SELECT extract(OBJECT_VALUE,
               '/p9:Person[ora:instanceof(AE,"p9:PersonType")]/AE/Name',
               'xmlns:p9="person9.xsd" xmlns:ora="http://xmlns.oracle.com/xdb"')
  FROM po_table;
```

The schema-location parameter is typically used in a heterogeneous XML Schema scenario. Heterogeneous XML schema-based data can be present in a single table. If your scenario involves a schema-based table, consider omitting the schema location parameter.

Consider a non-schema-based table of `XMLType`. Each row in the table is an XML document. Suppose that each row contains data for which XML schema information has been specified. If the data in the table is converted to XML schema-based data through a subsequent operation, then the rows in the table could pertain to different XML schemas. In such a case, you can specify not only the name and the namespace of the datatype to be matched, but also the schema-location URL.

Example 7–12 Using ora:instanceof with Heterogeneous XML Schema-Based Data

In the non-schema-based table `non_sch_p_tab`, the following query matches elements of type `PersonType` that pertain to XML schema `person9.xsd`.

```
SELECT extract(
         createSchemaBased(
           OBJECT_VALUE),
           '/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;
```

XML Schema: 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 can *contain* an element based on that same `complexType` , or a global element can contain a reference to itself. In both cases the reference can be direct or indirect. This kind of structure allows for instance documents where the element in question can appear an infinite number of times in a recursive hierarchy.

Example 7–13 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:defaultTable="DESCENDANT_TABLE"/>
    </xs:sequence>
    <xs:attribute name="personName" use="required" xdb:SQLName="PERSON_NAME">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:maxLength value="20"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:schema>
```

The XML schema in Example 7–13 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 XML Schema Dependencies Set GenTables Parameter to TRUE

Oracle XML DB supports XML schemas that define this kind of structure. It does this by detecting the cycles, breaking them, and storing the recursive elements as rows in a separate `XMLType` table that is created during XML schema registration.

Consequently, it is important to ensure that 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

SQL object types do not allow cycles. Cycles in the XML schema are broken while generating the object types, by introducing a `REF` attribute at the point at which the cycle would be completed. Thus, part of the data is stored out of line, yet it is still retrieved as part of the parent XML document.

Example 7–14 XML Schema: Cycling Between complexTypes

XML schemas permit cycling between definitions of `complexTypes`. Figure 7–3 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:

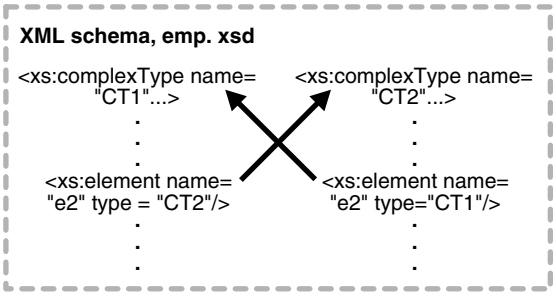
```
DECLARE
  doc VARCHAR2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:xdb="http://xmlns.oracle.com/xdb">
       <xs:complexType name="CT1" xdb:SQLType="CT1">
         <xs:sequence>
           <xs:element name="e1" type="xs:string"/>
           <xs:element 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 do support **weak cycles**, that is, cycles involving `REF` (reference) attributes. Cyclic XML schema definitions are mapped to SQL object types in such a way that cycles are avoided by forcing `SQLInline="false"` at the appropriate points. This creates a weak SQL cycle.

For the 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 7–3 Cross Referencing Between Different complexTypes in the Same XML Schema



Example 7–15 XML Schema: Cycling Between complexTypes, Self-Reference

Another example of a cyclic `complexType` involves the declaration of the `complexType` having a reference to itself. In this example, type `<SectionT>` 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. It is a varray of `REF` references to `XMLType` values, to avoid forming a cycle of SQL objects.

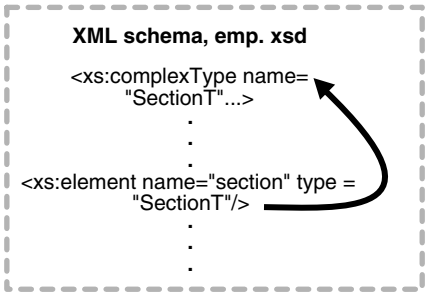
How a complexType Can Reference Itself

Assume that your XML schema, identified by "`http://www.oracle.com/PO.xsd`", has been registered. An `XMLType` table, `purchaseorder`, can then be created to store instances conforming to element `PurchaseOrder` of this XML schema, in an object-relational format:

```
CREATE TABLE purchaseorder OF XMLType
   ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

Figure 7–4 illustrates schematically how a `complexType` can reference itself.

Figure 7–4 complexType Self Referencing Within an XML Schema



See Also: "Cyclical References Between XML Schemas" on page 7-22

Hidden columns are created that correspond to the object type to which the `PurchaseOrder` element has been mapped. In addition, an `XMLExtra` object column is created, to store top-level instance data such as namespace declarations.

Note: `XMLDATA` is a pseudo-attribute of `XMLType` that enables direct access to the underlying object column. See Chapter 4, "XMLType Operations".

Cyclical References Between XML Schemas

XML schemas can depend on each other in such a way that they cannot be registered one after the other in the usual manner. Examples of such XML schemas follow:

Example 7–16 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;
/
```

The XML schemas can each be used to create a table:

```
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 of these XML schemas, which depend on each other, you must use the `FORCE` parameter in `DBMS_XMLSCHEMA.registerSchema` as follows:

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.

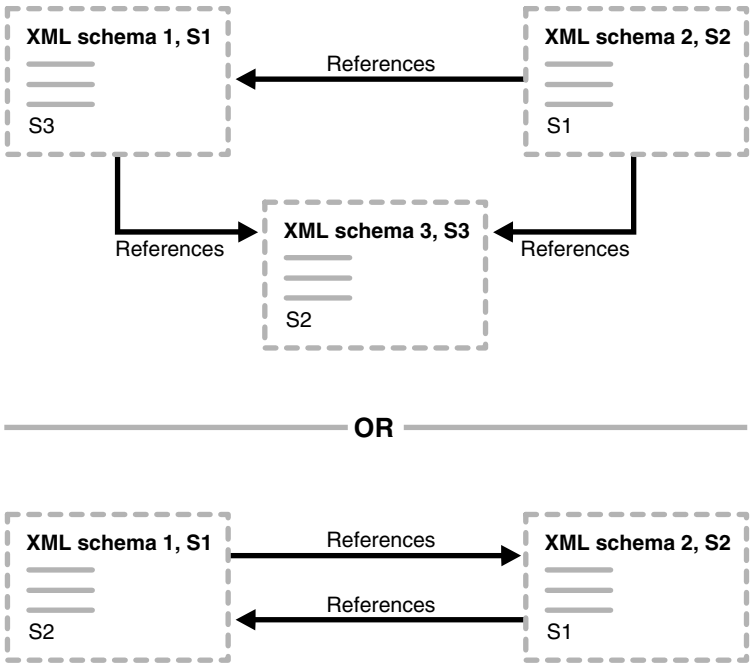
2. Register `s2.xsd` in `FORCE` mode:

   ```
   DBMS_XMLSCHEMA.registerSchema("s2.xsd", "<schema ..", ..., FORCE => TRUE)
   ```

   The second operation automatically compiles `s1.xsd` and makes both XML schemas valid.

See Figure 7–5. The preceding example is illustrated in the lower half of the figure.

Figure 7–5 Cyclical References Between XML Schemas



Guidelines for Using XML Schema with Oracle XML DB

This section describes guidelines for using XML schema with Oracle XML DB:

Using Bind Variables in XPath Expressions

When you use a bind variable, 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.

XPath Rewrite with Bind Variables

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-quotes (') or double-quotes (") inside the XPath string. The following example illustrates the use of the bind variable with XPath rewrite.

Example 7–17 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;
/

-- Create table corresponding to the Employee element
CREATE TABLE emp_bind_tab OF XMLType
  ELEMENT "bindtest.xsd#Employee";

-- 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
    WHERE existsNode(OBJECT_VALUE, '/Employee[EmployeeId="'||:1||'"] ') = 1;

SELECT PLAN_TABLE_OUTPUT
  FROM table(DBMS_XPLAN.display('plan_table', NULL, 'serial'))
/

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("SYS_ALIAS_1"."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 XPath rewrite, 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 is itself 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
WHERE extractValue(
        OBJECT_VALUE,
        '/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 SQL functions `extractValue`, `existsNode`, and so on, it looks at the XPath bind variables to check if they are really constants. If so, it uses them and rewrites the query. Hence there is a large difference depending on where the bind variable is used.

Constraints on Repetitive Elements in Schema-Based XML

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 7–18 Creating Constraints on Repetitive Elements in a Schema-Based Table

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 ADD 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
```

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 using either protocols (FTP, HTTP(S), or DAV) or PL/SQL function `DBMS_XDB.createResource`.
- The document is XML schema-based and contains large collections (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 the table properties appropriately in the element definition
  - Setting `xdb:storeVarrayAsTable="true"` in the schema definition, which turns this storage option on for all collections of the schema

---

Note: Annotation `storeVarrayAsTable="true"` causes element collections to be persisted as rows in an index-organized table (IOT). Oracle Text does not support IOTs. Do *not* use this annotation if you will need to use Oracle Text indexes for text-based `ora:contains` searches over a collection of elements. See "ora:contains Searches Over a Collection of Elements" on page 10-23. To provide for searching with Oracle Text indexes:

1. Set `genTables="false"` during schema registration.
2. Create the necessary tables *manually*, without using the clause `ORGANIZATION INDEX OVERFLOW`, so the tables will be *heap-organized* instead of index-organized (IOT).

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 XML 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.

8

XML Schema Evolution

This chapter describes how you can update your XML schema after you have registered it with Oracle XML DB. XML schema evolution is the process of updating your registered XML schema.

This chapter contains these topics:

Overview of XML Schema Evolution

Oracle XML DB supports the W3C XML Schema recommendation. XML instance documents that conform to an XML schema can be stored and retrieved using SQL and protocols such as FTP, HTTP(S), and WebDAV. In addition to specifying the structure of XML documents, XML schemas determine the mapping between XML and object-relational storage.

See Also: Chapter 5, "XML Schema Storage and Query: Basic"

Oracle XML DB supports XML schema evolution by providing PL/SQL procedure `copyEvolve` as part of PL/SQL package `DBMS_XMLSCHEMA`. Procedure `copyEvolve` copies existing instance documents to temporary `XMLType` tables to back them up, drops the old version of the XML schema (which also deletes the associated instance documents), registers the new version, and copies the backed-up instance documents to new `XMLType` tables.

With procedure `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 that are dependent on the XML schema, delete the old XML schema, and register the new XML schema at the same URL.

Limitations of Procedure DBMS_XMLSCHEMA.COPYEVOLVE

The following are the limitations of procedure `DBMS_XMLSCHEMA.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 procedure copyEvolve completes executing. See the section on "Top-Level Element Name Changes" on page 8-2 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.

Guidelines for Using Procedure DBMS_XMLSCHEMA.COPYEVOLVE

Here are some guidelines for using procedure DBMS_XMLSCHEMA.copyEvolve:

1. First, identify the XML schemas that are dependent on the XML schema that is to be evolved. You can acquire the URLs of the dependent XML schemas using the following query, where *schema_to_be_evolved* is the schema to be evolved, and *owner_of_schema_to_be_evolved* is its owner (database user).

```
SELECT dxs.SCHEMA_URL
    FROM DBA_DEPENDENCIES dd, DBA_XML_SCHEMAS dxs
    WHERE dd.REFERENCED_NAME = (SELECT INT_OBJNAME
                                  FROM DBA_XML_SCHEMAS
                                  WHERE SCHEMA_URL = schema_to_be_evolved
                                    AND OWNER = schema_to_be_evolved)
      AND dxs.OWNER = owner_of_schema_to_be_evolved
      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 procedure DBMS_XMLSCHEMA.copyEvolve, specifying the XML schema URLs, new schemas, and transformations.

Top-Level Element Name Changes

Procedure DBMS_XMLSCHEMA.copyEvolve assumes that top-level elements have not been dropped and that their names have not been changed in the new XML schemas. If there are such changes in your new XML schemas, you can call procedure 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 "Procedure DBMS_XMLSCHEMA.COPYEVOLVE: Parameters and Errors" on page 8-10 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, then procedure DBMS_XMLSCHEMA.copyEvolve waits for these sessions to release the locks so that it can acquire an exclusive lock. However this lock is released immediately to allow the rest of the process to continue.

Rollback When Procedure DBMS_XMLSCHEMA.COPYEVOLVE Raises an Error

Procedure DBMS_XMLSCHEMA.copyEvolve() either completely succeeds or raises an error, in which case it attempts to 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%';
```

Privileges Needed for XML Schema Evolution

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 database schemas, you need privileges such as the following:

- CREATE ANY TABLE
- CREATE ANY INDEX
- SELECT ANY TABLE
- UPDATE ANY TABLE
- INSERT ANY TABLE
- DELETE ANY TABLE
- DROP ANY TABLE

- ALTER ANY TABLE
- DROP ANY INDEX

To avoid having to grant all these privileges to the schema owner, Oracle recommends that the evolution be performed by a DBA if there are XML schema-based XMLType table or columns in other users' database schemas.

Revised Purchase-Order XML Schema

Example 8–1 shows a *partial* listing of a revised version of the purchase-order XML schema of Example 3–8. See Example D–2 on page D-16 for the *complete* revised schema listing. Text that is in **bold face** here is new or significantly different from that in the original schema (Example 3–8).

Example 8–1 Revised Purchase-Order XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0">
  <xs:element
    name="PurchaseOrder" type="PurchaseOrderType"
    xdb:defaultTable="PURCHASEORDER"
    xdb:columnProps=
      "CONSTRAINT purchaseorder_pkey PRIMARY KEY (XMLDATA.reference),
       CONSTRAINT valid_email_address FOREIGN KEY (XMLDATA.userid)
         REFERENCES hr.employees (EMAIL)"
    xdb:tableProps=
      "VARRAY XMLDATA.ACTIONS.ACTION STORE AS TABLE ACTION_TABLE
        ((CONSTRAINT action_pkey PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       VARRAY XMLDATA.LINEITEMS.LINEITEM STORE AS TABLE LINEITEM_TABLE
        ((constraint LINEITEM_PKEY primary key (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       lob (XMLDATA.NOTES) STORE AS (ENABLE STORAGE IN ROW STORAGE(INITIAL 4K NEXT 32K))"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
      <xs:element name="Reject" type="RejectionType" minOccurs="0" xdb:SQLName="REJECTION"/>
      <xs:element name="Requestor" type="RequestorType" xdb:SQLName="REQUESTOR"/>
      <xs:element name="User" type="UserType" xdb:SQLName="USERID"/>
      <xs:element name="CostCenter" type="CostCenterType" xdb:SQLName="COST_CENTER"/>
      <xs:element name="BillingAddress" type="AddressType" minOccurs="0"
                  xdb:SQLName="BILLING_ADDRESS"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                  xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                  xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
      <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
      <xs:element name="Notes" type="NotesType" minOccurs="0" xdb:SQLType="CLOB"
                  xdb:SQLName="NOTES"/>
    </xs:sequence>
    <xs:attribute name="Reference" type="ReferenceType" use="required" xdb:SQLName="REFERENCE"/>
    <xs:attribute name="DateCreated" type="xs:dateTime" use="required"
                  xdb:SQLType="TIMESTAMP WITH TIME ZONE"/>
  </xs:complexType>
  <xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
    <xs:sequence>
      <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded" xdb:SQLName="LINEITEM"
                  xdb:SQLCollType="LINEITEM_V"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
    <xs:sequence>
      <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
      <xs:element name="Quantity" type="quantityType"/>
```

```
    </xs:sequence>
    <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                  xdb:SQLType="NUMBER"/>
  </xs:complexType>
  <xs:complexType name="PartType" xdb:SQLType="PART_T">
    <xs:simpleContent>
      <xs:extension base="UPCCodeType">
        <xs:attribute name="Description" type="DescriptionType" use="required"
                      xdb:SQLName="DESCRIPTION"/>
        <xs:attribute name="UnitCost" type="moneyType" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:simpleType name="ReferenceType">
    <xs:restriction base="xs:string">
      <xs:minLength value="18"/>
      <xs:maxLength value="30"/>
    </xs:restriction>
  </xs:simpleType>

. . .

  <xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
    <xs:all>
      <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
      <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
      <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
    <xs:sequence>
      <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
      <xs:choice>
        <xs:element name="address" type="AddressType" minOccurs="0"/>
        <xs:element name="fullAddress" type="FullAddressType" minOccurs="0"
                    xdb:SQLName="SHIP_TO_ADDRESS"/>
      </xs:choice>
      <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
    </xs:sequence>
  </xs:complexType>

. . .

  <xs:simpleType name="NameType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="20"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="FullAddressType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>

. . .

  <xs:simpleType name="DescriptionType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="AddressType" xdb:SQLType="ADDRESS_T">
    <xs:sequence>
```

```
      <xs:element name="StreetLine1" type="StreetType"/>
      <xs:element name="StreetLine2" type="StreetType" minOccurs="0"/>
      <xs:element name="City" type="CityType"/>
      <xs:choice>
        <xs:sequence>
          <xs:element name="State" type="StateType"/>
          <xs:element name="ZipCode" type="ZipCodeType"/>
        </xs:sequence>
        <xs:sequence>
          <xs:element name="Province" type="ProvinceType"/>
          <xs:element name="PostCode" type="PostCodeType"/>
        </xs:sequence>
        <xs:sequence>
          <xs:element name="County" type="CountyType"/>
          <xs:element name="Postcode" type="PostCodeType"/>
        </xs:sequence>
      </xs:choice>
      <xs:element name="Country" type="CountryType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="StreetType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="128"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CityType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="64"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="StateType">
    <xs:restriction base="xs:string">
      <xs:minLength value="2"/>
      <xs:maxLength value="2"/>
      <xs:enumeration value="AK"/>
      <xs:enumeration value="AL"/>
      <xs:enumeration value="AR"/>

. . . -- A value for each US state abbreviation

      <xs:enumeration value="WY"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ZipCodeType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{5}"/>
      <xs:pattern value="\d{5}-\d{4}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CountryType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="64"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CountyType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="32"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="PostCodeType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
```

```
      <xs:maxLength value="12"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ProvinceType">
    <xs:restriction base="xs:string">
      <xs:minLength value="2"/>
      <xs:maxLength value="2"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NotesType">
    <xs:restriction base="xs:string">
      <xs:maxLength value="32767"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="UPCCodeType">
    <xs:restriction base="xs:string">
      <xs:minLength value="11"/>
      <xs:maxLength value="14"/>
      <xs:pattern value="\d{11}"/>
      <xs:pattern value="\d{12}"/>
      <xs:pattern value="\d{13}"/>
      <xs:pattern value="\d{14}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Note: As always:

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

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

Style Sheet to Update Existing Instance Documents

After you modify a registered XML schema, you must also update any existing XML instance documents that used the old version of the schema. You do this by applying an XSLT style sheet to each of the instance documents. The style sheet represents the difference between the old and new schemas.

Example 8–2 is a style sheet, in file `evolvePurchaseOrder.xsl`, that transforms existing purchase-order documents that use the old schema, so they will use the new schema instead.

Example 8–2 evolvePurchaseOrder.xsl: Style Sheet to Update Instance Documents

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsl:output method="xml" encoding="UTF-8"/>
  <xsl:template match="/PurchaseOrder">
    <PurchaseOrder>
      <xsl:attribute name="xsi:noNamespaceSchemaLocation">
        http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd
      </xsl:attribute>
      <xsl:for-each select="Reference">
```

```
  <xsl:attribute name="Reference">
    <xsl:value-of select="."/>
  </xsl:attribute>
</xsl:for-each>
<xsl:variable name="V264_394" select="'2004-01-01T12:00:00.000000-08:00'"/>
<xsl:attribute name="DateCreated">
  <xsl:value-of select="$V264_394"/>
</xsl:attribute>
<xsl:for-each select="Actions">
  <Actions>
    <xsl:for-each select="Action">
      <Action>
        <xsl:for-each select="User">
          <User>
            <xsl:value-of select="."/>
          </User>
        </xsl:for-each>
        <xsl:for-each select="Date">
          <Date>
            <xsl:value-of select="."/>
          </Date>
        </xsl:for-each>
      </Action>
    </xsl:for-each>
  </Actions>
</xsl:for-each>
<xsl:for-each select="Reject">
  <Reject>
    <xsl:for-each select="User">
      <User>
        <xsl:value-of select="."/>
      </User>
    </xsl:for-each>
    <xsl:for-each select="Date">
      <Date>
        <xsl:value-of select="."/>
      </Date>
    </xsl:for-each>
    <xsl:for-each select="Comments">
      <Comments>
        <xsl:value-of select="."/>
      </Comments>
    </xsl:for-each>
  </Reject>
</xsl:for-each>
<xsl:for-each select="Requestor">
  <Requestor>
    <xsl:value-of select="."/>
  </Requestor>
</xsl:for-each>
<xsl:for-each select="User">
  <User>
    <xsl:value-of select="."/>
  </User>
</xsl:for-each>
<xsl:for-each select="CostCenter">
  <CostCenter>
    <xsl:value-of select="."/>
  </CostCenter>
</xsl:for-each>
<ShippingInstructions>
  <xsl:for-each select="ShippingInstructions">
    <xsl:for-each select="name">
      <name>
        <xsl:value-of select="."/>
      </name>
```

```
        </xsl:for-each>
      </xsl:for-each>
      <xsl:for-each select="ShippingInstructions">
        <xsl:for-each select="address">
          <fullAddress>
            <xsl:value-of select="."/>
          </fullAddress>
        </xsl:for-each>
      </xsl:for-each>
      <xsl:for-each select="ShippingInstructions">
        <xsl:for-each select="telephone">
          <telephone>
            <xsl:value-of select="."/>
          </telephone>
        </xsl:for-each>
      </xsl:for-each>
    </ShippingInstructions>
    <xsl:for-each select="SpecialInstructions">
      <SpecialInstructions>
        <xsl:value-of select="."/>
      </SpecialInstructions>
    </xsl:for-each>
    <xsl:for-each select="LineItems">
      <LineItems>
        <xsl:for-each select="LineItem">
          <xsl:variable name="V22" select="."/>
          <LineItem>
            <xsl:for-each select="@ItemNumber">
              <xsl:attribute name="ItemNumber">
                <xsl:value-of select="."/>
              </xsl:attribute>
            </xsl:for-each>
            <xsl:for-each select="$V22/Part">
              <xsl:variable name="V24" select="."/>
              <xsl:for-each select="@Id">
                <Part>
                  <xsl:for-each select="$V22/Description">
                    <xsl:attribute name="Description">
                      <xsl:value-of select="."/>
                    </xsl:attribute>
                  </xsl:for-each>
                  <xsl:for-each select="$V24/@UnitPrice">
                    <xsl:attribute name="UnitCost">
                      <xsl:value-of select="."/>
                    </xsl:attribute>
                  </xsl:for-each>
                  <xsl:value-of select="."/>
                </Part>
              </xsl:for-each>
            </xsl:for-each>
            <xsl:for-each select="$V22/Part">
              <xsl:for-each select="@Quantity">
                <Quantity>
                  <xsl:value-of select="."/>
                </Quantity>
              </xsl:for-each>
            </xsl:for-each>
          </LineItem>
        </xsl:for-each>
      </LineItems>
    </xsl:for-each>
  </PurchaseOrder>
 </xsl:template>
</xsl:stylesheet>
```

Procedure DBMS_XMLSCHEMA.COPYEVOLVE: Parameters and Errors

Here is the signature of procedure `DBMS_XMLSCHEMA.copyEvolve`:

```
procedure copyEvolve(schemaURLs       IN XDB$STRING_LIST_T,
                     newSchemas       IN XMLSequenceType,
                     transforms       IN XMLSequenceType := NULL,
                     preserveOldDocs  IN BOOLEAN := FALSE,
                     mapTabName       IN VARCHAR2 := NULL,
                     generateTables   IN BOOLEAN := TRUE,
                     force            IN BOOLEAN := FALSE,
                     schemaOwners     IN XDB$STRING_LIST_T := NULL);
```

Table 8–1 describes the individual parameters. Table 8–2 describes the errors associated with the procedure.

Table 8–1 Parameters of Procedure DBMS_XMLSCHEMA.COPYEVOLVE

Parameter	Description
`schemaURLs`	Varray of URLs of XML schemas to be evolved (varray of `VARCHAR2(4000)`. This should include the dependent schemas as well. Unless the force parameter is TRUE, the URLs should be in the dependency order, that is, if URL A comes before URL B in the varray, then schema A should not be dependent on schema B but schema B may be dependent on schema A.
`newSchemas`	Varray of new XML schema documents (`XMLType` instances). Specify this in exactly the same order as the corresponding URLs. If no change is necessary in an XML schema, provide the unchanged schema.
`transforms`	Varray of XSL documents (`XMLType` instances) that 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 "Using Procedure DBMS_XMLSCHEMA.COPYEVOLVE".
`mapTabName`	Specifies the name of table that maps old `XMLType` table or column names to names of corresponding temporary tables.
`generateTables`	By default this parameter is TRUE; if 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 not be `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 8–2 Errors Associated with Procedure DBMS_XMLSCHEMA.COPYEVOLVE

Error Number and Message	Cause	Action
30942 XML Schema Evolution error for schema '<schema_url>' table "<owner_name>.<table_name>" column '<column_name>'	The given `XMLType` table or column that conforms to the given 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.
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.

Using Procedure DBMS_XMLSCHEMA.COPYEVOLVE

Example 8–3 Loading Revised XML Schema and XSL Style Sheet

This example loads the revised XML schema and the evolution XSL style sheet into the Oracle XML DB repository.

```
DECLARE
  res BOOLEAN;
BEGIN
  res := DBMS_XDB.createResource(              -- Load revised XML schema
           '/source/schemas/poSource/revisedPurchaseOrder.xsd',
           bfilename('XMLDIR', 'revisedPurchaseOrder.xsd'),
           nls_charset_id('AL32UTF8'));
  res := DBMS_XDB.createResource(              -- Load revised XSL style sheet
           '/source/schemas/poSource/evolvePurchaseOrder.xsl',
           bfilename('XMLDIR', 'evolvePurchaseOrder.xsl'),
           nls_charset_id('AL32UTF8'));
END;
/
```

Example 8–4 shows how to use procedure `DBMS_XMLSCHEMA.copyEvolve` to evolve the purchase-order XML schema.

Example 8–4 Using DBMS_XMLSCHEMA.COPYEVOLVE to Update an XML Schema

This example evolves XML schema `purchaseOrder.xsd` to `revisedPurchaseOrder.xsd` using XSL style sheet `evolvePurchaseOrder.xsl`.

```
BEGIN
  DBMS_XMLSCHEMA.copyEvolve(
    xdb$string_list_t('http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd'),
    XMLSequenceType(XDBURIType('/source/schemas/poSource/revisedPurchaseOrder.xsd').getXML()),
    XMLSequenceType(XDBURIType('/source/schemas/poSource/evolvePurchaseOrder.xsl').getXML()));
END;

SELECT extract(object_value, '/PurchaseOrder/LineItems/LineItem[1]') LINE_ITEM
  FROM purchaseorder
  WHERE existsNode(object_value, '/PurchaseOrder[@Reference="SBELL-2003030912333601PDT"]') = 1
/

LINE_ITEM
--------------------------------------------------------------------------------
<LineItem ItemNumber="1">
  <Part Description="A Night to Remember" UnitCost="39.95">715515009058</Part>
  <Quantity>2</Quantity>
</LineItem>
```

The same query would have produced the following result before the schema evolution:

```
LINE_ITEM
------------------------------------------------------------
<LineItem ItemNumber="1">
  <Description>A Night to Remember</Description>
  <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
</LineItem>
```

Procedure `DBMS_XMLSCHEMA.copyEvolve` is used to evolve registered XML schemas in such a way that existing XML instances continue to remain valid.

> **Caution:** Before executing procedure `DBMS_XMLSCHEMA.copyEvolve`, always *back up* all registered XML schemas and all XML documents that conform to XML schemas. Procedure `copyEvolve` *deletes* all documents that conform to registered XML schemas.

First, procedure `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. Procedure `copyEvolve` creates the tables and columns as follows:

- 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.

- It adds XMLType columns using a statement of the following form:

```
ALTER TABLE <Table_Name> ADD (<Column_Name> XMLType) XMLType column
  <Column_Name> xmlschema <Schema_Url> ELEMENT <Element_Name>
```

When a new 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 true for the preserveOldDocs parameter. In this case, procedure copyEvolve does not drop the temporary tables at the end. All temporary tables are created in the database schema of the current user. For XMLType tables the temp table will have the following columns:

Table 8–3 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 8–4 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.

Procedure copyEvolve stores information about the mapping from the old table or column name to the corresponding temporary table name in a separate table specified by the mapTabName parameter. If preserveOldDocs is true, the mapTabName parameter must not be NULL, and it must not be the name of any existing table in the current database schema. Each row in the mapping table has information about one of the old tables/columns. Table 8–5 shows the mapping table columns.

Table 8–5 Procedure copyEvolve Mapping Table

Column Name	Column Type	Comment
SCHEMA_URL	VARCHAR2(700)	URL of schema to which this table/column conforms.
SCHEMA_OWNER	VARCHAR2(30)	Owner of the schema.
ELEMENT_NAME	VARCHAR2(256)	Element to which this table/column conforms.

Table 8–5 (Cont.) Procedure copyEvolve Mapping Table

Column Name	Column Type	Comment
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 not be `NULL`. This ensures that the data in the old tables is not lost. This is useful if you do not want the tables to be created by the procedure, as described in section "Procedure DBMS_XMLSCHEMA.COPYEVOLVE: Parameters and Errors".

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.

9

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 SQL function `XMLtransform` (or `XMLType` method `transform()`) in the database.
- Using XDK transformation options in the middle tier, such as XSLT Processor for Java.

> **Note:** 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.

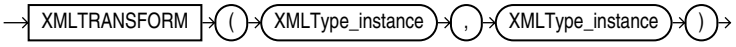
See Also:

- Chapter 3, "Using Oracle XML DB", the section, "XSL Transformation and Oracle XML DB" on page 3-70
- "PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)" on page 11-13
- Appendix C, "XSLT Primer"
- *Oracle XML Developer's Kit Programmer's Guide*, the chapter on XSQL Pages Publishing Framework

XMLTRANSFORM and XMLType.transform()

Figure 9–1 shows the syntax of SQL function `XMLtransform`. This 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.

Figure 9–1 XMLtransform Syntax



You can alternatively use `XMLType` method `transform()` as an alternative to SQL function `XMLtransform`; it has the same functionality.

Figure 9–2 shows how `XMLtransform` transforms an XML document by using an XSLT style sheet. It returns the processed output as XML, HTML, and so on, as specified by the XSLT style sheet. You typically use `XMLtransform` when retrieving or generating XML documents stored as `XMLType` in the database.

See Also: Figure 1–1, "Oracle XML DB Architecture: XMLType Storage and Repository" in Chapter 1, "Introduction to Oracle XML DB"

Figure 9–2 Using XMLTRANSFORM

XMLTRANSFORM and XMLType.transform() Examples

The examples in this section illustrate how to use SQL function `XMLtransform` and XMLType method `transform()` to transform XML data stored as `XMLType` to various formats.

Example 9–1 Registering XML Schema and Inserting XML Data

This example sets up the XML schema and tables needed to run other examples in this chapter. (The call to `deleteSchema` here ensures that there is no existing XML schema before creating one. If no such schema exists, then `deleteSchema` produces an error.)

```
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>'));
```

Example 9–2 Using XMLTRANSFORM and DBURITYPE to Retrieve a Style Sheet

`DBURIType` is described in Chapter 19, "Accessing Data Through URIs".

```
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;
```

This produces the following output (pretty-printed here for readability):

```
RESULT
------------------------------------------------------------
<td>
  <b>ipo:purchaseOrder:
    <b>shipTo:
      <b>name:Helen Zoe</b>
      <b>street:100 Broadway</b>
      <b>city:Cardiff</b>
      <b>state:Wales</b>
      <b>country:UK</b>
      <b>zip:CF2 1QJ</b>
    </b>
    <b>billTo:
      <b>name:Robert Smith</b>
      <b>street:8 Oak Avenue</b>
      <b>city:Old Town</b>
      <b>state:CA</b>
      <b>country:US</b>
      <b>zip:95819</b>
```

```
    </b>
    <b>items:</b>
  </b>
</td>
```

Example 9–3 Using XMLTRANSFORM and a Subquery to Retrieve a 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 9–4 Using XMLType.transform() with a Transient Style Sheet

This example transforms an `XMLType` instance 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 validation status of XML documents, the following functions and procedures are provided:

XMLIsValid

PL/SQL function XMLIsValid 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

PL/SQL procedure schemaValidate validates an 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

PL/SQL function isSchemaValidated 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

PL/SQL procedure setSchemaValidated 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 is 1.

isSchemaValid

PL/SQL function `isSchemaValid` 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 an 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 9–5 Using isSchemaValid()

```
SELECT x.xmlcol.isSchemaValid('http://www.example.com/schemas/ipo.xsd',
               'purchaseOrder')
    FROM po_tab x;
```

Example 9–6 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 9–7 Using schemaValidate() Within Triggers

XMLType method schemaValidate() can be used within INSERT and UPDATE TRIGGERS to ensure that all instances stored in the table are validated against the XML schema:

```
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 9–8 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 procedures described in the preceding section facilitate validation checking. Of these, isSchemaValid is the only one that throws errors that indicate why the validation has failed.

10

Full-Text Search Over XML

This chapter describes full-text search over XML using Oracle. It explains how to use SQL function `contains` and XPath function `ora:contains`, the two functions used by Oracle Database to do full-text search over XML data.

> **See Also:** *Oracle Text Reference* and *Oracle Text Application Developer's Guide* for more information on Oracle Text

This chapter contains these topics:

Overview of Full-Text Search for XML

Oracle supports full-text search on documents that are managed by the Oracle Database.

If your documents are XML, then you can use the XML structure of the document to restrict the full-text search. For example, you may want to find all purchase orders that contain the word "electric" using full-text search. If the purchase orders are in XML form, then you can restrict the search by finding all purchase orders that contain the word "electric" in a comment, or by finding all purchase orders that contain the word "electric" in a comment under line items.

If your XML documents are of type `XMLType`, then you can project the results of your query using the XML structure of the document. For example, after finding all purchase orders that contain the word "electric" in a comment, you may want to return just the comments, or just the comments that contain the word "electric".

Comparison of Full-Text Search and Other Search Types

Full-text search differs from structured search or substring search in the following ways:

- A full-text search looks for whole *words* rather than substrings. A substring search for comments that contain the *string* "law" might 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 SQL function `contains`:

  ```
  ... WHERE contains(doc, 'electric INPATH (/purchaseOrder/items/item/comment)')
            > 0 ...
  ```

  Function `contains` is an extension to SQL, and can be used in any query. It requires a `CONTEXT` full-text index.

- Include the full-text predicate inside the structure, using 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 Full-Text Search Examples

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`.

Schema and Data for Full-Text Search Examples

Examples in this chapter are based on "The Purchase Order Schema", W3C XML Schema Part 0: Primer.

See Also:

http://www.w3.org/TR/xmlschema-0/#POSchema

The data in the examples is "Purchase-Order XML Document, po001.xml" on page 10-30. Some of the performance examples are based on a larger 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 here use datatype VARCHAR2; others use XMLType. All examples that use VARCHAR2 will also work with XMLType.

Overview of CONTAINS and ora:contains

This section contains these topics:

- Overview of SQL Function CONTAINS
- Overview of XPath Function ora:contains
- Comparison of CONTAINS and ora:contains

Overview of SQL Function CONTAINS

SQL function contains returns a positive number for rows where [schema.]column matches text_query, and zero otherwise. It is a user-defined function, a standard extension method in SQL. It requires an index of type CONTEXT. If there is no CONTEXT index on the column being searched, then contains throws an error.

Syntax

```
contains([schema.]column, text_query VARCHAR2 [,label NUMBER])
RETURN NUMBER
```

Example 10–1 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 10–2 CONTAINS with a Structured Predicate

SQL function 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 10–3 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 10–4 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 XPath Function ora:contains

XPath function `ora:contains` can be used in an XPath expression inside an XQuery expression or in a call to SQL function `existsNode`, `extract`, or `extractValue`. It is used to restrict a structural search with a full-text predicate. Function `ora:contains` returns a positive integer when the *`input_text`* matches *`text_query`* (the higher the number, the more relevant the match), and zero otherwise. When used in an XQuery expression, the XQuery return type is `xs:integer()`; when used in an XPath expression outside of an XQuery expression, the XPath return type is `number`.

Argument *`input_text`* must evaluate to a single text node or an attribute. The syntax and semantics of *`text_query`* in `ora:contains` are the same as *`text_query`* in `contains`, with the following restrictions:

- Argument *`text_query`* cannot include any structure operators (`WITHIN`, `INPATH`, or `HASPATH`).
- If the `weight` score-weighting operator is used, the weights will be ignored.

Function `ora:contains` extends XPath through a standard mechanism: it is a user-defined function in the Oracle XML DB namespace, `ora`.

Syntax

```
ora:contains(input_text NODE*, text_query STRING
             [,policy_name STRING]
             [,policy_owner STRING])
```

Example 10–5 shows a call to `ora:contains` in the XPath parameter to `existsNode`. Note that the third parameter to `existsNode` (the Oracle XML DB namespace, `ora`) is required. This example uses table `purchase_orders_xmltype`.

Example 10–5 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 10-18 for more on the `ora:contains` XPath function

Comparison of CONTAINS and ora:contains

SQL function `contains`:

- 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 SQL function `score`)
- 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)

XPath function `ora:contains`:

- 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 Using Function CONTAINS

The second argument to SQL function `contains`, `text_query`, is a string that specifies the full-text search. `text_query` has its own language, based on the SQL/MM Full-Text standard.

See Also:

- ISO/IEC 13249-2:2000, Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text, International Organization For Standardization, 2000
- *Oracle Text Reference* for more information 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 in any of the following ways:

- `AND`, `OR`, `NOT`
- `and`, `or`, `not`
- `&`, `|`, `~`

Note that `NOT` is a *binary*, not a unary operator here. The expression `alpha NOT(beta)` is equivalent to `alpha AND` *unary-not*`(beta)`, where unary-not stands for unary negation.

> **See Also:** *Oracle Text Reference* for complete information on the operators you can use in `contains` and `ora:contains`

Example 10–6 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 10–7 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`.

Stemming: $

The `text_query` language supports stemmed search. Example 10–8 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 10–8 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 10–9.

Example 10–9 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 SQL Function

SQL function `contains` has a related function, `score`, which 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 10–10, `score(10)` returns the score for each row in the result set. SQL function `score` returns the relevance of a row in the result set with respect to a particular call to function `contains`. A call to `score` is linked to a call to `contains` by a `LABEL` (in this case the number 10).

Example 10–10 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`.

Function `score` always returns `0` if, for the corresponding `contains`, the `text_query` argument 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 a CONTAINS Search

SQL function `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 them.

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 Structure Operator

The `WITHIN` operator restricts a query to some section within an XML document. A search for purchase orders that contain the word "lawn" somewhere inside a comment section might use `WITHIN`. Section names are case-sensitive.

Example 10–11 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 10–12 finds all documents that contain the word "`lawn`" within a section "`comment`", where that occurrence of "`lawn`" is also within a section "`item`".

Example 10–12 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 10–12 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 10–13 finds all purchase orders that contain the word `10` in the `orderDate` attribute of a `purchaseOrder` element.

Example 10–13 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`.

By default, the minus sign ("`-`") is treated as a word separator: "`1999-10-20`" is treated as the three words "`1999`", "`10`" and "`20`". So this query returns one row.

Text in an attribute is not a part of the main searchable document. 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 10–14.

Example 10–14 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 10–15.

Example 10–15 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 10–15 will return no rows. Example 10–16, which omits the parentheses around `lawn AND electric`, on the other hand, will return one row.

Example 10–16 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`.

Operator `WITHIN` has a higher precedence than `AND`, so Example 10–16 is parsed as Example 10–17.

Example 10–17 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 preceding examples have used the `WITHIN` operator to search within a section. A **section** can be a:

- **path** or **zone** section

  This is a concatenation, in document order, of all text nodes that are descendants of a node, with whitespace separating the text nodes. To convert from a node to a zone section, you must serialize the node and replace all tags with whitespace. path sections have the same scope and behavior as zone sections, except that path sections support queries with `INPATH` and `HASPATH` structure operators.

- **field** section

  This is the same as a zone section, except that repeating nodes in a document are concatenated into a single section, with whitespace as a separator.

- attribute section
- special section (sentence or paragraph)

> **See Also:** *Oracle Text Reference* for more information on special sections

INPATH Structure Operator

Operator `WITHIN` provides an easy and intuitive way to express simple structure restrictions in the `text_query`. For queries that use abundant XML structure, you can use operator `INPATH` plus a text path instead of nested `WITHIN` operators.

Operator `INPATH` takes a `text_query` on the left and a Text Path, enclosed in parentheses, on the right. Example 10–18 finds `purchaseOrders` that contain the word "electric" in the path `/purchaseOrder/items/item/comment`.

Example 10–18 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 10–19.

Example 10–19 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`.

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 Specification" on page 10-29 for the Text Path grammar

Example 10–20 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 10–20 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 10–21 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 one row. Note that the scope of the query is *not* a "`comment`", but the set of "`items`" that each have a "`comment`" as a descendant.

Example 10–21 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 purchase-order data, `purchaseOrder/items/item`, `/purchaseOrder/items/item`, and `./purchaseOrder/items/item` are all equivalent.

- If you want to search within an attribute value, then the direct parent of the attribute must be specified (wildcards cannot be used).

- A Text Path may not end in a wildcard (`*`).

  > **See Also:** "Text Path BNF Specification" on page 10-29 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 10–22 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 10–22 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 10–23.

Example 10–23 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 Structure Operator

Operator `HASPATH` takes only one operand: a Text Path, enclosed in parentheses, on the right. Use `HASPATH` when you want to find documents that contain a particular section in a particular path, possibly with 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 SQL function `existsNode` instead of structure operator `HASPATH`.

Example 10–24 finds purchaseOrders that have some item that has a USPrice.

Example 10–24 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 10–25 finds purchaseOrders that have some item that has a USPrice that text-equals "148.95".

> **See Also:** "Text Path Compared to XPath" on page 10-10 for an explanation of text-equals

Example 10–25 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 10–26 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 10–26 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 CONTAINS Result

The result of a SQL query with a contains expression in the WHERE clause is always a set of rows (and possibly score information), or a projection over the rows that match the query. If you want to return only a part of each XML document that satisfies the contains expression, then use SQL functions extract and extractValue. Note that extract and extractValue operate on XMLType, so the following examples use the table purchase_orders_xmltype.

Example 10–27 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 10–27 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 10–27 is *two* instances of "comment". Function contains indicates 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 expression, then you must repeat that predicate in the extract. You do that with ora:contains, which is an XPath function.

Example 10–28 returns only the "comment" that matches the contains expression.

Example 10–28 Using EXTRACT and ora:contains to Project the Result 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 with the CONTEXT Index

This section contains these topics:

- Introduction to the CONTEXT Index
- Effect of the CONTEXT Index on CONTAINS
- 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 10–29.

Example 10–29 Simple CONTEXT Index on Table PURCHASE_ORDERS

```
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 10–30.

See Also: "CONTEXT Index: Preferences" on page 10-15

Example 10–30 Simple CONTEXT Index on Table PURCHASE_ORDERS 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 index types, 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 10–29 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 10–31 creates a CONTEXT index on a column of type XMLType.

Example 10–31 Simple CONTEXT Index on Table PURCHASE_ORDERS_xmltype

```
CREATE INDEX po_index_xmltype ON purchase_orders_xmltype(doc)
  INDEXTYPE IS CTXSYS.CONTEXT;
```

This example uses table purchase_orders_xmltype. The section group defaults to PATH_SECTION_GROUP.

If you have a table of type XMLType, then you need to use object syntax to create the CONTEXT index as shown in Example 10–32.

Example 10–32 Simple CONTEXT Index on XMLType Table

```
CREATE INDEX po_index_xmltype_table
  ON purchase_orders_xmltype_table (OBJECT_VALUE)
  INDEXTYPE IS CTXSYS.CONTEXT;
```

This example uses table purchase_orders_xmltype.

You can query the table using the syntax in Example 10–33.

Example 10–33 CONTAINS Query on XMLType Table

```
SELECT extract(OBJECT_VALUE, '/purchaseOrder/@orderDate') "Order Date"
  FROM purchase_orders_xmltype_table
  WHERE contains(OBJECT_VALUE, '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 can 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 SQL function contains. If you call contains, and the column given in the first argument does not have an index of type CONTEXT, then an error is raised.

The syntax and semantics of text_query depend on the choices you make when you build the CONTEXT index. For example:

- What counts as a word?
- Are very common words processed?
- What is a common word?
- Is the text search case-sensitive?
- Can the text search include themes (concepts) as well as keywords?

CONTEXT Index: Preferences

A preference can be considered a collection of indexing choices. Preferences include section group, datastore, filter, wordlist, stoplist and storage. This section shows how to set up a lexer preference to make searches case-sensitive.

You can use procedure CTX_DDL.create_preference (or CTX_DDL.create_stoplist) to create a preference. Override default choices in that preference group by setting attributes of the new preference, using procedure CTX_DDL.set_attribute. Then use the preference in a CONTEXT index by including *preference type preference_name* in the PARAMETERS string of CREATE INDEX.

Once a preference has been created, you can use it to build any number of indexes.

Making Search Case-Sensitive Full-text searches with contains are **case-insensitive** by default. That is, when matching words in text_query against words in the document, case is not considered. Section names and attribute names, however, are always **case-sensitive**.

If you want full-text searches to be case-sensitive, then you need to make that choice when building the CONTEXT index. Example 10–34 returns 1 row, because "HURRY" in text_query matches "Hurry" in the purchaseOrder with the default case-insensitive index.

Example 10–34 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 10–35 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 10–35 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 10–36 builds a CONTEXT index using the new my_lexer lexer preference.

Example 10–36 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 10–34 returns no rows, because "HURRY" in text_query no longer matches "Hurry" in the purchaseOrder. Example 10–37 returns one row, because the text_query term "Hurry" exactly matches the word "Hurry" in the purchaseOrder.

Example 10–37 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 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 leaf nodes that do not repeat).

> **See Also:** *Oracle Text Reference* for a detailed description of the XML_SECTION_GROUP section group type

Choosing a Section Group When choosing a section group to use with your index, you can choose a supplied section group, take the default, or create a new section group based on the section group type you have chosen.

There are supplied section groups for section group types PATH_SECTION_GROUP, AUTO_SECTION_GROUP, and NULL_SECTION_GROUP. The supplied section groups are owned by CTXSYS and have the same name as their section group types. For example, the supplied section group of section group type PATH_SECTION_GROUP is CTXSYS.PATH_SECTION_GROUP.

There is no supplied section group for section group type XML_SECTION_GROUP, because a default XML_SECTION_GROUP would be empty and therefore meaningless. If you want to use section group type XML_SECTION_GROUP, then you must create a new section group and specify each node that you want to include as a section.

When you create a CONTEXT index on data of type XMLType, the default section group is the supplied section group CTXSYS.PATH_SECTION_GROUP. If the data is VARCHAR or CLOB, then the default section group is CTXSYS.NULL_SECTION_GROUP.

> **See Also:** *Oracle Text Reference* for instructions on creating your own section group

To associate a section group with an index, add `section group <section group name>` to the PARAMETERS string, as in Example 10–38.

Example 10–38 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".

ora:contains returns a number; it does *not* return a score. It returns a positive number if the text_query matches the input_text. Otherwise it returns zero.

Full-Text Search Using Function ora:contains

The ora:contains argument *text_query* is a string that specifies the full-text search. The ora:contains *text_query* is the same as the contains *text_query*, with the following restrictions:

- ora:contains *text_query* must *not* include any of the structure operators WITHIN, INPATH, or HASPATH
- ora:contains *text_query* 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 operator MINUS (-) or threshold (>)
- Selective, corpus-based expansion operator FUZZY (?) or soundex (!)

> **See Also:** "XPath Rewrite and the CONTEXT Index" on page 10-27

Example 10–39 shows a full-text search using an arbitrary combination of Boolean operators and $ (stemming).

Example 10–39 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 Using Function CONTAINS" on page 10-5 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`.

Structure: Restricting the Scope of an ora:contains Query

When you use `ora:contains` in an XPath expression, the scope is defined by argument *`input_text`*. This argument is evaluated in the current XPath context. If the result is a single text node or an attribute, then that node is the target of the `ora:contains` search. If *`input_text`* does not evaluate to a single text node or an attribute, an error is raised.

The policy determines the matching rules for `ora:contains`. The section group associated with the default policy for `ora:contains` is of type `NULL_SECTION_GROUP`.

`ora:contains` can be used anywhere in an XPath expression, and its *`input_text`* argument can be any XPath expression that evaluates to a single text node or an attribute.

Structure: Projecting the ora:contains 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 10–40 ora:contains in EXISTSNODE and EXTRACT

This example returns the `orderDate` for each `purchaseOrder` that has a `comment` that contains the word "`lawn`". It uses table `purchase_orders_xmltype`.

```
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;
```

Function `existsNode` restricts the result to rows (documents) where the `purchaseOrder` element includes some `comment` that contains the word "`lawn`". Function `extract` then returns the value of attribute `orderDate` from those `purchaseOrder` elements. If `//comment` had been extracted, then both comments from the sample document would have been returned, not just the comment that matches the `WHERE` clause.

See Also: Example 10–27, "Using EXTRACT to Scope the Results of a CONTAINS Query" on page 10-12

Policies for ora:contains Queries

The CONTEXT index on a column determines the semantics of contains queries on that column. Because ora:contains does not rely on a supporting index, some other means must be found to provide many of the same choices when doing ora:contains queries. A **policy** is a collection of preferences that can be associated with an ora:contains query to give the same sort of semantic control as the indexing choices give to the contains user.

Introduction to Policies for ora:contains Queries

When using SQL function contains, indexing preferences affect the semantics of the query. You create a preference using procedure CTX_DDL.create_preference (or CTX_DDL.create_stoplist). You override default choices by setting attributes of the new preference, using procedure CTX_DDL.set_attribute. Then you use the preference in a CONTEXT index by including *preference_type preference_name* in the PARAMETERS string of CREATE INDEX.

See Also: "CONTEXT Index: Preferences" on page 10-15

Because ora:contains does not have a supporting index, a different mechanism is needed to apply preferences to a query. That mechanism is a policy, consisting of a collection of preferences, and it is used as a parameter to ora:contains.

Policy Example: Supplied Stoplist Example 10–41 creates a policy with an empty stopwords list.

Example 10–41 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 user CTXSYS. You could create a new stoplist to include in this policy, or you could reuse a stoplist (or lexer) definition that you created for a CONTEXT index.

Refer to this policy in an ora:contains expression to search for all words, including the most common ones (stopwords). Example 10–42 returns zero comments, because "is" is a stopword by default and cannot be queried.

Example 10–42 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 10–43 uses the policy created in Example 10–41 to specify an empty stopword list. This query finds "is" and returns 1 comment.

Example 10–43 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`. (This policy was implicitly named as all uppercase in Example 10–41. Because XPath is case-sensitive, it must be referred to in the XPath predicate using all uppercase: `MY_NOSTOPWORDS_POLICY`, not `my_nostopwords_policy`.)

Note: As always:

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

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

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 10-20 for an example of building a policy with a predefined stoplist
- "Policy Example: User-Defined Lexer" on page 10-21 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 procedure `CTX_DDL.set_attribute`.
3. Use the preference as a parameter to `CTX_DDL.create_policy`.
4. Use the policy name as the third argument to `ora:contains` in a query.

Once you have created a preference, you can reuse it in other policies or in `CONTEXT` index definitions. You can use any policy with any `ora:contains` query.

Example 10–44 returns 1 row, because "HURRY" in `text_query` matches "Hurry" in the `purchaseOrder` with the default case-insensitive index.

Example 10–44 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 10–45 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 10–45 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 10–46 creates a new policy `my_policy` and specifies only the lexer. All other preferences are defaulted.

Example 10–46 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 10–47 uses the new policy in a query. It returns no rows, because "HURRY" in `text_query` no longer matches "Hurry" in the `purchaseOrder`.

Example 10–47 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 10–48 returns one row, because the `text_query` term "`Hurry`" exactly matches the word "`Hurry`" in the `purchaseOrder`.

Example 10–48 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 10–46 on page 10-22, 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 Searches Over a Collection of Elements

It is common to use the schema annotation `storeVarrayAsTable="true"`, and set parameter `genTables="true"` during schema registration. This automatically creates the tables needed to store the corresponding XML data.

However, if you use these settings, then you will *not* be able to use Oracle Text indexes for text-based `ora:contains` searches over a collection of elements – that is, a document with repetitions of the same element type. The `storeVarrayAsTable = "true"` schema annotation causes element collections to be persisted as rows in an index-organized table (IOT), and *Oracle Text does not support IOTs*.

To be able to use Oracle Text to search the contents of element collections, you must set parameter `genTables="false"` during schema registration, then create the necessary tables *manually*.

Example 10–49 shows how to manually create a table that corresponds to an element definition in an XML schema. It is identical to Example 3–11 on page 3-27 (compare), except that it does *not* use the clause `ORGANIZATION INDEX OVERFLOW`. The lack of

this clause causes the table to be *heap-organized* instead of index-organized, enabling searching with Oracle Text indexes.

Example 10–49 Creating a Heap-Organized 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, SYS_NC_ARRAY_INDEX$)))
  VARRAY "XMLDATA"."lineitems"."lineitem"
    STORE AS TABLE lineitem_table
                  ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)));
```

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 command `AUTOTRACE`. Execution plans can also be produced using SQL commands `TRACE` and `TKPROF`. A description of commands `AUTOTRACE`, `trace` and `tkprof` is outside the scope of this chapter.

This section contains these topics:

- Use a Primary Filter in the Query
- Use a CTXXPATH Index
- XPath 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 10–50 examines every row in the table (does a full table scan), as we can see from the Plan in Example 10–51. In this example, `ora:contains` is evaluated for every row.

Example 10–50 ora:contains in EXISTSNODE, Large 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 10–51 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 10–52, and add a selective `id` predicate to the query, as shown in Example 10–53, then it is apparent from Example 10–54 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 10–52 B-Tree Index on ID

```
CREATE INDEX id_index ON purchase_orders_xmltype_big(id);
```

This example uses table `purchase_orders`.

Example 10–53 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 10–54 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 10–55 produces the execution plan shown in Example 10–56.

Example 10–55 ora:contains in EXISTSNODE, Large 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 10–56 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 10–57. You can create a CTXXPATH index and a CONTEXT index on the same column.

Example 10–57 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 10–55 again and you will see from the plan, shown in Example 10–58, that the query now uses the CTXXPATH index.

Example 10–58 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 10–55, 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 10–59), then `CTXXPATH` will process those predicates and therefore will be a useful primary filter. `CTXXPATH` handles both string and number equality predicates.

Example 10–59 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 10–60. 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 10–60 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

XPath 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.

Benefits of XPath 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 XPath rewrite are:

- The rewritten query can directly access the underlying object-relational tables instead of processing the whole XML document.
- The rewritten query can make use of any available indexes.

XPath rewrite is a performance optimization. XPath rewrite only makes sense if the XML data is stored object-relationally, which in turn requires the XML data to be XML schema-based.

See Also:

- "Rewriting of XPath Expressions: XPath Rewrite" on page 1-18 for more on the benefits of XPath rewrite
- Chapter 6, "XPath Rewrite" for a full discussion of XPath rewrite

From Documents to Nodes Consider Example 10–61, 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 this XPath expression:

```
/purchaseOrder/items/item/comment[ora:contains(text(), "electric")>0]
```

Example 10–61 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 column `/purchaseOrder/items/item/comment` (if such a column exists) and test each cell there to see if it matches "`electric`".

This is the first XPath-rewrite step. If the first argument to `ora:contains` (`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 10-28, 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 rewritten 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 for ora:contains Queries" on page 10-20, 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 Also: *Oracle Text Reference*

XPath Rewrite: Summary A query with existsNode, extract or extractValue, where the XPath includes ora:contains, may be considered for XPath 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.

XPath rewrite can speed up queries significantly, especially if there is a suitable CONTEXT index.

Text Path BNF Specification

```
HasPathArg           ::=   LocationPath
                         |  EqualityExpr
InPathArg            ::=   LocationPath
LocationPath         ::=   RelativeLocationPath
                         |  AbsoluteLocationPath
AbsoluteLocationPath ::=   ("/" RelativeLocationPath)
                         |  ("//" RelativeLocationPath)
RelativeLocationPath ::=   Step
                         |  (RelativeLocationPath "/" Step)
                         |  (RelativeLocationPath "//" Step)
Step                 ::=   ("@" NCName)
                         |  NCName
```

```
                          |  (NCName Predicate)
                          |  Dot
                          |  "*"
Predicate           ::=   ("[" OrExp "]")
                          |  ("[" Digit+ "]")
OrExpr              ::=   AndExpr
                          |  (OrExpr "or" AndExpr)
AndExpr             ::=   BooleanExpr
                          |  (AndExpr "and" BooleanExpr)
BooleanExpr         ::=   RelativeLocationPath
                          |  EqualityExpr
                          |  ("(" OrExpr ")")
                          |  ("not" "(" OrExpr ")")
EqualityExpr        ::=   (RelativeLocationPath "=" Literal)
                          |  (Literal "=" RelativeLocationPath)
                          |  (RelativeLocationPath "=" Literal)
                          |  (Literal "!=" RelativeLocationPath)
                          |  (RelativeLocationPath "=" Literal)
                          |  (Literal "!=" RelativeLocationPath)
Literal             ::=   (DoubleQuote [~"]* DoubleQuote)
                          |  (SingleQuote [~']* SingleQuote)
NCName              ::=   (Letter |  Underscore) NCNameChar*
NCNameChar          ::=   Letter
                          |  Digit
                          |  Dot
                          |  Dash
                          |  Underscore
Letter              ::=   ([a-z] | [A-Z])
Digit               ::=   [0-9]
Dot                 ::=   "."
Dash                ::=   "-"
Underscore          ::=   "_"
```

Support for Full-Text XML Examples

This section contains these topics:

- Purchase-Order XML Document, po001.xml
- CREATE TABLE Statements
- Purchase-Order XML Schema for Full-Text Search Examples

Purchase-Order XML Document, po001.xml

Example 10–62 Purchase Order XML Document, po001.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation="xmlschema/po.xsd"
               orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
```

```
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

CREATE TABLE Statements

Example 10–63 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 10–64 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 10–65 CREATE TABLE purchase_orders_xmltype_table

```
CREATE TABLE purchase_orders_xmltype_table OF XMLType;

INSERT INTO purchase_orders_xmltype_table
  VALUES (
    XMLType ('<?xml version="1.0" encoding="UTF-8"?>
               <purchaseOrder
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:noNamespaceSchemaLocation="xmlschema/po.xsd"
                 orderDate="1999-10-20">
```

```
                  <shipTo country="US">
                    <name>Alice Smith</name>
                    <street>123 Maple Street</street>
                    <city>Mill Valley</city>
                    <state>CA</state>
                    <zip>90952</zip>
                  </shipTo>
                  <billTo country="US">
                    <name>Robert Smith</name>
                    <street>8 Oak Avenue</street>
                    <city>Old Town</city>
                    <state>PA</state>
                    <zip>95819</zip>
                  </billTo>
                  <comment>Hurry, my lawn is going wild!</comment>
                  <items>
                    <item partNum="872-AA">
                      <productName>Lawnmower</productName>
                      <quantity>1</quantity>
                      <USPrice>148.95</USPrice>
                      <comment>Confirm this is electric</comment>
                    </item>
                    <item partNum="926-AA">
                      <productName>Baby Monitor</productName>
                      <quantity>1</quantity>
                      <USPrice>39.98</USPrice>
                      <shipDate>1999-05-21</shipDate>
                    </item>
                  </items>
                </purchaseOrder>'));
COMMIT;
```

Purchase-Order XML Schema for Full-Text Search Examples

Example 10–66 Purchase-Order XML Schema for Full-Text Search Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
  <xsd:element name="comment" type="xsd:string"/>
  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>
  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
```

```
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
  </xsd:complexType>
  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="USPrice" type="xsd:decimal"/>
            <xsd:element ref="comment" minOccurs="0"/>
            <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="partNum" type="SKU" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Part III

Using 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:

11

PL/SQL API for XMLType

This chapter describes the use of the APIs for XMLType in PL/SQL.

This chapter contains these topics:

Overview of PL/SQL APIs for XMLType

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.

API Features

The PL/SQL APIs for XMLType allow you to perform the following tasks:

- Create XMLType tables, columns, and views
- Construct XMLType instances from data encoded in different character sets.
- Access XMLType data

- Manipulate `XMLType` data

See Also:

- "Oracle XML DB Features", for an overview of the Oracle XML DB architecture and new features.
- Chapter 4, "XMLType Operations"
- *Oracle Database PL/SQL Packages and Types Reference*

Lazy Loading of XML Data (Lazy Manifestation)

Because `XMLType` provides 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 Supports XML Schema

The `XMLType` datatype includes support for XML schemas. You can create an XML schema and annotate it with mappings from XML to object-relational storage. To take advantage of the PL/SQL DOM API, first create an XML schema and register it. Then, when you create `XMLType` tables and columns, you can specify that these conform to the registered XML schema.

XMLType Supports Data in Different Character Sets

`XMLType` instances can be created from data encoded in any Oracle-supported character set by using the PL/SQL `XMLType` constructor or `XMLType` method `createXML()`. The source XML data must be supplied using datatype `BFILE` or `BLOB`. The encoding of the data is specified through argument `csid`. When this argument is zero (`0`), the encoding of the source data is determined from the XML prolog, as specified in Appendix F of the XML 1.0 Reference.

Method `getBlobVal()` retrieves the XML contents in the requested character set.

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

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

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

PL/SQL DOM API for XMLType (DBMS_XMLDOM)

This section describes the PL/SQL DOM API for `XMLType`, `DBMS_XMLDOM`.

Overview of the W3C Document Object Model (DOM) Recommendation

Skip this section if you are familiar with the generic DOM specifications recommended by the World Wide Web Consortium (W3C).

The Document Object Model (DOM) recommended by the W3C is a universal API for accessing the structure of XML documents. It was originally developed to formalize Dynamic HTML, which allows animation, interaction, and dynamic updating of Web pages. DOM provides a language-neutral and platform-neutral object model for Web pages and XML document structures. DOM describes language-independent and platform-independent interfaces to access and operate on XML components and elements. It expresses the structure of an XML document in a universal, content-neutral way. Applications can be written to dynamically delete, add, and edit the content, attributes, and style of XML documents. DOM makes it possible to create applications that work properly on all browsers, servers, and platforms.

Oracle XDK Extensions to the W3C DOM Standard

Oracle XML Developer's Kit (Oracle XDK) extends the W3C DOM API in various ways. All of these extensions are supported by Oracle XML DB except those relating to client-side operations that are not applicable in the database. This type of procedural processing is available through the SAX interface in the Oracle XDK Java and C components.

See Also: *Oracle XML Developer's Kit Programmer's Guide*

Supported W3C DOM Recommendations

All Oracle XML DB APIs for accessing and manipulating XML comply with standard XML processing requirements as approved by the W3C. The PL/SQL DOM supports Levels 1 and 2 from the W3C DOM specifications.

- In Oracle9*i* release 1 (9.0.1), Oracle XDK for PL/SQL implemented DOM Level 1.0 and parts of DOM Level 2.0.
- In Oracle9*i* release 2 (9.2) and Oracle Database 10*g* release 1 (10.1), the PL/SQL API for `XMLType` implements DOM Levels 1.0 and Level 2.0 Core, and is fully integrated in the database through extensions to the `XMLType` API.

The following briefly describes each level:

- **DOM Level 1.0 –** The first formal Level of the DOM specifications, completed in October 1998. Level 1.0 defines support for XML 1.0 and HTML.
- **DOM Level 2.0 –** Completed in November 2000, Level 2.0 extends Level 1.0 with support for XML 1.0 with namespaces and adds support for Cascading Style Sheets (CSS) and events (user-interface events and tree manipulation events), and enhances tree manipulations (tree ranges and traversal mechanisms). CSS are a simple mechanism for adding style (fonts, colors, spacing, and so on) to Web documents.

Difference Between DOM and SAX

The generic APIs for XML can be classified in two main categories:

- *Tree-based.* DOM is the primary generic tree-based API for XML.

- Event-based. SAX (Simple API for XML) is the primary generic event-based programming interface between an XML parser and an XML application.

DOM works by creating objects. These objects have child objects and properties, and the child objects have child objects and properties, and so on. Objects are referenced either by moving down the object hierarchy or by explicitly giving an HTML element an ID attribute. For example:

```
<img src="employee_jdoe.gif" ID="0123jdoe">
```

Examples of structural manipulations are:

- Reordering elements
- Adding or deleting elements
- Adding or deleting attributes
- Renaming elements

PL/SQL DOM API for XMLType (DBMS_XMLDOM): Features

Oracle XML DB extends the Oracle Database XML development platform beyond SQL support for storage and retrieval of XML data. It lets you operate on `XMLType` instances using DOM in PL/SQL, Java, and C.

The *default* action for the PL/SQL DOM API for `XMLType` (`DBMS_XMLDOM`) is to do the following:

- Produce a parse tree that can be accessed by DOM APIs.
- Validate, if a DTD is found; otherwise, do not validate.
- Raise an application error if parsing fails.

DTD validation occurs when the object document is manifested. If lazy manifestation is employed, then the document is validated when it is used.

The PL/SQL DOM API exploits a C-based representation of XML in the server and operates on XML schema-based XML instances. The PL/SQL, Java, and C DOM APIs for `XMLType` comply with the W3C DOM Recommendations to define and implement structured storage of XML 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 11-6, for a description of W3C DOM Recommendations.

XML Schema Support

The PL/SQL DOM API for `XMLType` supports XML schema. Oracle XML DB uses annotations within an XML schema as metadata to determine the structure of an XML document structure and the mapping of the document to a database schema.

> **Note:** For backward compatibility and flexibility, the PL/SQL DOM supports both XML schema-based documents and non-schema-based documents.

After an XML schema is registered with Oracle XML DB, the PL/SQL DOM API for `XMLType` builds an in-memory tree representation of an associated 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. Nodes of some node types can have child nodes of various

types, while nodes of other node types must be leaf nodes, which do not have child nodes.

Enhanced Performance

Oracle XML DB uses DOM to provide a standard way to translate data between XML and multiple back-end data sources. This eliminates the need to use separate XML translation techniques for the different data sources in your environment. Applications needing to exchange XML data can use a single native XML database to cache XML documents. Oracle XML DB can thus speed up application performance by acting as an intermediate cache between your Web applications and your back-end data sources, whether they are in relational databases or file systems.

See Also: Chapter 13, "Java API for XMLType"

Designing End-to-End Applications Using Oracle XDK and Oracle XML DB

When you build applications based on Oracle XML DB, you do not need the additional components in Oracle XDK. However, you can use Oracle XDK components with Oracle XML DB to deploy a full suite of XML-enabled applications that run end-to-end. You can use features in Oracle XDK for:

- Simple API for XML (SAX) interface processing. SAX is an XML standard interface provided by XML parsers and used by procedural and event-based applications.
- DOM interface processing, for structural and recursive object-based processing.

Oracle XDK contain the basic building blocks for creating applications that run on a client, in a browser or a plug-in. Such applications typically read, manipulate, transform and view XML documents. To provide a broad variety of deployment options, Oracle XDK is available for Java, C, and C++. Oracle XDK is fully supported and comes with a commercial redistribution license.

Oracle XDK for Java consists of these components:

- **XML Parsers –** Creates and parses XML using industry standard DOM and SAX interfaces. Supports Java, C, C++, and JAXP.
- **XSL Processor –** Transforms or renders XML into other text-based formats such as HTML. Supports Java, C, and C++.
- **XML Schema Processor –** Allows use of XML simple and complex datatypes. Supports Java, C, and C++.
- **XML Class Generator, Oracle JAXB Class Generator –** Automatically generate C++ and Java classes, respectively, from DTDs and XML schemas, to send XML data from Web forms or applications. Class generators accept an input file and create a set of output classes that have corresponding functionality. 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 SQL Utility –** Generates XML documents, DTDs, and XML schemas from SQL queries. Supports Java.
- **TransX Utility –** Loads data encapsulated in XML into the database. Has additional functionality useful for installations.
- **XSQL Servlet –** Combines XML, SQL, and XSLT in the server to deliver dynamic web content.
- **XML Pipeline Processor –** Invokes Java processes through XML control files.

- **XSLT VM and Compiler –** Provides a high-performance C-based XSLT transformation engine that uses compiled style sheets.
- **XML Java Beans –** Parses, transforms, compares, retrieves, and compresses XML documents using Java components.

See Also: *Oracle XML Developer's Kit Programmer's Guide*

Using PL/SQL DOM API for XMLType: Preparing XML Data

To prepare data for using PL/SQL DOM APIs in Oracle XML DB:

1. Create a standard XML schema.
2. Annotate the XML schema with definitions for the SQL objects you use.
3. Register the XML schema, to generate the necessary database mappings.

You can then do any of the following:

- Use `XMLType` views to wrap existing relational or object-relational data in XML formats, making it available to your applications in XML form. See "Wrapping Existing Data into XML with XMLType Views" on page 11-7.
- Insert XML data into `XMLType` columns.
- Use Oracle XML DB DOM PL/SQL and Java APIs to manipulate XML data stored in `XMLType` columns and tables.

Defining an XML Schema Mapping to SQL Object Types

An XML schema must be registered before it can be referenced by an XML document. When you register an XML schema, elements and attributes it declares are mapped to attributes of corresponding SQL object types within the database.

After XML schema registration, XML documents that conform to the XML schema and reference it can be managed by Oracle XML DB. Tables and columns for storing the conforming documents can be created for root elements defined by the XML schema.

See Also: Chapter 5, "XML Schema Storage and Query: Basic"

An XML schema is registered by using PL/SQL package `DBMS_XMLSCHEMA` and by specifying the schema document and its *schema-location URL*. This URL is a name that uniquely identifies the registered schema within the database; it need not correspond to any real location—in particular, it need not indicate where the schema document is located.

The *target namespace* of the schema is another URL used in the XML schema. It specifies a namespace for the XML-schema elements and types. An XML document should specify both the namespace of the root element and the schema-location URL identifying the schema that defines this element.

When documents are inserted into Oracle XML DB using path-based protocols such as HTTP(S) and FTP, the XML schema to which the document conforms is *registered implicitly*, provided its name and location are specified and it has not yet been registered.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

DOM Fidelity for XML Schema Mapping

Elements and attributes declared within the XML schema get mapped to separate attributes of the corresponding SQL object type. Other information encoded in an XML document, such as comments, processing instructions, namespace declarations and prefixe definitions, and whitespace, is not represented directly.

To store this additional information, binary attribute `SYS_XDBPD$` is present in all generated SQL object types. This database attribute stores all information in the original XML document that is not stored using the other database attributes. Retaining this accessory information ensures *DOM fidelity* for XML documents stored in Oracle XML DB: an XML document retrieved from the database is identical to the original document that was stored.

Note: In this book, the `SYS_XDBPD$` attribute has been omitted from most examples, for simplicity. However, the attribute is always present in SQL object types generated by schema registration.

Wrapping Existing Data into XML with XMLType Views

To make existing relational and object-relational data available to your XML applications, you can create `XMLType` views, wrapping the data in an XML format. You can then accese this XML data using the PL/SQL DOM API.

After you register an XML schema containing annotations that represent the mapping between XML types and SQL object types, you can create an `XMLType` view that conforms to the XML schema.

> **See Also:** Chapter 18, "XMLType Views"

DBMS_XMLDOM Methods Supported

All `DBMS_XMLDOM` methods are supported by Oracle XML DB, with the *exception* of the following:

- `writeExternalDTDToFile()`
- `writeExternalDTDToBuffer()`
- `writeExternalDTDToClob()`

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for descriptions of the individual `DBMS_XMLDOM` methods

PL/SQL DOM API for XMLType: Node Types

In the DOM specification, the term "document" is used to describe a container for many different kinds of information or data, which the DOM objectifies. The DOM specifies the way elements within an XML document container are used to create an object-based tree structure and to define and expose interfaces to manage and use the objects stored in XML documents. Additionally, the DOM supports storage of documents in diverse systems.

When a request such as `getNodeType(myNode)` is given, it returns `myNodeType`, which is the node type supported by the parent node. These constants represent the different types that a node can adopt:

- `ELEMENT_NODE`

- ATTRIBUTE_NODE
- TEXT_NODE
- CDATA_SECTION_NODE
- ENTITY_REFERENCE_NODE
- ENTITY_NODE
- PROCESSING_INSTRUCTION_NODE
- COMMENT_NODE
- DOCUMENT_NODE
- DOCUMENT_TYPE_NODE
- DOCUMENT_FRAGMENT_NODE
- NOTATION_NODE

Table 11–1 shows the node types for XML and HTML and the allowed corresponding children node types.

Table 11–1 XML and HTML DOM Node Types and Their Child Node Types

Node Type	Children Node Types
Document	Element (maximum of one), ProcessingInstruction, Comment, DocumentType (maximum of one)
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	No children
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Text, EntityReference
ProcessingInstruction	No children
Comment	No children
Text	No children
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.

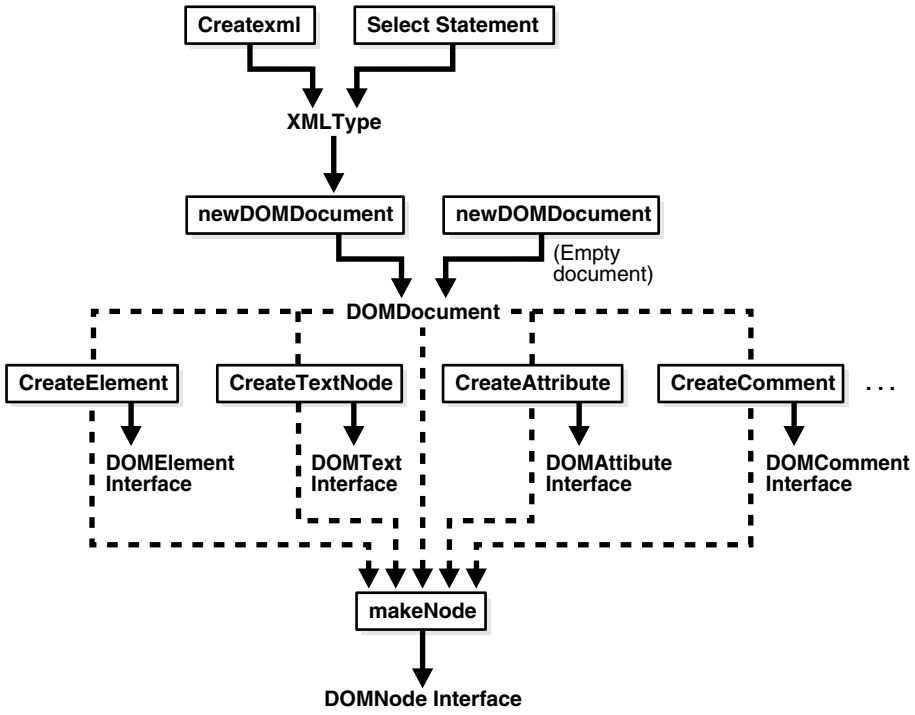
Using PL/SQL DOM API for XMLType (DBMS_XMLDOM)

Figure 11–1 illustrates how you use PL/SQL DOM API for `XMLType` (`DBMS_XMLDOM`).

You can create a DOM document (`DOMDocument`) from an existing `XMLType` or as an empty document.

1. The `newDOMDocument` procedure processes the `XMLType` instance or empty document. This creates a `DOMDocument` instance.
2. You can use DOM API methods such as `createElement`, `createText`, `createAttribute`, and `createComment` to traverse and extend the DOM tree.
3. The results of methods such as `DOMElement` and `DOMText` can also be passed to `makeNode` to obtain the `DOMNode` interface.

Figure 11–1 Using PL/SQL DOM API for XMLType



PL/SQL DOM API for XMLType – Examples

This section presents examples of using the PL/SQL DOM API for `XMLType`.

Remember to call procedure `freeDocument` for *each* `DOMDocument` instance, when you are through with the instance. You can still access `XMLType` instances on which `DOMDocument` instances were built, even after the `DOMDocuments` have been freed.

Example 11–1 Creating and Manipulating a DOM Document

This example creates a hierarchical, in-memory representation of an XML document – a DOM document. It uses a *handle* to this DOM document to manipulate it: print it, change part of it, and print it again after the change. Manipulating the DOM document by its handle also indirectly affects the XML data represented by the document, so that querying that data after the change shows the changed result.

The in-memory document is created from an `XMLType` variable using PL/SQL function `newDOMDocument`. The handle to this document is created using function `makeNode`. The document is written to a `VARCHAR2` buffer using function `writeToBuffer`, and the buffer is printed using `DBMS_OUTPUT.put_line`.

After manipulating the document using various `DBMS_XMLDOM` procedures, the (changed) data in the `XMLType` variable is inserted into a table and queried, showing the change. It is only when the data is inserted into a database table that it becomes persistent; until then, it exists in memory only. This persistence is demonstrated by the fact that the database query is made after the in-memory document (`DOMDocument` instance) has been freed.

```
CREATE TABLE person OF XMLType;

DECLARE
  var       XMLType;
  doc       DBMS_XMLDOM.DOMDocument;
```

```
  ndoc      DBMS_XMLDOM.DOMNode;
  docelem   DBMS_XMLDOM.DOMElement;
  node      DBMS_XMLDOM.DOMNode;
  childnode DBMS_XMLDOM.DOMNode;
  nodelist  DBMS_XMLDOM.DOMNodelist;
  buf       VARCHAR2(2000);
BEGIN
  var := XMLType('<PERSON><NAME>ramesh</NAME></PERSON>');

  -- Create DOMDocument handle
  doc     := DBMS_XMLDOM.newDOMDocument(var);
  ndoc    := DBMS_XMLDOM.makeNode(doc);

  DBMS_XMLDOM.writeToBuffer(ndoc, buf);
  DBMS_OUTPUT.put_line('Before:'||buf);

  docelem := DBMS_XMLDOM.getDocumentElement(doc);

  -- Access element
  nodelist := DBMS_XMLDOM.getElementsByTagName(docelem, 'NAME');
  node := DBMS_XMLDOM.item(nodelist, 0);
  childnode := DBMS_XMLDOM.getFirstChild(node);

  -- Manipulate element
  DBMS_XMLDOM.setNodeValue(childnode, 'raj');
  DBMS_XMLDOM.writeToBuffer(ndoc, buf);
  DBMS_OUTPUT.put_line('After:'||buf);
  DBMS_XMLDOM.freeDocument(doc);
  INSERT INTO person VALUES (var);
END;
/
```

This produces the following output:

```
Before:<PERSON>
  <NAME>ramesh</NAME>
</PERSON>

After:<PERSON>
  <NAME>raj</NAME>
</PERSON>
```

This query confirms that the data has changed:

```
SELECT * FROM person;
SYS_NC_ROWINFO$
---------------
<PERSON>
  <NAME>raj</NAME>
</PERSON>

1 row selected.
```

Example 11–2 Creating an Element Node and Obtaining Information About It

This example creates an empty DOM document, and then adds an element node (<ELEM>) to the document. DMBS_XMLDOM API node procedures are used to obtain the name (<ELEM>), value (NULL), and type (1 = element node) of the element node.

```
DECLARE
```

```
  doc   DBMS_XMLDOM.DOMDocument;
  elem  DBMS_XMLDOM.DOMElement;
  nelem DBMS_XMLDOM.DOMNode;
BEGIN
  doc := DBMS_XMLDOM.newDOMDocument;
  elem := DBMS_XMLDOM.createElement(doc, 'ELEM');
  nelem := DBMS_XMLDOM.makeNode(elem);
  DBMS_OUTPUT.put_line('Node name = ' || DBMS_XMLDOM.getNodeName(nelem));
  DBMS_OUTPUT.put_line('Node value = '|| DBMS_XMLDOM.getNodeValue(nelem));
  DBMS_OUTPUT.put_line('Node type = ' || DBMS_XMLDOM.getNodeType(nelem));
  DBMS_XMLDOM.freeDocument(doc);
END;
/
```

This produces the following output:

```
Node name = ELEM
Node value =
Node type = 1
```

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

The PL/SQL Parser API for `XMLType` (`DBMS_XMLPARSER`) builds a result tree that can be accessed by PL/SQL APIs. If parsing fails, it raises an error.

See *Oracle Database PL/SQL Packages and Types Reference* for descriptions of the individual methods of the PL/SQL Parser API for `XMLType` (`DBMS_XMLPARSER`).

The following `DBMS_XMLPARSER` methods are not supported:

- `parseDTD()`
- `parseDTDBuffer()`
- `parseDTDClob()`
- `setDocType()`
- `setErrorLog()`

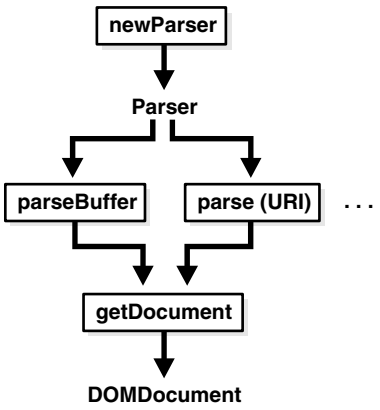
Using PL/SQL Parser API for XMLType (DBMS_XMLPARSER)

Figure 11–2 illustrates how to use the PL/SQL Parser for `XMLType` (`DBMS_XMLPARSER`). These are the steps:

1. Construct a parser instance using method.
2. Parse XML documents using methods such as `parseBuffer`, `parseClob`, and `parse(URI)`. An error is raised if the input is not a valid XML document.

3. Call `getDocument` on the parser to obtain a `DOMDocument` interface.

Figure 11–2 Using PL/SQL Parser API for XMLType



Example 11–3 Parsing an XML Document

This example parses a simple XML document. It creates an XML parser (instance of `DBMS_XMLPARSER.parser`) and uses it to parse the XML document (text) in variable `indoc`. Parsing creates a DOM document, which is retrieved from the parser using `DBMS_XMLPARSER.getDocument`. A DOM node is created that contains the entire document, and the node is printed. After freeing (destroying) the DOM document, the parser instance is freed using `DBMS_XMLPARSER.freeParser`.

```
DECLARE
  indoc    VARCHAR2(2000);
  indomdoc DBMS_XMLDOM.DOMDocument;
  innode   DBMS_XMLDOM.DOMNode;
  myparser DBMS_XMLPARSER.parser;
  buf      VARCHAR2(2000);
BEGIN
  indoc := '<emp><name>De Selby</name></emp>';
  myParser := DBMS_XMLPARSER.newParser;
  DBMS_XMLPARSER.parseBuffer(myParser, indoc);
  indomdoc := DBMS_XMLPARSER.getDocument(myParser);
  innode := DBMS_XMLDOM.makeNode(indomdoc);
  DBMS_XMLDOM.writeToBuffer(innode, buf);
  DBMS_OUTPUT.put_line(buf);
  DBMS_XMLDOM.freeDocument(indomdoc);
  DBMS_XMLPARSER.freeParser(myParser);
END;
/
```

This produces the following output:

```
<emp><name>De Selby</name></emp>
```

PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)

W3C XSL Recommendation describes rules for transforming a source tree into a result tree. A transformation expressed in Extensible Stylesheet Language Transformation (XSLT) language is called an XSL style sheet. The transformation specified is achieved by associating patterns with templates defined in the XSLT style sheet. A template is instantiated to create part of the result tree.

Enabling Transformations and Conversions with XSLT

The Oracle XML DB PL/SQL DOM API for `XMLType` also supports 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.

PL/SQL package `DBMS_XSLPROCESSOR` provides a convenient and efficient way of applying a single style sheet to multiple documents. The performance of this package is better than that of `transform()` because the style sheet is parsed only once.

Note: The XML Parser for PL/SQL in Oracle XDK parses an XML document (or a standalone DTD) so that the XML document can be processed by an application, typically running on the client. PL/SQL APIs for `XMLType` are used for applications that run on the server and are natively integrated in the database. Benefits include performance improvements and enhanced access and manipulation options.

See Also:

- Appendix C, "XSLT Primer"
- Chapter 9, "Transforming and Validating XMLType Data"

PL/SQL XSLT Processor for XMLType: Features

PL/SQL XSLT Processor for `XMLType` (`DBMS_XSLPROCESSOR`) is the Oracle XML DB implementation of the XSL processor. This follows the W3C XSLT final recommendation (REC-xslt-19991116). It includes the required action of an XSL processor in terms of how it must read XSLT style sheets and the transformations it must achieve.

The types and methods of the PL/SQL XSLT Processor API are made available by the PL/SQL package, `DBMS_XSLPROCESSOR`. The methods in this package use two PL/SQL datatypes specific to the XSL Processor implementation: `PROCESSOR` and `STYLESHEET` .

All `DBMS_XSLPROCESSOR` methods are supported by Oracle XML DB, with the *exception* of method `setErrorLog()`.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for descriptions of the individual `DBMS_XSLPROCESSOR` methods

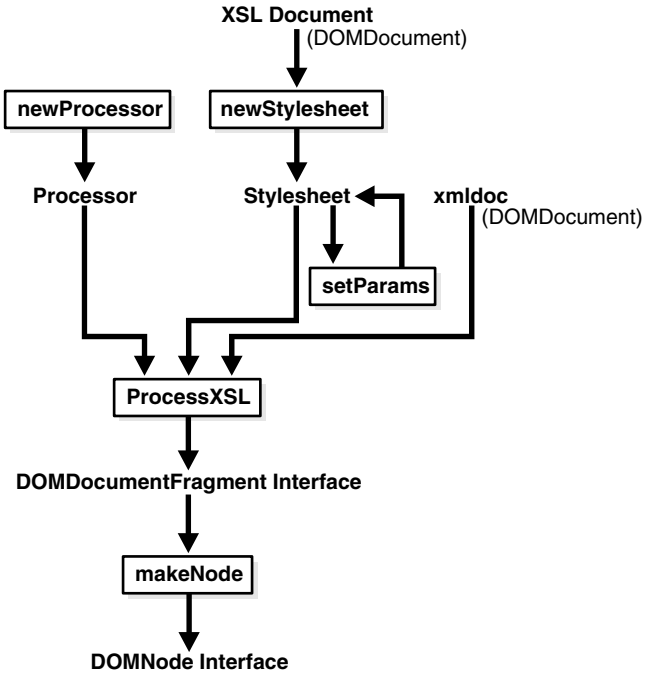
Using PL/SQL Parser API for XMLType (DBMS_XSLPROCESSOR)

Figure 11–3 illustrates how to use XSLT Processor for `XMLType` (`DBMS_XSLPROCESSOR`). These are the steps:

1. Construct an XSLT processor using `newProcessor`.
2. Use `newStylesheet` to build a `STYLESHEET` object from a DOM document.

3. Optionally, you can set parameters for the `STYLESHEET` object using `setParams`.
4. Use `processXSL` to transform a DOM document using the processor and `STYLESHEET` object.
5. Use the PL/SQL DOM API for `XMLType` to manipulate the result of XSLT processing.

Figure 11–3 Using PL/SQL XSLT Processor for XMLType



Example 11–4 Transforming an XML Document Using an XSL Style Sheet

This example transforms an XML document using procedure `processXSL`. It uses the same parser instance to create two different DOM documents: the XML text to transform and the XSLT style sheet. An XSL processor instance is created, which applies the style sheet to the source XML to produce a new DOM fragment. A DOM node (`outnode`) is created from this fragment, and the node content is printed. The output DOM fragment, parser, and XSLT processor instances are freed using procedures `freeDocFrag`, `freeParser`, and `freeProcessor`, respectively.

```
DECLARE
  indoc      VARCHAR2(2000);
  xsldoc     VARCHAR2(2000);
  myParser   DBMS_XMLPARSER.parser;
  indomdoc   DBMS_XMLDOM.DOMDocument;
  xsltdomdoc DBMS_XMLDOM.DOMDocument;
  xsl        DBMS_XSLPROCESSOR.stylesheet;
  outdomdocf DBMS_XMLDOM.DOMDocumentFragment;
  outnode    DBMS_XMLDOM.DOMNode;
  proc       DBMS_XSLPROCESSOR.processor;
  buf        VARCHAR2(2000);
BEGIN
  indoc := '<emp><empno>1</empno>
              <fname>robert</fname>
              <lname>smith</lname>
              <sal>1000</sal>
```

```
              <job>engineer</job>
             </emp>';
  xsldoc := '<?xml version="1.0"?>
             <xsl:stylesheet version="1.0"
                             xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
               <xsl:output encoding="utf-8"/>
               <!-- alphabetizes an xml tree -->
               <xsl:template match="*">
                 <xsl:copy>
                   <xsl:apply-templates select="*|text()">
                     <xsl:sort select="name(.)" data-type="text"
                               order="ascending"/>
                   </xsl:apply-templates>
                 </xsl:copy>
               </xsl:template>
               <xsl:template match="text()">
                 <xsl:value-of select="normalize-space(.)"/>
               </xsl:template>
             </xsl:stylesheet>';
  myParser := DBMS_XMLPARSER.newParser;
  DBMS_XMLPARSER.parseBuffer(myParser, indoc);
  indomdoc   := DBMS_XMLPARSER.getDocument(myParser);
  DBMS_XMLPARSER.parseBuffer(myParser, xsldoc);
  xsltdomdoc := DBMS_XMLPARSER.getDocument(myParser);
  xsl        := DBMS_XSLPROCESSOR.newStyleSheet(xsltdomdoc, '');
  proc       := DBMS_XSLPROCESSOR.newProcessor;
  --apply stylesheet to DOM document
  outdomdocf := DBMS_XSLPROCESSOR.processXSL(proc, xsl, indomdoc);
  outnode    := DBMS_XMLDOM.makeNode(outdomdocf);
  -- PL/SQL DOM API for XMLType can be used here
  DBMS_XMLDOM.writeToBuffer(outnode, buf);
  DBMS_OUTPUT.put_line(buf);
  DBMS_XMLDOM.freeDocument(indomdoc);
  DBMS_XMLDOM.freeDocument(xsltdomdoc);
  DBMS_XMLDOM.freeDocFrag(outdomdocf);
  DBMS_XMLPARSER.freeParser(myParser);
  DBMS_XSLPROCESSOR.freeProcessor(proc);
END;
/
```

This produces the following output:

```
<emp>
<empno>1</empno>
<fname>robert</fname>
<job>engineer</job>
<lname>smith</l
name>
<sal>1000</sal>
</emp>
```

12

Package DBMS_XMLSTORE

This chapter introduces you to the PL/SQL package DBMS_XMLSTORE. This package is used to insert, update, and delete data from XML documents in object-relational tables.

This chapter contains these topics:

Overview of PL/SQL Package DBMS_XMLSTORE

The DBMS_XMLSTORE package enables DML operations to be performed on relational tables using XML. It takes a canonical XML mapping similar to that produced by package DBMS_XMLGEN; converts it to object-relational constructs; and 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, so it 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 CLOB and VARCHAR.
- PL/SQL functions insertXML, updateXML, and deleteXML, which are also present in package DBMS_XMLSAVE, have been enhanced in package DBMS_XMLSTORE to take XMLType instances in addition to CLOB values and strings. This provides for better integration with Oracle XML DB functionality.

Using Package DBMS_XMLSTORE

To use PL/SQL package DBMS_XMLSTORE, follow these steps:

1. Create a context handle by calling function DBMS_XMLSTORE.**newContext** and supplying it with the table name to use for the DML operations. For case sensitivity, double-quote the string that is passed to the function.

By default, XML documents are expected to identify rows with the <ROW> tag. This is the same default used by package DBMS_XMLGEN when generating XML. This may be overridden by calling the setRowTag function.

2. ***For Inserts:*** You can set the list of columns to insert calling function DBMS_XMLSTORE.**setUpdateColumn** for each column. This is highly recommended, since it will improve performance. The default behavior is to insert values for all of the columns whose corresponding elements are present in the XML document.
3. ***For Updates:*** You must specify one or more (pseudo-) key columns using function DBMS_XMLSTORE.**setKeyColumn**. These are used to specify which *rows* are to be updated. In SQL, you would do that using a WHERE clause in an UPDATE statement, specifying a combination of columns that uniquely identify the rows to be updated. The columns that you use with setKeyColumn need *not* be actual keys of the table—as long as they uniquely specify a row, they can be used.

   For example, in the employees table, column employee_id uniquely identifies rows (it is in fact a key of the table). If the XML document that you use to update the table contains element <EMPLOYEE_ID>2176</EMPLOYEE_ID>, then the rows where employee_id equals 2176 are updated.

   The list of *update* columns can also be specified, using DBMS_XMLSTORE.setUpdateColumn. This is recommended, for better performance. The default behavior is to update *all* of the columns in the row(s) identified by setKeyColumn whose corresponding elements are present in the XML document.
4. ***For Deletions:*** As for updates, you specify (pseudo-) key columns to identify the row(s) to delete.
5. Provide a document to PL/SQL function insertXML, updateXML, or deleteXML.
6. This last step may be repeated multiple times, with several XML documents.
7. Close the context with function DBMS_XMLSTORE.**closeContext**.

Inserting with DBMS_XMLSTORE

To insert an XML document into a table or view, you supply the table or view name and the document. DBMS_XMLSTORE parses the document and then creates an INSERT statement into which it binds all the values. By default, DBMS_XMLSTORE inserts values into all the columns represented by elements in the XML document.

Example 12–1 Inserting data with specified columns

This example uses DBM_XMLSTORE to insert the information for two new employees into the employees table. The information is provided in the form of XML data.

```
SELECT employee_id AS EMP_ID, salary, hire_date, job_id, email, last_name
  FROM employees WHERE department_id = 30;

EMP_ID     SALARY HIRE_DATE JOB_ID     EMAIL       LAST_NAME
------ ---------- --------- ---------- ---------- ----------
   114      11000 07-DEC-94 PU_MAN     DRAPHEAL     Raphaely
   115       3100 18-MAY-95 PU_CLERK   AKHOO            Khoo
   116       2900 24-DEC-97 PU_CLERK   SBAIDA          Baida
   117       2800 24-JUL-97 PU_CLERK   STOBIAS        Tobias
   118       2600 15-NOV-98 PU_CLERK   GHIMURO        Himuro
   119       2500 10-AUG-99 PU_CLERK   KCOLMENA   Colmenares

6 rows selected.
```

```
DECLARE
  insCtx DBMS_XMLSTORE.ctxType;
  rows NUMBER;
  xmlDoc CLOB :=
    '<ROWSET>
       <ROW num="1">
         <EMPLOYEE_ID>920</EMPLOYEE_ID>
         <SALARY>1800</SALARY>
         <DEPARTMENT_ID>30</DEPARTMENT_ID>
         <HIRE_DATE>17-DEC-2002</HIRE_DATE>
         <LAST_NAME>Strauss</LAST_NAME>
         <EMAIL>JSTRAUSS</EMAIL>
         <JOB_ID>ST_CLERK</JOB_ID>
       </ROW>
       <ROW>
         <EMPLOYEE_ID>921</EMPLOYEE_ID>
         <SALARY>2000</SALARY>
         <DEPARTMENT_ID>30</DEPARTMENT_ID>
         <HIRE_DATE>31-DEC-2004</HIRE_DATE>
         <LAST_NAME>Jones</LAST_NAME>
         <EMAIL>EJONES</EMAIL>
         <JOB_ID>ST_CLERK</JOB_ID>
       </ROW>
     </ROWSET>';
BEGIN
  insCtx := DBMS_XMLSTORE.newContext('HR.EMPLOYEES'); -- Get saved context
  DBMS_XMLSTORE.clearUpdateColumnList(insCtx); -- Clear the update settings

  -- Set the columns to be updated as a list of values
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'EMPLOYEE_ID');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'SALARY');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'HIRE_DATE');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'DEPARTMENT_ID');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'JOB_ID');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'EMAIL');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'LAST_NAME');

  -- Insert the doc.
  rows := DBMS_XMLSTORE.insertXML(insCtx, xmlDoc);
  DBMS_OUTPUT.put_line(rows || ' rows inserted.');

  -- Close the context
  DBMS_XMLSTORE.closeContext(insCtx);
END;
/

2 rows inserted.

PL/SQL procedure successfully completed.

SELECT employee_id AS EMP_ID, salary, hire_date, job_id, email, last_name
  FROM employees WHERE department_id = 30;

EMP_ID     SALARY HIRE_DATE JOB_ID     EMAIL       LAST_NAME
------ ---------- --------- ---------- ---------- ----------
   114      11000 07-DEC-94 PU_MAN     DRAPHEAL     Raphaely
   115       3100 18-MAY-95 PU_CLERK   AKHOO            Khoo
   116       2900 24-DEC-97 PU_CLERK   SBAIDA          Baida
   117       2800 24-JUL-97 PU_CLERK   STOBIAS        Tobias
```

```
   118       2600 15-NOV-98 PU_CLERK   GHIMURO        Himuro
   119       2500 10-AUG-99 PU_CLERK   KCOLMENA   Colmenares
   920       1800 17-DEC-02 ST_CLERK   STRAUSS       Strauss
   921       2000 31-DEC-04 ST_CLERK   EJONES          Jones

8 rows selected.
```

Updating with DBMS_XMLSTORE

To update (modify) existing data using package `DBMS_XMLSTORE`, you must specify which rows to update. In SQL, you would do that using a `WHERE` clause in an `UPDATE` statement. With `DBMS_XMLSTORE`, you do it by calling procedure `setKeyColumn` once for *each* of the columns that are used collectively to identify the row.

You can think of this set of columns as acting like a set of key columns: *together, they specify a unique row* to be updated. However, the columns that you use (with `setKeyColumn`) need *not* be actual keys of the table—as long as they uniquely specify a row, they can be used with calls to `setKeyColumn`.

Example 12–2 Updating Data With Key Columns

This example uses `DBM_XMLSTORE` to update information. Assuming that the first name for employee number 188 is incorrectly recorded as Kelly, this example corrects that first name to Pat. Since column `employee_id` is in fact a primary key for table `employees`, a single call to `setKeyColumn` specifying column `employee_id` is sufficient to identify a unique row for updating.

```
SELECT employee_id, first_name FROM employees WHERE employee_id = 188;

EMPLOYEE_ID FIRST_NAME
----------- ----------
        188 Kelly

1 row selected.

DECLARE
  updCtx DBMS_XMLSTORE.ctxType;
  rows NUMBER;
  xmlDoc CLOB :=
    '<ROWSET>
       <ROW>
         <EMPLOYEE_ID>188</EMPLOYEE_ID>
         <FIRST_NAME>Pat</FIRST_NAME>
       </ROW>
     </ROWSET>';
BEGIN
   updCtx := DBMS_XMLSTORE.newContext('HR.EMPLOYEES'); -- get the context
   DBMS_XMLSTORE.clearUpdateColumnList(updCtx);        -- clear update settings

   -- Specify that column employee_id is a "key" to identify the row to update.
   DBMS_XMLSTORE.setKeyColumn(updCtx, 'EMPLOYEE_ID');
   rows := DBMS_XMLSTORE.updateXML(updCtx, xmlDoc);    -- update the table
   DBMS_XMLSTORE.closeContext(updCtx);                 -- close the context
END;
/

SELECT employee_id, first_name FROM employees WHERE employee_id = 188;

EMPLOYEE_ID FIRST_NAME
----------- ----------
```

```
        188 Pat

1 row selected.
```

This UPDATE statement is equivalent to the use of DBM_XMLSTORE in this example:

```
UPDATE hr.employees SET first_name = 'Pat' WHERE employee_id = 188;
```

Deleting with DBMS_XMLSTORE

Deletions are treated similarly to updates: you specify the (pseudo-) key columns that identify the rows to delete.

Example 12–3 Simple deleteXML() Example

```
SELECT employee_id FROM employees WHERE employee_id = 188;

EMPLOYEE_ID
-----------
        188

1 row selected.

DECLARE
  delCtx DBMS_XMLSTORE.ctxType;
  rows NUMBER;
  xmlDoc CLOB :=
    '<ROWSET>
       <ROW>
         <EMPLOYEE_ID>188</EMPLOYEE_ID>
         <DEPARTMENT_ID>50</DEPARTMENT_ID>
       </ROW>
     </ROWSET>';
BEGIN
  delCtx  := DBMS_XMLSTORE.newContext('HR.EMPLOYEES');
  DBMS_XMLSTORE.setKeyColumn(delCtx, 'EMPLOYEE_ID');
  rows := DBMS_XMLSTORE.deleteXML(delCtx, xmlDoc);
  DBMS_XMLSTORE.closeContext(delCtx);
END;
/

SELECT employee_id FROM employees WHERE employee_id = 188;

no rows selected.
```

13

Java API for XMLType

This chapter describes how to use `XMLType` in Java, including fetching `XMLType` data through Java Database Connectivity (JDBC).

This chapter contains these topics:

- Overview of Java DOM API for XMLType
- Java DOM API for XMLType
- Loading a Large XML Document into the Database with JDBC
- Java DOM API for XMLType Features
- Java DOM API for XMLType Classes

Overview of Java DOM API for XMLType

Oracle XML DB supports the Java Document Object Model (DOM) Application Program Interface (API) for XMLType. This is a generic API for client and server, for both XML schema-based and non-schema-based documents. It is implemented using 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 Database XML Java API Reference*

Java DOM API for XMLType

Java DOM API for `XMLType` handles all kinds of valid XML documents irrespective of how they are stored in Oracle XML DB. It presents to the application a uniform view of the XML document 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 11, "PL/SQL API for XMLType", the Oracle XML DB DOM APIs are compliant with the 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 method `getBlobVal()` to retrieve the XML contents in the requested character set.

Accessing XML Documents in Repository

Oracle XML DB resource API for Java API allows Java applications to access XML documents stored in Oracle XML DB Repository. Naming conforms to the Java binding for DOM as specified by the W3C DOM Recommendation. The repository hierarchy can store both XML schema-based and non-schema-based documents.

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, method `createXML()`.

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 SQL datatype `XMLType`. The Java (JDBC) API for `XMLType` interface can implement the DOM document interface.

Example 13–1 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 13–2 XMLType Java: Selecting XMLType Data

You can select the `XMLType` data in JDBC in one of two ways:

- Use method `getClobVal()`, `getStringVal()` or `getBlobVal(csid)` in SQL, and obtain 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.createXML(orset.getOPAQUE(1));

// get the XML as a string...
String poString = poxml.getStringVal();
```

Example 13–3 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 13–4 XMLType Java: Returning XMLType Data

This example illustrates how to bind an `OUT` variable of `XMLType` to a SQL statement. The output parameter is registered as type `XMLType`.

```
public void doCall (String[] args)
  throws Exception
  {

//  CREATE OR REPLACE FUNCTION getPurchaseOrder(reference VARCHAR2)
//  RETURN XMLTYPE
//  AS
//    xml XMLTYPE;
//  BEGIN
//    SELECT OBJECT_VALUE INTO xml
//      FROM purchaseorder
//      WHERE extractValue(OBJECT_VALUE,'/PurchaseOrder/Reference') = reference;
//      RETURN xml;
//  END;

    String SQLTEXT = "{? = call getPurchaseOrder('BLAKE-20021009123333601PDT')}";
    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 Java Database Connectivity (JDBC).

> **Note:** `XMLType` methods `extract()`, `transform()`, and `existsNode()` only work with the *thick* JDBC driver.
>
> Not all `oracle.xdb.XMLType` functions are supported by the thin JDBC driver. If you do not use `oracle.xdb.XMLType` classes and OCI driver, you could lose performance benefits associated with the intelligent handling of XML.

Example 13–5 XMLType Java: Updating, Inserting, or Deleting XMLType Data

You can insert an `XMLType` in Java in one of two ways:

- Bind a CLOB instance or a string to an INSERT, UPDATE, or DELETE statement, and use the XMLType constructor inside SQL to construct the XML instance:

```
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 13–6 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 13–7 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 Database Connectivity (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
----------------------

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 13–8 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 13–9 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 13–9 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*, you get an **`XML`**`Document` instance
- If the connection is *thick* or *kprb*, you get an **`XDB`**`Document` 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, schema-based or non-schema-based
- Any type of underlying storage used by the document:
  - Character Large Object (CLOB)
  - Binary Large Object (BLOB)
  - object-relational

The Java DOM API for `XMLType` supports deep 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

The Java API for `XMLType` also allows applications to create XML documents programmatically. That way, applications can create XML documents on the fly (or dynamically). Such documents can conform to a registered XML schema or not.

Creating XML Schema-Based Documents

To create XML schema-based documents, Java DOM API for `XMLType` uses an extension to specify which XML schema URL to use. For XML schema-based documents, it also verifies that the DOM being created conforms to the specified XML schema, that is, that the appropriate children are being inserted under the appropriate documents.

> **Note:** 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-schema-based documents.
- In the format specified by the XML schema for XML schema-based documents.

Example 13–10 Creating a DOM Object with the Java DOM API

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

`XDBDocument` is a class that represents the DOM for the instantiated XML document. You can retrieve the `XMLType` value from the XML document using the constructor `XMLType` constructor that takes a `Document` argument:

```
XMLType createXML(Connection conn, Document domdoc)
```

Table 13–1 lists the Java DOM API for `XMLType` classes and the W3C DOM interfaces they implement.

Table 13–1 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`

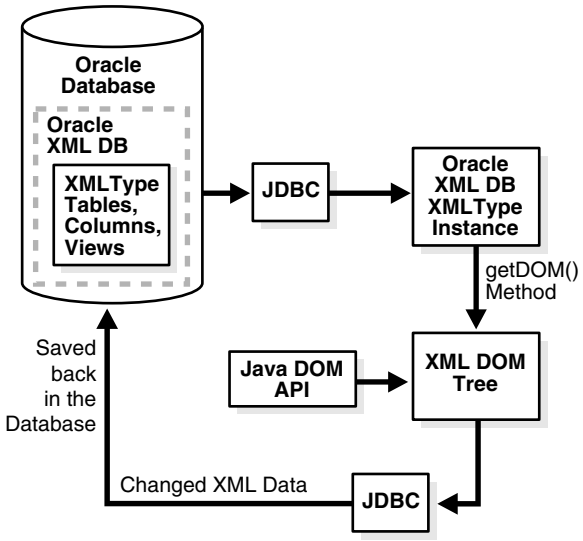
Using Java DOM API for XMLType

Figure 13–1 illustrates how to use the Java DOM API for `XMLType`.[1] These are the steps:

1. Retrieve the XML data from the `XMLType` table or `XMLType` column in the table. When you fetch XML data, Oracle creates 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 manipulate elements of the DOM tree. The XMLType instance holds the modified data, but the data is sent back using a JDBC update.

The `XMLType` and `XDBDocument` instances should be closed using the `close()` method in the respective classes. This releases any underlying memory that is held.

Figure 13–1 Using Java DOM API for XMLType



[1] This assumes that your XML data is pre-registered with an XML schema, and that it is stored in an `XMLType` column.

14

Using the C API for XML

This chapter provides a guideline for using the C API for XML with Oracle XML DB.

This chapter contains these topics:

- Overview of the C API for XML (XDK and Oracle XML DB)
- Using OCI and the C API for XML with Oracle XML DB
- XML Context Parameter
- Initializing and Terminating an XML Context
- OCI Usage
- Common XMLType Operations in C

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide* "XML Parser for C"

Overview of 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[1] API for XML. It can be used for XML data that is inside or outside the database. This API also includes performance-improving extensions that you can use:

- In 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 the functionality needed by XDK and Oracle XML DB applications. It provides XSLT and XML Schema implementations. Although DOM 2.0 Recommendation was followed

[1] DOM refers to compliance with the World Wide Web Consortium (W3C) DOM 2.0 Recommendation.

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()`.

The 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 Parameter. 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 Parameter

An *XML context* is a required parameter to all the C DOM API functions. This opaque context encapsulates information about the data encoding, the error message language, and so on. The contents of the context are different for Oracle XDK applications and Oracle XML DB. For Oracle XML DB, there are two OCI functions that initialize (`OCIXmlDbInitXmlCtx()`) and terminate (`OCIXmlDbFreeXmlCtx()`) an XML context.

OCIXmlDbInitXmlCtx() Syntax

The syntax of function `OCIXmlDbInitXmlCtx()` is as follows:

```
xmlctx *OCIXmlDbInitXMlCtx (OCIEnv        *envhp,
                            OCISvcHp      *svchp,
                            OCIError      *errhp,
                            ocixmldbparam *params,
                            ub4           num_params );
```

Table 14–1 describes the parameters.

Table 14–1 OCIXmlDbInitXMlCtx() Parameters

Parameter	Description
`envhp (IN)`	The OCI environment handle.
`svchp (IN)`	The OCI service handle.
`errhp (IN)`	The OCI error handle.

Table 14–1 (Cont.) OCIXmlDbInitXMlCtx() Parameters

Parameter	Description
`params (IN)`	An array of optional values: ▪ OCI duration. Default value is `OCI_DURATION_SESSION`. ▪ Error handler, which is a user-registered callback: `void (*err_handler) (sword errcode, (CONST OraText *) errmsg);` `void (*err_handler) (sword errcode, (CONST OraText *) errmsg);`
`num_params (IN)`	Number of parameters to be read from `params`.

OCIXmlDbFreeXmlCtx() Syntax

The syntax of function `OCIXmlDbFreeXmlCtx()` is as follows, where parameter `xctx (IN)` is the XML context to terminate.:

```
void OCIXmlDbFreeXmlCtx (xmlctx *xctx);
```

Initializing and Terminating an XML Context

Example 14–1 is a C program that uses the C DOM API to construct an XML document and save it to Oracle Database in table `my_table`. It calls OCI functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()` to initialize and terminate the XML context. These OCI functions are defined in header file `ocixmldb.h`.

The C code in Example 14–1 assumes that the following SQL code has first been executed to create table `my_table` in database schema:

```
CONNECT CAPIUSER/CAPIUSER

CREATE TABLE my_table OF XMLType;
```

Example 14–2 queries table `my_table` to show the data that was inserted by Example 14–1.

Example 14–1 Using OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx()

This example shows how to use OCI functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()` to initialize and terminate the XML context. It constructs an XML document using the C DOM API and saves it to the database. The code uses helper functions `exec_bind_xml`, `init_oci_handles`, and `free_oci_handles`, which are not listed here. The complete listing of this example, including the helper functions, can be found in Appendix D, "Oracle-Supplied XML Schemas and Examples", "Initializing and Terminating an XML Context (OCI)" on page D-24.

```
#ifndef S_ORACLE
#include <s.h>
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
#ifndef XML_ORACLE
#include <xml.h>
#endif
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
```

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>

typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
        oratext *password;
} test_ctx;

/* Helper function 1: execute a sql statement which binds xml data */
STATICF sword exec_bind_xml(OCISvcCtx *svchp,
                    OCIError *errhp,
                    OCIStmt *stmthp,
                    void *xml,
                    OCIType *xmltdo,
                    OraText *sqlstmt);

/* Helper function 2: Initialize OCI handles and connect */
STATICF sword init_oci_handles(test_ctx *ctx);

/* Helper function 3: Free OCI handles and disconnect */
STATICF sword free_oci_handles(test_ctx *ctx);

void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;

  oratext ins_stmt[] = "insert into my_table values (:1)";
  oratext tlpxml_test_sch[] = "<TOP/>";
  ctx->username = (oratext *)"CAPIUSER";
  ctx->password = (oratext *)"CAPIUSER";

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

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

  /* Start processing - first, check that this DOM supports XML 1.0 */
  printf("\n\nSupports XML 1.0? : %s\n",
         XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
         "YES" : "NO");
```

```
  /* Parse a document */
  if (!(doc = XmlLoadDom(xctx, &err, "buffer", tlpxml_test_sch,
                         "buffer_length", sizeof(tlpxml_test_sch)-1,
                         "validate", TRUE, NULL)))
  {
    printf("Parse failed, code %d\n", err);
  }
  else
  {
    /* Get the document element */
    top = (xmlnode *)XmlDomGetDocElem(xctx, doc);

    /* Print out the top element */
    printf("\n\nOriginal top element is :\n");
    XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

    /* Print out the document-note that the changes are reflected here */
    printf("\n\nOriginal document is :\n");
    XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

    /* Create some elements and add them to the document */
    quux = (xmlnode *) XmlDomCreateElem(xctx ,doc, (oratext *) "QUUX");
    foo = (xmlnode *) XmlDomCreateElem(xctx, doc, (oratext *) "FOO");
    foo_data = (xmlnode *) XmlDomCreateText(xctx, doc, (oratext *) "data");
    foo_data = XmlDomAppendChild(xctx, (xmlnode *) foo, (xmlnode *) foo_data);
    foo = XmlDomAppendChild(xctx, quux, foo);
    quux = XmlDomAppendChild(xctx, top, quux);

    /* Print out the top element */
    printf("\n\nNow the top element is :\n");
    XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

    /* Print out the document. Note that the changes are reflected here */
    printf("\n\nNow the document is :\n");
    XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

    /* Insert the document into my_table */
    status = OCITypeByName(ctx->envhp, ctx->errhp, ctx->svchp,
                           (const text *) "SYS", (ub4) strlen((char *)"SYS"),
                           (const text *) "XMLTYPE",
                           (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
                           (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                           (OCIType **) &xmltdo);
    if (status == OCI_SUCCESS)
    {
      exec_bind_xml(ctx->svchp, ctx->errhp, ctx->stmthp, (void *)doc, xmltdo,
                    ins_stmt);
    }
  }
  /* Free xml ctx */
  OCIXmlDbFreeXmlCtx(xctx);

  /* Free envhp, svchp, errhp, stmthp */
  free_oci_handles(ctx);
}
```

The output from compiling and running this C program is as follows:

```
Supports XML 1.0? : YES
```

```
Original top element is :
<TOP/>

Original document is :
<TOP/>

Now the top element is :
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>

Now the document is :
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>
```

This is the result of querying the constructed document in `my_table`:

```
SELECT * FROM my_table;

SYS_NC_ROWINFO$
---------------
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>

1 row selected.
```

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

You can construct new `XMLType` instances on the client using `XmlLoadDom()`, as follows:

1. Initialize the `xmlctx` as in Example 14–1.
2. Construct the XML data from a user buffer, local file, or URI. The return value, a (`xmldocnode*`), can be used in the rest of the common C API.
3. If required, you can cast (`xmldocnode *`) to (`void*`) and provide it directly as the bind value.

You can construct empty `XMLType` instances with `XMLCreateDocument()`. This is similar to using `OCIObjectNew()` for other types.

Common XMLType Operations in C

Table 14–2 provides the `XMLType` functional equivalent of common XML operations.

Table 14–2 Common XMLType Operations in C

Description	C API XMLType Function
Create empty `XMLType` instance	`XmlCreateDocument()`
Create from a source buffer	`XmlLoadDom()`
Extract an XPath expression	`XmlXPathEvalexpr()` and family
Transform using an XSLT 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 14–2 Using the DOM to Count Ordered Parts

This example shows how to use the DOM to determine how many instances of a particular part have been ordered. The part in question has `Id 37429158722`. See Appendix D, "Oracle-Supplied XML Schemas and Examples", Example D–4 on page D-24 for the definitions of helper functions `exec_bind_xml`, `free_oci_handles`, and `init_oci_handles`.

```
#ifndef S_ORACLE
#include <s.h>
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
#ifndef XML_ORACLE
#include <xml.h>
#endif
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>

typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
```

```
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
        oratext *password;
} test_ctx;

/* Helper function 1: execute a sql statement which binds xml data */
STATICF sword exec_bind_xml(OCISvcCtx *svchp,
                    OCIError *errhp,
                    OCIStmt *stmthp,
                    void *xml,
                    OCIType *xmltdo,
                    OraText *sqlstmt);

/* Helper function 2: Initialize OCI handles and connect */
STATICF sword init_oci_handles(test_ctx *ctx);

/* Helper function 3: Free OCI handles and disconnect */
STATICF sword free_oci_handles(test_ctx *ctx);

void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;
  ub4 xmlsize = 0;
  OCIDefine *defnp = (OCIDefine *) 0;
  oratext sel_stmt[] = "SELECT SYS_NC_ROWINFO$ FROM PURCHASEORDER";
  xmlnodelist *litems = (xmlnodelist *)0;
  xmlnode *item = (xmlnode *)item;
  xmlnode *part;
  xmlnamedmap *attrs;
  xmlnode *id;
  xmlnode *qty;
  oratext *idval;
  oratext *qtyval;
  ub4 total_qty;
  int i;
  int numdocs;

  ctx->username = (oratext *)"OE";
  ctx->password = (oratext *)"OE";

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

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

```
  /* Start processing */
  printf("\n\n Supports XML 1.0? : %s\n",
         XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
         "YES" : "NO");

  /* Get the documents from the database using a select statement */
  status = OCITypeByName(ctx->envhp, ctx->errhp, ctx->svchp, (const text *) "SYS",
                         (ub4) strlen((char *)"SYS"), (const text *) "XMLTYPE",
                         (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
                         (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                         (OCIType **) &xmltdo);
  status = OCIStmtPrepare(ctx->stmthp, ctx->errhp,
                 (CONST OraText *)sel_stmt, (ub4) strlen((char *)sel_stmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  status = OCIDefineByPos(ctx->stmthp, &defnp, ctx->errhp, (ub4) 1, (dvoid *) 0,
                 (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                 (ub2 *)0, (ub4) OCI_DEFAULT);
  status = OCIDefineObject(defnp, ctx->errhp, (OCIType *) xmltdo,
                  (dvoid **) &doc,
                  &xmlsize, (dvoid **) 0, (ub4 *) 0);
  status = OCIStmtExecute(ctx->svchp, ctx->stmthp, ctx->errhp, (ub4) 0, (ub4) 0,
                 (CONST OCISnapshot*) 0, (OCISnapshot*) 0, (ub4) OCI_DEFAULT);

  /* Initialize variables */
  total_qty = 0;
  numdocs = 0;

  /* Loop through all the documents */
  while ((status = OCIStmtFetch2(ctx->stmthp, ctx->errhp, (ub4) 1, (ub4) OCI_
FETCH_NEXT,
                                 (ub4)1, (ub4) OCI_DEFAULT)) == 0)
  {
    numdocs++;

    /* Get all the LineItem elements */
    litems = XmlDomGetDocElemsByTag(xctx, doc, (oratext *)"LineItem");
    i = 0;

    /* Loop through all LineItems */
    while (item = XmlDomGetNodeListItem(xctx, litems, i))
    {
      /* Get the part */
      part = XmlDomGetLastChild(xctx, item);

      /* Get the attributes */
      attrs = XmlDomGetAttrs(xctx, (xmlelemnode *)part);

      /* Get the id attribute and its value */
      id = XmlDomGetNamedItem(xctx, attrs, (oratext *)"Id");
      idval = XmlDomGetNodeValue(xctx, id);

      /* We are only interested in parts with id 37429158722 */
      if (idval && (strlen((char *)idval) == 11 )
          && !strncmp((char *)idval, (char *)"37429158722", 11))
      {
        /* Get the quantity attribute and its value.*/
        qty = XmlDomGetNamedItem(xctx, attrs, (oratext *)"Quantity");
        qtyval = XmlDomGetNodeValue(xctx, qty);

        /* Add the quantity to total_qty */
```

```
        total_qty += atoi((char *)qtyval);
       }
       i++;
     }
     XmlFreeDocument(xctx, doc);
     doc = (xmldocnode *)0;
   }
   printf("Total quantity needed for part 37429158722 = %d\n", total_qty);
   printf("Number of documents in table PURCHASEORDER = %d\n", numdocs);

   /* Free Xml Ctx */
   OCIXmlDbFreeXmlCtx(xctx);

   /* Free envhp, svchp, errhp, stmthp */
   free_oci_handles(ctx);
}
```

The output from compiling and running this C program is as follows:

```
Supports XML 1.0? : YES
Total quantity needed for part 37429158722 = 42
Number of documents in table PURCHASEORDER = 132
```

15

Using Oracle Data Provider for .NET with Oracle XML DB

Oracle Data Provider for Microsoft .NET (ODP.NET) is an implementation of a data provider for Oracle Database. It uses Oracle native APIs to offer fast and reliable access to Oracle data and features from any .NET application. It also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library. ODP.NET supports the following LOB datatypes natively with .NET: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`.

This chapter describes how to use ODP.NET with Oracle XML DB. It contains these topics:

ODP.NET XML Support and Oracle XML DB

ODP.NET supports XML natively in the database, through Oracle XML DB. ODP.NET XML support includes the following features:

- Stores XML data natively in Oracle Database as `XMLType`.
- Accesses relational and object-relational data as XML data from Oracle Database to a Microsoft .NET environment, and processes the XML using Microsoft .NET framework.
- Saves changes to the database server using XML data.

For the .NET application developer, these features include the following:

- Enhancements to the `OracleCommand`, `OracleConnection`, and `OracleDataReader` classes. Provides the following XML-specific classes:
  - `OracleXmlType`
  - `OracleXmlStream`
  - `OracleXmlQueryProperties`
  - `OracleXmlSaveProperties`

ODP.NET Sample Code

Example 15–1 retrieves `XMLType` data from the database to .NET and outputs the results:

Example 15–1 Retrieve XMLType Data to .NET

```
//Create OracleCommand and query XMLType
OracleCommand xmlCmd = new OracleCommand();
poCmd.CommandText = "SELECT po FROM po_tab";
poCmd.Connection = conn;
// Execute OracleCommand and output XML results to an OracleDataReader
OracleDataReader poReader = poCmd.ExecuteReader();
// ODP.NET native XML data type object from Oracle XML DB
OracleXmlType poXml;
string str = ""; //read XML results
while (poReader.Read())
{
  // Return OracleXmlType object of the specified XmlType column
  poXml = poReader.GetOracleXmlType(0);
  // Concatenate output for all the records
  str = str + poXml.Value;
} //Output XML results to the screen
Console.WriteLine(str);
```

See Also: *Oracle Data Provider for .NET Developer's Guide* for complete information about Oracle .NET support for Oracle XML DB.

Part IV

Viewing Existing Data as XML

Part IV of this manual introduces you to ways you can view your existing data as XML. It contains the following chapters:

16

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.

Overview of Generating XML Using Standard SQL/XML Functions

You can generate XML data using any of the following standard SQL/XML functions supported by Oracle XML DB:

- XMLROOT SQL Function on page 16-20
- XMLSERIALIZE SQL Function on page 16-20
- XMLPARSE SQL Function on page 16-21

Overview of Generating XML Using Oracle Database SQL Functions

You can generate XML data using any of the following Oracle Database SQL functions:

- XMLSEQUENCE SQL Function on page 16-11 . Only the cursor version of this function generates XML.
- XMLCOLATTVAL SQL Function on page 16-22
- XMLCDATA SQL Function on page 16-23
- Generating XML Using SQL Function SYS_XMLGEN on page 16-48. This operates on rows, generating XML documents.
- Generating XML Using SQL Function SYS_XMLAGG on page 16-56. This operates on groups of rows, aggregating several XML documents into one.

Overview of Generating XML Using DBMS_XMLGEN

You can generate XML from SQL queries using PL/SQL package `DBMS_XMLGEN`.

See Also:

- "Generating XML Using DBMS_XMLGEN" on page 16-24
- *Oracle Database PL/SQL Packages and Types Reference*, description of `DBMS_XMLGEN`

Overview of Generating XML with XSQL Pages Publishing Framework

You can generate XML using XSQL Pages Publishing Framework. XSQL Pages Publishing Framework, also known as XSQL Servlet, is part of the XDK for Java.

See Also: Generating XML Using XSQL Pages Publishing Framework on page 16-56

Overview of Generating XML Using XML SQL Utility (XSU)

You can use XML SQL Utility (XSU) to perform the following tasks on data in `XMLType` tables and columns:

- Transform data retrieved from object-relational database tables or views into XML.
- Extract data from an XML document, and using a canonical mapping, insert the data into appropriate columns or attributes of a table or a view.
- Extract data from an XML document and apply this data to updating or deleting values of the appropriate columns or attributes.

See Also:

- "Generating XML Using XML SQL Utility (XSU)" on page 16-59
- Chapter 3, "Using Oracle XML DB"
- Chapter 9, "Transforming and Validating XMLType Data"
- Chapter 11, "PL/SQL API for XMLType"
- Chapter 13, "Java API for XMLType"

Overview of Generating XML Using DBURIType

You can use an instance of `DBURIType` to construct XML documents that contain database data and whose structure reflects the database structure.

See Also: Chapter 19, "Accessing Data Through URIs"

Generating XML Using SQL Functions

This section describes Oracle XML DB SQL functions that you can use to generate XML data. Many of these functions belong to the SQL/XML standard, a SQL standard for XML:

- XMLELEMENT and XMLATTRIBUTES SQL Functions on page 16-4
- XMLFOREST SQL Function on page 16-9
- XMLCONCAT SQL Function on page 16-14
- XMLAGG SQL Function on page 16-15
- XMLPI SQL Function on page 16-18
- XMLCOMMENT SQL Function on page 16-19
- XMLROOT SQL Function on page 16-20
- XMLSERIALIZE SQL Function on page 16-20
- XMLPARSE SQL Function on page 16-21

The SQL/XML standard is ISO/IEC 9075–14:2005(E), Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML). As part of the SQL standard, it is aligned with SQL:2003. It is being developed under the auspices of these two standards bodies:

- ISO/IEC JTC1/SC32 ("International Organization for Standardization and International Electrotechnical Committee Joint Technical Committee 1, Information technology, Subcommittee 32, Data Management and Interchange").
- INCITS Technical Committee H2 ("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. Committee H2 is the committee responsible for SQL and SQL/MM.

This standardization process is ongoing. Please refer to `http://www.sqlx.org` for the latest information about `XMLQuery` and `XMLTable`.

Other XML-generating SQL functions presented in this section are Oracle Database-specific:

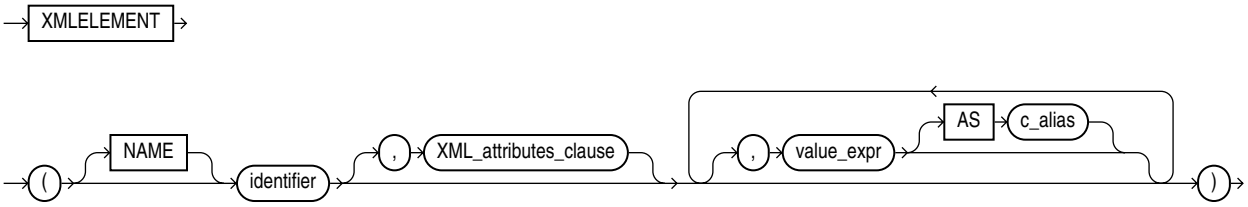
- XMLSEQUENCE SQL Function on page 16-11 . Only the cursor version of this function generates XML.
- XMLCOLATTVAL SQL Function on page 16-22
- XMLCDATA SQL Function on page 16-23
- Generating XML Using SQL Function SYS_XMLGEN on page 16-48. This operates on relational rows, generating XML documents.
- Generating XML Using SQL Function SYS_XMLAGG on page 16-56. This operates on groups of relational rows, aggregating several XML documents into one.

All of the XML-generation SQL functions convert scalars and user-defined datatype instances to their canonical XML format. In this canonical mapping, user-defined datatype attributes are mapped to XML elements.

XMLELEMENT and XMLATTRIBUTES SQL Functions

You use SQL function `XMLElement` to construct XML instances from relational data. It takes as arguments an element name, an optional collection of attributes for the element, and zero or more additional arguments that make up the element content. It returns an `XMLType` instance.

Figure 16–1 XMLELEMENT Syntax



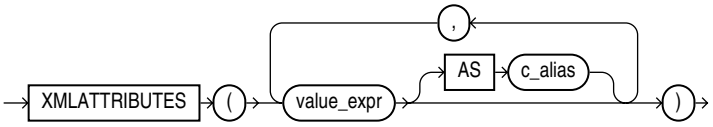
The first argument to function `XMLElement` (*`identifier`*, in Figure 16–1) is an identifier that names the *root* XML element to be created. This root-element identifier need not be a column name or a column reference, and it cannot be an expression to be evaluated. If the *`identifier`* is `NULL`, then *no* root element is generated.

The optional *`XML-attributes-clause`* argument of function `XMLElement` specifies the attributes of the root element to be generated. Figure 16–2 shows the syntax of this argument.

In addition to the optional *`XML-attributes-clause`* argument, function `XMLElement` accepts zero or more *`value_expr`* arguments that make up the *content* of the root element (child elements and text content). If an *`XML-attributes-clause`* argument is also present, these content arguments must follow the *`XML-attributes-clause`* argument. Each of the content-argument expressions is evaluated, and the result is converted to XML format. If a value argument evaluates to `NULL`, then no content is created for that argument.

The optional *`XML-attributes-clause`* argument uses function `XMLAttributes` to specify the *attributes* of the root element. Function `XMLAttributes` can be used only in a call to function `XMLElement`; it cannot be used on its own.

Figure 16–2 XMLAttributes Clause Syntax (XMLATTRIBUTES)



Argument *XML-attributes-clause* itself contains one or more *value_expr* expressions as arguments to function XMLAttributes. These are evaluated to obtain the values for the attributes of the root element. (Do not confuse these *value_expr* arguments to function XMLAttributes with the *value_expr* arguments to function XMLElement, which specify the content of the root element.) The optional AS *c_alias* clause for each *value_expr* specifies that the attribute name is *c_alias* (a string literal).

If an attribute value expression evaluates to NULL, then no corresponding attribute is created. The datatype of an attribute value expression cannot be an object type or a collection.

Escaping Characters in Generated XML Data

As specified by the SQL/XML standard, characters in explicit *identifiers* are *not* escaped in any way – it is up to you to ensure that valid XML names are used. This applies to all SQL/XML functions; in particular, it applies to the root-element identifier of XMLElement (*identifier*, in Figure 16–1) and to attribute identifier aliases named with AS clauses of XMLAttributes (see Figure 16–2).

However, other XML data that is generated is *escaped*, to provide that only valid XML NameChar characters are generated. As part of generating a valid XML element or attribute name from a SQL identifier, each character that is disallowed in an XML name is replaced with an underscore character (_), followed by the hexadecimal Unicode representation of the original character, followed by a second underscore character. For example, the colon character (:) is escaped by replacing it with _003A_, where 003A is the hexadecimal Unicode representation.

Escaping applies to characters in the evaluated *value_expr* arguments to *all* SQL/XML functions, including XMLElement and XMLAttributes. It applies also to the characters of an attribute identifier that is defined implicitly from an XMLAttributes attribute value expression that is *not* followed by an AS clause: the escaped form of the SQL column name is used as the name of the attribute.

Formatting of XML Dates and Timestamps

The XML Schema standard specifies that dates and timestamps in XML data be in standard formats. XML generation functions in Oracle XML DB produce XML dates and timestamps according to this standard.

In releases prior to Oracle Database 10g Release 2, the database settings for date and timestamp formats were used for XML, instead of the XML Schema standard formats. You can reproduce this *previous* behavior by setting the database event 19119, level 0x8, as follows:

```
ALTER SESSION SET EVENTS '19119 TRACE NAME CONTEXT FOREVER, LEVEL 0x8';
```

If you otherwise need to produce a non-standard XML date or timestamp, use SQL function to_char – see Example 16–1.

See Also:

http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#isoformats for the XML Schema specification of XML date and timestamp formats

XMLElement Examples

This section provides examples that use SQL function `XMLElement`.

Example 16–1 XMLELEMENT: Formatting a Date

This example shows how to produce an XML date with a format different from the XML Schema standard format.

```
-- With standard XML date format:
SELECT XMLElement("Date", hire_date)
  FROM hr.employees
  WHERE employee_id = 203;

XMLELEMENT("DATE",HIRE_DATE)
----------------------------
<Date>1994-06-07</Date>

1 row selected.

-- With an alternative date format:
SELECT XMLElement("Date", to_char(hire_date))
  FROM hr.employees
  WHERE employee_id = 203;

XMLELEMENT("DATE",TO_CHAR(HIRE_DATE))
-------------------------------------
<Date>07-JUN-94</Date>

1 row selected.
```

Example 16–2 XMLELEMENT: Generating an Element for Each Employee

This example produces an `Emp` element for each employee, with the employee name as its content:

```
SELECT e.employee_id,
       XMLELEMENT ("Emp", e.first_name ||' '|| e.last_name) AS "RESULT"
   FROM hr.employees e
   WHERE employee_id > 200;
```

This query produces the following typical result:

```
EMPLOYEE_ID RESULT
----------- ---------------------------------
        201 <Emp>Michael Hartstein</Emp>
        202 <Emp>Pat Fay</Emp>
        203 <Emp>Susan Mavris</Emp>
        204 <Emp>Hermann Baer</Emp>
        205 <Emp>Shelley Higgins</Emp>
        206 <Emp>William Gietz</Emp>

6 rows selected.
```

SQL function `XMLElement` can also be nested, to produce XML data with a nested structure.

Example 16–3 XMLELEMENT: Generating Nested XML

To produce an `Emp` element for each employee, with elements that provide the employee name and hire date, do the following:

```
SELECT XMLElement("Emp",
                  XMLElement("name", e.first_name ||' '|| e.last_name),
                  XMLElement("hiredate", e.hire_date)) AS "RESULT"
FROM hr.employees e
WHERE employee_id > 200 ;
```

This query produces the following typical XML result:

```
RESULT
--------------------------------------------------------------------------------
<Emp><name>Michael Hartstein</name><hiredate>1996-02-17</hiredate></Emp>
<Emp><name>Pat Fay</name><hiredate>1997-08-17</hiredate></Emp>
<Emp><name>Susan Mavris</name><hiredate>1994-06-07</hiredate></Emp>
<Emp><name>Hermann Baer</name><hiredate>1994-06-07</hiredate></Emp>
<Emp><name>Shelley Higgins</name><hiredate>1994-06-07</hiredate></Emp>
<Emp><name>William Gietz</name><hiredate>1994-06-07</hiredate></Emp>

6 rows selected.
```

Example 16–4 XMLELEMENT: Generating Employee Elements with ID and Name Attributes

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.first_name ||' ' || e.last_name AS "name"))
  AS "RESULT"
  FROM hr.employees e
  WHERE employee_id > 200;
```

This query produces the following typical XML result fragment:

```
RESULT
------------------------------------------------
<Emp ID="201" name="Michael Hartstein"></Emp>
<Emp ID="202" name="Pat Fay"></Emp>
<Emp ID="203" name="Susan Mavris"></Emp>
<Emp ID="204" name="Hermann Baer"></Emp>
<Emp ID="205" name="Shelley Higgins"></Emp>
<Emp ID="206" name="William Gietz"></Emp>

6 rows selected.
```

As mentioned in "Escaping Characters in Generated XML Data" on page 16-5, characters in the root-element name and the names of any attributes defined by `AS` clauses are *not* escaped. Characters in an identifier name are escaped only if the name is created from an evaluated expression (such as a column reference). The following query shows that the root-element name and the attribute name are *not* escaped. Invalid XML is produced because greater-than sign (>) and a comma (,) are not allowed in XML element and attribute names.

```
SELECT XMLElement("Emp->Special",
                  XMLAttributes(e.last_name || ', ' || e.first_name
                                AS "Last,First"))
   AS "RESULT"
```

```
   FROM hr.employees e
   WHERE employee_id = 201;
```

This query produces the following result, which is not well-formed XML:

```
RESULT
--------------------------------------------------------------------
<Emp->Special Last,First="Hartstein, Michael"></Emp->Special>

1 row selected.
```

A full description of character escaping is included in the SQL/XML standard.

Example 16–5 XMLELEMENT: Using Namespaces to Create a Schema-Based XML Document

This 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:

```
SELECT XMLElement("Employee",
                  XMLAttributes('http://www.w3.org/2001/XMLSchema' AS
                                  "xmlns:xsi",
                                'http://www.oracle.com/Employee.xsd' AS
                                  "xsi:nonamespaceSchemaLocation"),
                  XMLForest(employee_id, last_name, salary)) AS "RESULT"
   FROM hr.employees
   WHERE department_id = 10;
```

This creates the following XML document that conforms to XML schema `Employee.xsd` (the actual result is *not* pretty-printed).

```
RESULT
--------------------------------------------------------------------------------
<Employee xmlns:xsi="http://www.w3.org/2001/XMLSchema"
          xsi:nonamespaceSchemaLocation="http://www.oracle.com/Employee.xsd">
   <EMPLOYEE_ID>200</EMPLOYEE_ID>
   <LAST_NAME>Whalen</LAST_NAME>
   <SALARY>4400</SALARY>
</Employee>

1 row selected.
```

Example 16–6 XMLELEMENT: Generating an Element from a User-Defined Datatype Instance

Example 16–10 shows an XML document with employee information. You can generate a hierarchical XML document with the employee and department information as follows:

```
CREATE OR REPLACE TYPE emp_t AS OBJECT ("@EMPNO" NUMBER(4),
                                        ENAME VARCHAR2(10));
/
Type created.

CREATE OR REPLACE TYPE emplist_t AS TABLE OF emp_t;
/
Type created.

CREATE OR REPLACE TYPE dept_t AS OBJECT ("@DEPTNO" NUMBER(2),
```

```
                                        DNAME VARCHAR2(14),
                                        EMP_LIST emplist_t);
/
Type created.

SELECT XMLElement("Department",
                  dept_t(department_id,
                         department_name,
                         CAST(MULTISET(SELECT employee_id, last_name
                                       FROM hr.employees e
                                       WHERE e.department_id = d.department_id)
                              AS emplist_t)))
  AS deptxml
  FROM hr.departments d
  WHERE d.department_id = 10;
```

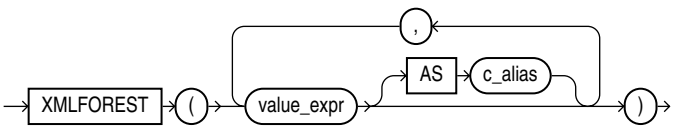
This produces an XML document which contains the `Department` element and the canonical mapping of type `dept_t`.

```
DEPTXML
-------------
<Department>
  <DEPT_T DEPTNO="10">
    <DNAME>ACCOUNTING</DNAME>
    <EMPLIST>
      <EMP_T EMPNO="7782">
        <ENAME>CLARK</ENAME>
      </EMP_T>
      <EMP_T EMPNO="7839">
        <ENAME>KING</ENAME>
      </EMP_T>
      <EMP_T EMPNO="7934">
        <ENAME>MILLER</ENAME>
      </EMP_T>
    </EMPLIST>
  </DEPT_T>
</Department>

1 row selected.
```

XMLFOREST SQL Function

SQL function `XMLForest` produces a forest of XML elements from its arguments, which are expressions to be evaluated, with optional aliases. Figure 16–3 describes the `XMLForest` syntax.

Figure 16–3 XMLFOREST Syntax



Each of the value expressions (*value_expr* in Figure 16–3) is converted to XML format, and, optionally, identifier *c_alias* is used as the attribute identifier.

For an object type or collection, the `AS` clause is required. For other types, the `AS` clause is optional. For a given expression, if the `AS` clause is omitted, then characters in the evaluated value expression are *escaped* to form the name of the enclosing tag of the

element. The escaping is as defined in "Escaping Characters in Generated XML Data" on page 16-5. If the value expression evaluates to NULL, then no element is created for that expression.

Example 16–7 XMLFOREST: Generating Elements with Attribute and Child Elements

This example generates an Emp element for each employee, with a name attribute and elements with the employee hire date and department as the content.

```
SELECT XMLElement("Emp",
                  XMLAttributes(e.first_name ||' '|| e.last_name AS "name"),
                  XMLForest(e.hire_date, e.department AS "department"))
AS "RESULT"
FROM employees e WHERE e.department_id = 20;
```

(The WHERE clause is used here to keep the example brief.) This query produces the following XML result:

```
RESULT
-------------------------------------
<Emp name="Michael Hartstein">
  <HIRE_DATE>1996-02-17</HIRE_DATE>
  <department>20</department>
</Emp>
<Emp name="Pat Fay">
  <HIRE_DATE>1997-08-17</HIRE_DATE>
  <department>20</department>
</Emp>

2 rows selected.
```

> **See Also:** Example 16–22, "XMLCOLATTVAL: Generating Elements with Attribute and Child Elements"

Example 16–8 XMLFOREST: Generating an Element from a User-Defined Datatype Instance

You can also use SQL function XMLForest to generate hierarchical XML from user-defined datatype instances.

```
SELECT XMLForest(
  dept_t(department_id,
         department_name,
         CAST (MULTISET (SELECT employee_id, last_name
                         FROM hr.employees e
                         WHERE e.department_id = d.department_id)
               AS emplist_t))
         AS "Department")
  AS deptxml
  FROM hr.departments d
  WHERE department_id=10;
```

This produces an XML document with element Department containing attribute DEPTNO and child element DNAME.

```
DEPTXML
--------------------------------
<Department DEPTNO="10">
  <DNAME>Administration</DNAME>
    <EMP_LIST>
      <EMP_T EMPNO="200">
        <ENAME>Whalen</ENAME>
```

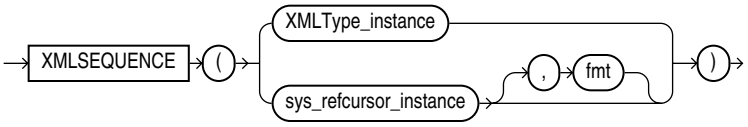
```
      </EMP_T>
    </EMP_LIST>
</Department>

1 row selected.
```

You may want to compare this example with Example 16–6 and Example 16–27.

XMLSEQUENCE SQL Function

SQL function `XMLSequence` returns an `XMLSequenceType` value (a varray of `XMLType` instances). Because it returns a collection, this function can be used in the `FROM` clause of SQL queries. See Figure 16–4.

Figure 16–4 XMLSEQUENCE Syntax



Example 16–9 XMLSEQUENCE Returns Only Top-Level Element Nodes

Function `XMLSequence` returns only top-level element nodes. That is, it will not shred attributes or text nodes.

```
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.
```

Function `XMLSequence` has two forms:

- The first form takes as input 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` instance and an optional instance of the `XMLFormat` object, and returns a varray of `XMLType` instances corresponding to each row of the cursor. This form can be used to construct `XMLType` instances from arbitrary SQL queries. This use of `XMLFormat` does *not* support XML schemas.

Function `XMLSequence` is essential for effective SQL queries involving `XMLType` instances.

Example 16–10 XMLSEQUENCE: Generating One XML Document from Another

Consider the following XMLType table containing an XML document with employee information:

```
CREATE TABLE emp_xml_tab OF XMLType;
```

```
Table created.

INSERT INTO emp_xml_tab VALUES( XMLType('<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>'));

1 row created.

COMMIT;
```

To create a new XML document containing only employees who earn $50,000 or more, you can use the following query:

```
SELECT sys_XMLAgg(value(em), XMLFormat('EMPLOYEES'))
  FROM emp_xml_tab doc, table(XMLSequence(extract(value(doc),
                                          '/EMPLOYEES/EMP'))) em
  WHERE extractValue(value(em), '/EMP/SALARY') >= 50000;
```

These are the steps involved in this query:

1. Function `extract` returns a fragment of `EMP` elements.
2. Function `XMLSequence` gathers a collection of these top-level elements into `XMLType` instances and returns that.
3. Function `table` makes a table value from the collection. The table value is then used in the query `FROM` clause.

The query returns the following XML document:

```
SYS_XMLAGG(VALUE(EM),XMLFORMAT('EMPLOYEES'))
--------------------------------------------
<?xml version="1.0"?>
<EMPLOYEES>
  <EMP>
    <EMPNO>112</EMPNO>
    <EMPNAME>Joe</EMPNAME>
    <SALARY>50000</SALARY>
  </EMP>
  <EMP>
    <EMPNO>217</EMPNO>
    <EMPNAME>Jane</EMPNAME>
    <SALARY>60000</SALARY>
  </EMP>
</EMPLOYEES>

1 row selected.
```

Example 16–11 XMLSEQUENCE: Generate a Document for Each Row of a Cursor

In this example, SQL function `XMLSequence` is used to create an XML document for each row of a cursor expression, and it returns an `XMLSequenceType` value (a varray of `XMLType` instances).

```
SELECT value(em).getClobVal() AS "XMLTYPE"
  FROM table(XMLSequence(Cursor(SELECT *
                                FROM hr.employees
                                WHERE employee_id = 104))) em;
```

This query returns the following XML:

```
XMLTYPE
--------------------------------------------------
 <ROW>
  <EMPLOYEE_ID>104</EMPLOYEE_ID>
  <FIRST_NAME>Bruce</FIRST_NAME>
  <LAST_NAME>Ernst</LAST_NAME>
  <EMAIL>BERNST</EMAIL>
  <PHONE_NUMBER>590.423.4568</PHONE_NUMBER>
  <HIRE_DATE>21-MAY-91</HIRE_DATE>
  <JOB_ID>IT_PROG</JOB_ID>
  <SALARY>6000</SALARY>
  <MANAGER_ID>103</MANAGER_ID>
  <DEPARTMENT_ID>60</DEPARTMENT_ID>
 </ROW>

1 row selected.
```

The tag used for each row can be changed using the `XMLFormat` object.

Example 16–12 XMLSEQUENCE: Unnesting Collections in XML Documents into SQL Rows

Because SQL function `XMLSequence` is a table function, it can be used to unnest the elements inside an XML document. For example, consider the following `XMLType` table `dept_xml_tab` containing XML documents:

```
CREATE TABLE dept_xml_tab OF XMLType;

Table created.

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>'));

1 row created.

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>'));

1 row created.

COMMIT;
```

You can use SQL function `XMLSequence` to unnest the `Employee` list items as top-level SQL rows:

```
SELECT extractValue(OBJECT_VALUE, '/Department/@deptno') AS deptno,
       extractValue(value(em), '/Employee/@empno') AS empno,
       extractValue(value(em), '/Employee/Ename') AS ename
  FROM dept_xml_tab,
       table(XMLSequence(extract(OBJECT_VALUE,
                                 '/Department/EmployeeList/Employee'))) em;
```

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`, function `table` is applied. Here, function `extract` creates a new `XMLType` instance that contains a fragment of all employee elements. This is fed to SQL function `XMLSequence`, which creates a collection of all employees.

Function `TABLE` 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.

Function `extractValue` extracts out the scalar values for the department number, employee number, and name.

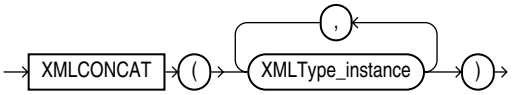
See Also: Chapter 4, "XMLType Operations"

XMLCONCAT SQL Function

SQL function `XMLConcat` concatenates all of its arguments to create an XML fragment. Figure 16–5 shows the `XMLConcat` syntax. Function `XMLConcat` has two forms:

- The first form takes an `XMLSequenceType` value (a varray of `XMLType` instances) and returns a single `XMLType` instance that is the concatenation of all of the elements of the varray. This form is useful to collapse lists of `XMLTypes` into a single instance.
- The second form takes an arbitrary number of `XMLType` instances and concatenates them together. If one of the values is `NULL`, then it is ignored in the result. If all the values are `NULL`, then the result is `NULL`. This form is used to concatenate arbitrary number of `XMLType` instances in the same row. Function `XMLAgg` can be used to concatenate `XMLType` instances across rows.

Figure 16–5 XMLCONCAT Syntax



Example 16–13 XMLCONCAT: Concatenating XMLType Instances from a Sequence

This example uses function `XMLConcat` to return a concatenation of `XMLType` instances from an `XMLSequenceType` value (a varray of `XMLType` instances).

```
SELECT XMLConcat(XMLSequenceType(
                  XMLType('<PartNo>1236</PartNo>'),
                  XMLType('<PartName>Widget</PartName>'),
                  XMLType('<PartPrice>29.99</PartPrice>'))).getClobVal()
  AS "RESULT"
  FROM DUAL;
```

This query returns a single XML fragment (the actual output is not pretty-printed):

```
RESULT
---------------
<PartNo>1236</PartNo>
<PartName>Widget</PartName>
<PartPrice>29.99</PartPrice>

1 row selected.
```

Example 16–14 XMLCONCAT: Concatenating XML Elements

The following example creates an XML element for the first and the last names and then concatenates the result:

```
SELECT XMLConcat(XMLElement("first", e.first_name),
                 XMLElement("last", e.last_name))
  AS "RESULT"
  FROM employees e;
```

This query produces the following XML fragment:

```
RESULT
-----------------------------------------
<first>Den</first><last>Raphaely</last>
<first>Alexander</first><last>Khoo</last>
<first>Shelli</first><last>Baida</last>
<first>Sigal</first><last>Tobias</last>
<first>Guy</first><last>Himuro</last>
<first>Karen</first><last>Colmenares</last>

6 rows selected.
```

XMLAGG SQL Function

SQL function `XMLAgg` is an aggregate function that produces a forest of XML elements from a collection of XML elements.

Figure 16–6 XMLAGG Syntax

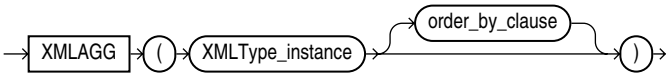


Figure 16–6 describes the XMLAgg() syntax, where the order_by_clause is the following:

```
ORDER BY [list of: expr [ASC|DESC] [NULLS {FIRST|LAST}]]
```

Numeric literals are *not* interpreted as column positions. For example, ORDER BY 1 does not mean order by the first column. Instead, numeric literals are interpreted as any other literals.

As with SQL function XMLConcat, any arguments that are NULL are dropped from the result. Function XMLAgg is similar to function sys_XMLAgg, except that it returns a forest of nodes and does not take the XMLFormat parameter. Function XMLAgg 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.

Function XMLAgg produces one aggregated XML result for each group. If there is no group by specified in the query, then it returns a single aggregated XML result for all the rows of the query.

Example 16–15 XMLAGG: Generating Department Elements with a List of Employee Elements

This 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. (The actual result is *not* pretty-printed.)

```
SELECT XMLElement("Department", XMLAgg(XMLElement("Employee",
                                                  e.job_id||' '||e.last_name)
                                       ORDER BY e.last_name))
  AS "Dept_list"
  FROM hr.employees e
  WHERE e.department_id = 30 OR e.department_id = 40;

Dept_list
------------------
<Department>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>HR_REP Mavris</Employee>
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Tobias</Employee>
</Department>

1 row selected.
```

The result is a *single* row, because XMLAgg aggregates the rows. You can use the GROUP BY clause to group the returned set of rows into multiple groups. (The actual result of the following query is *not* pretty-printed.)

```
SELECT XMLElement("Department", XMLAttributes(department_id AS "deptno"),
                  XMLAgg(XMLElement("Employee", e.job_id||' '||e.last_name)))
   AS "Dept_list"
   FROM hr.employees e
   GROUP BY e.department_id;

Dept_list
------------------
<Department deptno="30">
```

```
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Himuro</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Tobias</Employee>
</Department>

<Department deptno="40">
  <Employee>HR_REP Mavris</Employee>
</Department>

2 rows selected.
```

You can order the employees within each department by using the `ORDER BY` clause inside the `XMLAgg` expression.

> **Note:** Within the *order_by_clause*, Oracle Database does not interpret number literals as column positions, as it does in other uses of this clause, but simply as number literals.

Example 16–16 XMLAGG: Generating Nested Elements

Function `XMLAgg` can be used to reflect the hierarchical nature of some relationships that exist in tables. This example generates a department element for department 30. Within this element is a child element for each employee of the department. Within each employee element is a `dependent` element for each dependent of that employee.

First, this query shows the employees of department 30.

```
SELECT last_name, employee_id FROM employees WHERE department_id = 30;

LAST_NAME                 EMPLOYEE_ID
------------------------- -----------
Raphaely                          114
Khoo                              115
Baida                             116
Tobias                            117
Himuro                            118
Colmenares                        119

6 rows selected.
```

A `dependents` table is created, to hold the dependents of each employee.

```
CREATE TABLE hr.dependents (id NUMBER(4) PRIMARY KEY,
                            employee_id NUMBER(4),
                            name VARCHAR2(10));
Table created.
INSERT INTO dependents VALUES (1, 114, 'MARK');
1 row created.
INSERT INTO dependents VALUES (2, 114, 'JACK');
1 row created.
INSERT INTO dependents VALUES (3, 115, 'JANE');
1 row created.
INSERT INTO dependents VALUES (4, 116, 'HELEN');
1 row created.
INSERT INTO dependents VALUES (5, 116, 'FRANK');
1 row created.
COMMIT;
```

```
Commit complete.
```

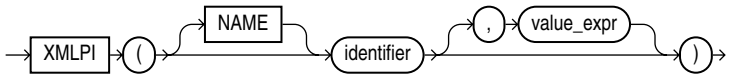
This query generates the XML data for department that contains the information on dependents (the actual output is *not* pretty-printed):

```
SELECT
  XMLElement(
    "Department",
    XMLAttributes(d.department_name AS "name"),
    (SELECT
       XMLAgg(XMLElement("emp",
                         XMLAttributes(e.last_name AS name),
                         (SELECT XMLAgg(XMLElement("dependent",
                                        XMLAttributes(de.name AS "name")))
                            FROM dependents de
                            WHERE de.employee_id = e.employee_id)))
       FROM employees e
       WHERE e.department_id = d.department_id)) AS "dept_list"
  FROM departments d
  WHERE department_id = 30;

dept_list
--------------------------------------------------------------------------------
<Department name="Purchasing">
  <emp NAME="Raphaely">
    <dependent name="MARK"></dependent>
    <dependent name="JACK"></dependent>
  </emp><emp NAME="Khoo">
    <dependent name="JANE"></dependent>
  </emp>
  <emp NAME="Baida">
    <dependent name="HELEN"></dependent>
    <dependent name="FRANK"></dependent>
  </emp><emp NAME="Tobias"></emp>
  <emp NAME="Himuro"></emp>
  <emp NAME="Colmenares"></emp>
</Department>

1 row selected.
```

XMLPI SQL Function

You use SQL function `XMLPI` to generate XML processing instructions (PIs). Figure 16–7 shows the syntax:

Figure 16–7 XMLPI Syntax



Argument *value_expr* is evaluated, and the string result is appended to the optional identifier (*identifier*), separated by a space. This concatenation is then enclosed betweeen `"<?"` and `"?>"` to create the processing instruction. That is, if *string-result* is the result of evaluating *value_expr*, then the generated processing instruction is <?*identifier string-result?*>. If *string-result* is the empty string, `''`, then the function returns <?*identifier?*>.

As an alternative to using keyword NAME followed by a *literal* string *identifier*, you can use keyword EVALNAME followed by an expression that *evaluates* to a string to be used as the identifier.

An error is raised if the constructed XML is not a legal XML processing instruction. In particular:

- *identifier* must *not* be the word "xml" (uppercase, lowercase, or mixed case).
- *string-result* must *not* contain the character sequence "?>".

Function XMLPI returns an instance of XMLType. If *string-result* is NULL, then it returns NULL.

Example 16–17 Using XMLPI

```
SELECT XMLPI(NAME "OrderAnalysisComp", 'imported, reconfigured, disassembled')
  AS pi FROM DUAL;
```

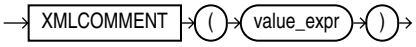
This results in the following output :

```
PI
------------------------------------------------------------
<?OrderAnalysisComp imported, reconfigured, disassembled?>

1 row selected.
```

XMLCOMMENT SQL Function

You use SQL function XMLComment to generate XML comments. Figure 16–8 shows the syntax:

Figure 16–8 XMLComment Syntax



Argument *value_expr* is evaluated to a string, and the result is used as the body of the generated XML comment; that is, the result is <!--*string-result*-->, where *string-result* is the string result of evaluating *value_expr*. If *string-result* is the empty string, then the comment is empty: <!---->.

An error is raised if the constructed XML is not a legal XML comment. In particular, *string-result* must *not* contain two consecutive hyphens (-): "--".

Function XMLComment returns an instance of XMLType. If *string-result* is NULL, then the function returns NULL.

Example 16–18 Using XMLCOMMENT

```
SELECT XMLComment('This is a comment') AS cmnt FROM DUAL;
```

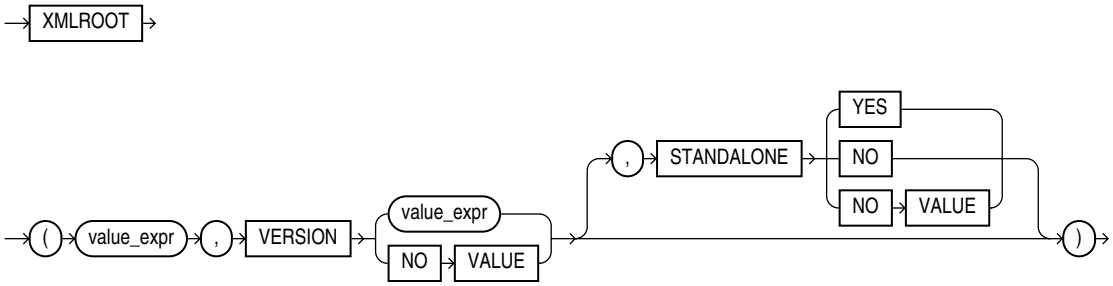
This query results in the following output:

```
CMNT
--------------------------
<!--This is a comment-->
```

XMLROOT SQL Function

You use SQL function `XMLRoot` to add a `VERSION` property, and optionally a `STANDALONE` property, to the root information item of an XML value. Typically, this is done to ensure data-model compliance. Figure 16–9 shows the syntax of `XMLRoot`:

Figure 16–9 XMLRoot Syntax



First argument *`xml-expression`* is evaluated, and the indicated properties (`VERSION`, `STANDALONE`) and their values are added to a new prolog for the resulting `XMLType` instance. If the evaluated *`xml-expression`* already contains a prolog, then an error is raised.

Second argument *`string-valued-expression`* (which follows keyword `VERSION`) is evaluated, and the resulting string is used as the value of the prolog `version` property. The value of the prolog `standalone` property (lowercase) is taken from the optional third argument `STANDALONE YES` or `NO` value. If `NOVALUE` is used for `VERSION`, then `"version=1.0"` is used in the resulting prolog. If `NOVALUE` is used for `STANDALONE`, then the `standalone` property is omitted from the resulting prolog.

Function `XMLRoot` returns an instance of `XMLType`. If first argument *`xml-expression`* evaluates to `NULL`, then the function returns `NULL`.

Example 16–19 Using XMLRoot

```
SELECT XMLRoot(XMLType('<poid>143598</poid>'), VERSION '1.0', STANDALONE YES)
  AS xmlroot FROM DUAL;
```

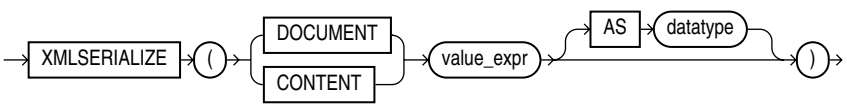
This results in the following output :

```
XMLROOT
--------------------------------------
<?xml version="1.0" standalone="yes"?>
<poid>143598</poid>

1 row selected.
```

XMLSERIALIZE SQL Function

You use SQL function `XMLSerialize` to obtain a string or a LOB representation of XML data. Figure 16–10 shows the syntax:

Figure 16–10 XMLSerialize Syntax



Argument *value_expr* is evaluated, and the resulting XMLType instance is serialized to produce the content of the created string or LOB. If present[1], the specified *datatype* must be one of the following (the default datatype is CLOB):

- VARCHAR2
- VARCHAR
- CLOB

If you specify DOCUMENT, then the result of evaluating *value_expr* must be a well-formed document; in particular, it must have a single root. If the result is not a well-formed document, then an error is raised. If you specify CONTENT, however, then the result of *value_expr* is *not* checked for well-formedness.

If the underlying CLOB value or string has encoding information, then an appropriate encoding="..." declaration is added to the prolog.

If *value_expr* evaluates to NULL or to the empty string (''), then function XMLSerialize returns NULL.

Example 16–20 Using XMLSERIALIZE

```
SELECT XMLSerialize(DOCUMENT XMLType('<poid>143598</poid>') AS CLOB)
  AS xmlserialize_doc FROM DUAL;
```

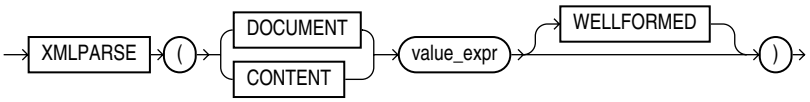
This results in the following output:

```
XMLSERIALIZE_DOC
-------------------
<poid>143598</poid>
```

XMLPARSE SQL Function

You use SQL function XMLParse to parse a string containing XML data and generate a corresponding value of XMLType. Figure 16–11 shows the syntax:

Figure 16–11 XMLParse Syntax



Argument *value_expr* is evaluated to produce the string that is parsed. If you specify DOCUMENT, then *value_expr* must correspond to a *singly rooted*, well-formed XML document. If you specify CONTENT, then *value_expr* need only correspond to a well-formed XML fragment; that is, it need not be singly rooted.

Keyword WELLFORMED is an Oracle XML DB extension to the SQL/XML standard. When you specify WELLFORMED, you are informing the parser that argument *value_expr* is well-formed, so Oracle XML DB does *not* check to ensure that it is in fact well-formed.

Function XMLParse returns an instance of XMLType. If *value_expr* evaluates to NULL, then the function returns NULL.

[1] The SQL/XML standard requires argument *data-type* to be present, but it is *optional* in the Oracle XML DB implementation of the standard, for ease of use.

Example 16–21 Using XMLPARSE

```
SELECT XMLParse(CONTENT
                '124 <purchaseOrder poNo="12435">
                       <customerName> Acme Enterprises</customerName>
                       <itemNo>32987457</itemNo>
                     </purchaseOrder>'
                WELLFORMED)
  AS po FROM DUAL d;
```

This results in the following output :

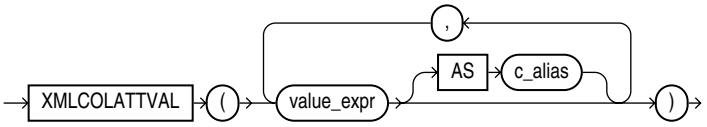
```
PO
--------------------------------------------
124 <purchaseOrder poNo="12435">
<customerName>Acme Enterprises</customerName>
<itemNo>32987457</itemNo>
</purchaseOrder>
```

> **See Also:** http://www.w3.org/TR/REC-xml/, *Extensible Markup Language (XML) 1.0*, for the definition of well-formed XML documents and fragments

XMLCOLATTVAL SQL Function

SQL function `XMLColAttVal` generates a forest of XML `column` elements containing the values of the arguments passed in. This function is an Oracle Database extension to the SQL/XML ANSI-ISO standard functions. Figure 16–12 shows the `XMLColAttVal` syntax.

Figure 16–12 XMLCOLATTVAL Syntax



The arguments are used as the values of the `name` attribute of the `column` element. The *`c_alias`* values are used as the attribute identifiers.

As an alternative to using keyword `AS` followed by a *literal* string *`c_alias`*, you can use `AS EVALNAME` followed by an expression that *evaluates* to a string to be used as the attribute identifier.

Because argument values *`value_expr`* are used only as attribute *values*, they need *not* be escaped in any way. This is in contrast to function `XMLForest`. It means that you can use `XMLColAttVal` to transport SQL columns and values without escaping.

Example 16–22 XMLCOLATTVAL: Generating Elements with Attribute and Child Elements

This example generates an `Emp` element for each employee, with a `name` attribute and elements with the employee hire date and department as the content.

```
SELECT XMLElement("Emp",
                  XMLAttributes(e.first_name ||' '||e.last_name AS "fullname" ),
                  XMLColAttVal(e.hire_date, e.department_id AS "department"))
  AS "RESULT"
  FROM hr.employees e
```

```
  WHERE e.department_id = 30;
```

This query produces the following XML result (the actual result is *not* pretty-printed):

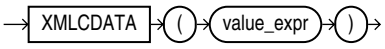
```
RESULT
------------------------------------------------------------
<Emp fullname="Den Raphaely">
  <column name = "HIRE_DATE">1994-12-07</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Alexander Khoo">
  <column name = "HIRE_DATE">1995-05-18</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Shelli Baida">
  <column name = "HIRE_DATE">1997-12-24</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Sigal Tobias">
  <column name = "HIRE_DATE">1997-07-24</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Guy Himuro">
  <column name = "HIRE_DATE">1998-11-15</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Karen Colmenares">
  <column name = "HIRE_DATE">1999-08-10</column>
  <column name = "department">30</column>
</Emp>

6 rows selected.
```

See Also: Example 16–7, "XMLFOREST: Generating Elements with Attribute and Child Elements"

XMLCDATA SQL Function

You use SQL function `XMLCDATA` to generate an XML `CDATA` section. Figure 16–13 shows the syntax:

Figure 16–13 XMLCDATA Syntax



Argument *value_expr* is evaluated to a string, and the result is used as the body of the generated XML `CDATA` section, `<![CDATA[`*string-result*`]]>`, where *string-result* is the result of evaluating *value_expr*. If *string-result* is the empty string, then the `CDATA` section is empty: `<![CDATA[]]>`.

An error is raised if the constructed XML is not a legal XML `CDATA` section. In particular, *string-result* must *not* contain two consecutive right brackets (`]`): "`]]`".

Function `XMLCDATA` returns an instance of `XMLType`. If *string-result* is `NULL`, then the function returns `NULL`.

Example 16–23 Using XMLCDATA

```
SELECT XMLElement("PurchaseOrder",
                  XMLElement("Address",
```

```
                              XMLCDATA('100 Pennsylvania Ave.'),
                              XMLElement("City", 'Washington, D.C.')))
  AS RESULT FROM DUAL;
```

This results in the following output (the actual output is not pretty-printed):

```
RESULT
--------------------------
<PurchaseOrder>
  <Address>
    <![CDATA[100 Pennsylvania Ave.]]>
    <City>Washington, D.C.</City>
  </Address>
</PurchaseOrder>
```

Generating XML Using DBMS_XMLGEN

PL/SQL package `DBMS_XMLGEN` creates XML documents from SQL query results. It retrieves an XML document as a `CLOB` or `XMLType` value.

It provides a `fetch` interface, whereby you can specify the maximum number of rows to retrieve and the number of rows to skip. For example, the first fetch could retrieve a maximum of ten rows, skipping the first four. This is especially useful for pagination requirements in Web applications.

Package `DBMS_XMLGEN` also provides options for changing tag names for `ROW`, `ROWSET`, and so on. The parameters of the package can restrict the number of rows retrieved and the enclosing tag names.

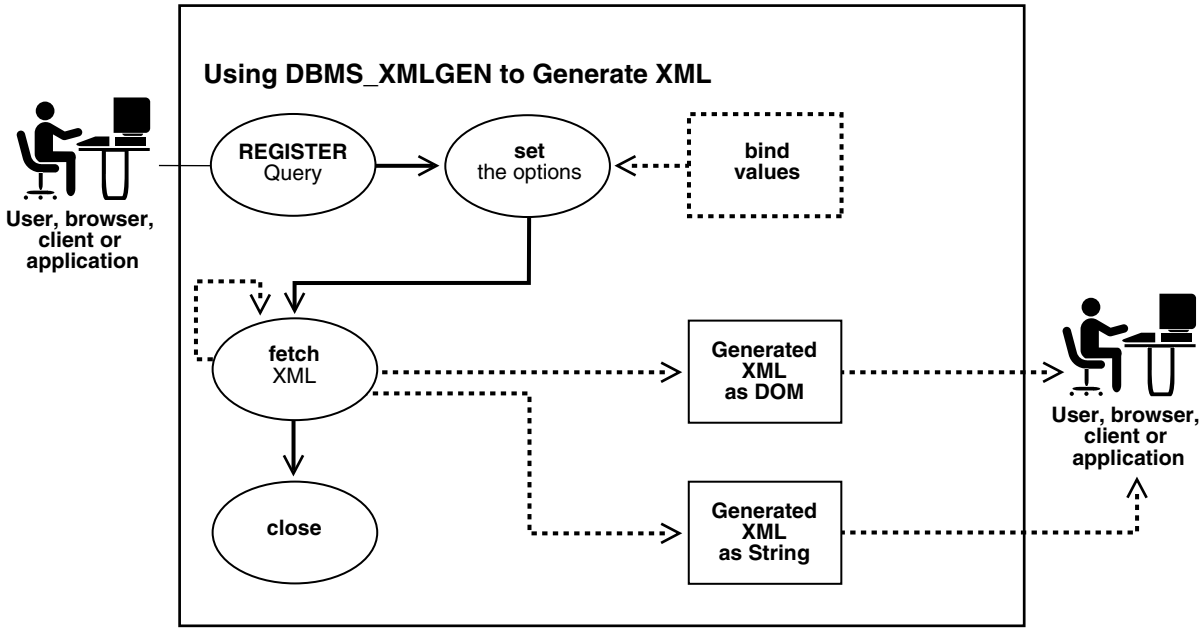
> **See Also:** "Generating XML with XSU's OracleXMLQuery", in *Oracle XML Developer's Kit Programmer's Guide* (compare the functionality of `OracleXMLQuery` with `DBMS_XMLGEN`)

Using DBMS_XMLGEN

Figure 16–14 illustrates how to use package `DBMS_XMLGEN`. The steps are as follows:

1. Get the context from the package by supplying a SQL query and calling 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 `getXML()` or `getXMLType()`. By setting the maximum number of rows to be retrieved for each fetch using `setMaxRows()`, you can call either of these functions repeatedly, retrieving up to the maximum number of rows for each call. These functions return XML data (as a `CLOB` value and as an instance of `XMLType`, respectively), unless there are no rows retrieved; in that case, these functions return `NULL`. To determine how many rows were retrieved, use 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.

Figure 16–14 Using DBMS_XMLGEN



In conjunction with a SQL query, method `DBMS_XMLGEN.getXML()` typically returns a result like the following as a `CLOB` value:

```
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>100</EMPLOYEE_ID>
  <FIRST_NAME>Steven</FIRST_NAME>
  <LAST_NAME>King</LAST_NAME>
  <EMAIL>SKING</EMAIL>
  <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
  <HIRE_DATE>17-JUN-87</HIRE_DATE>
  <JOB_ID>AD_PRES</JOB_ID>
  <SALARY>24000</SALARY>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>101</EMPLOYEE_ID>
  <FIRST_NAME>Neena</FIRST_NAME>
  <LAST_NAME>Kochhar</LAST_NAME>
  <EMAIL>NKOCHHAR</EMAIL>
  <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
  <HIRE_DATE>21-SEP-89</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
</ROWSET>
```

The default mapping between relational data and XML data is as follows:

- Each row returned by the SQL query maps to an XML element with the default element name `ROW`.

- Each column returned by the SQL query maps to a child element of the ROW element.
- The entire result is wrapped in a ROWSET element.
- Binary data is transformed to its hexadecimal representation.

Element names ROW and ROWSET can be replaced with names you choose, using DBMS_XMLGEN procedures setRowTagName() and setRowSetTagName(), respectively.

The CLOB value returned by getXML() has the same encoding as the database character set. If the database character set is SHIFTJIS, then the XML document returned is also SHIFTJIS.

Functions and Procedures of Package DBMS_XMLGEN

Table 16–1 describes the functions and procedures of package DBMS_XMLGEN.

Table 16–1 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 function getXML to specify whether to generate a DTD or XML schema or neither (NONE). Only the NONE specification is supported in 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 *Parameter:* queryString (IN)- the query string, the result of which must be converted to XML *Returns:* Context handle. Call this function first to obtain a handle that you can use in the getXML() and other functions to get the XML back from the result.
FUNCTION newContext(queryString IN SYS_REFCURSOR) RETURN ctxHandle;	Creates a new context handle from a PL/SQL cursor variable. The context handle can be used for the rest of the functions.
FUNCTION newContextFromHierarchy(queryString IN VARCHAR2) RETURN ctxHandle; Introduced in Oracle Database 10*g* Release 1 (10.0.1).	*Parameter:* queryString (IN) - the query string, the result of which must be converted to XML. The query is a hierarchical query typically formed using a CONNECT BY clause, and the result must have the same property as the result set generated by a CONNECT BY query. The result set must have only two columns, the level number and an XML value. The level number is used to determine the hierarchical position of the XML value within the result XML document. *Returns:* Context handle. Call this function first to obtain a handle that you can use in the getXML() and other functions to get a hierarchical XML with recursive elements back from the result.

Table 16–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
`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. A `NULL` value for `rowTag` indicates that you do not want the `ROW` element to be present. Call this procedure to set the name of the `ROW` element, if you do not want the default `ROW` name to show up. You can also set `rowTag` to `NULL` to suppress the `ROW` element itself. However, since getXML returns complete XML documents, not XML fragments, there must be a (single) root element. Therefore, an error is raised if both the `rowTag` value and the `rowSetTag` value (see `setRowSetTag`, next) are `NULL` and there is more than one column or row in the output.
`setRowSetTag()`	Sets the name of the document root element. The default name is `ROWSET`
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 root element to be used in the output. A `NULL` value for `rowSetTag` indicates that you do *not* want the `ROWSET` element to be present. Call this procedure to set the name of the document root element, if you do not want the default name `ROWSET` to be used. You can set `rowSetTag` to `NULL` to suppress printing of the document root element. However, since function`getXML` returns complete XML documents, not XML fragments, there must be a (single) root element. Therefore, an error is raised if both the `rowTag` value and the `rowSetTag` value (see `setRowTag`, previous) are `NULL` and there is more than one column or row in the output, or if the `rowSetTag` value is `NULL` and there is more than one row in the output.
`getXML()`	Gets the XML document by fetching the maximum number of rows specified. It appends the XML document to the `CLOB` passed in.

Table 16–1 (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 XML data using function`getXML` . This count does not include the number of rows *skipped* before generating XML data.

Table 16–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
FUNCTION `getNumRowsProcessed(ctx IN ctxHandle) RETURN number;`	*Parameter:* `queryString(IN)` - the query string, the result of which needs to be converted to XML *Returns:* The number of SQL rows that were processed in the last call to `getXML`. You can call this to find out if the end of the result set has been reached. This does not include the number of rows *skipped* before generating XML data. Use this function to determine the terminating condition if you are calling `getXML` in a loop. Note that `getXML` would always generate an XML document even if there are no rows present.
`setMaxRows()`	Sets the maximum number of rows to fetch from the SQL query result for every invocation of the `getXML` call. It is an error to call this function on a context handle created by `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 function `newContextFormHierarchy()`.
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 escape conversions.

Table 16–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
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);`	*Parameter:* `ctx(IN)` - the context handle. If you have a collection of `NUMBER`, say, the default tag name for the collection elements is `NUMBER`. You can override this action and generate the collection column name with the `_ITEM` tag appended to it, by calling this procedure.
`restartQuery()`	Restarts the query and generate the XML from the first row again.
PROCEDURE `restartQuery(ctx IN ctxHandle);`	*Parameter:* `ctx(IN)` - the context handle corresponding to the current query. You can call this to start executing the query again, without having to create a new context.
`closeContext()`	Closes a given context and releases all resources associated with that context, including the SQL cursor and bind and define buffers, and so on.
PROCEDURE `closeContext(ctx IN ctxHandle);`	*Parameter:* `ctx(IN)` - the context handle to close. Closes all resources associated with this handle. After this you cannot use the handle for any other `DBMS_XMLGEN` function call.
Conversion Functions	
FUNCTION `convert( xmlData IN varchar2, flag IN NUMBER := ENTITY_ENCODE) RETURN VARCHAR2;`	Encodes or decodes the XML data string argument. ■ Encoding refers to replacing entity references such as '<' to their escaped equivalent, such as '`<`'. ■ Decoding refers to the reverse conversion.
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);` Introduced 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/>`.

Table 16–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
PROCEDURE `useNullAttributeIndicator( ctx IN ctxHandle, attrind IN BOOLEAN := TRUE);` Introduced in Oracle9*i* Release 2 (9.2.0.2).	`useNullAttributeIndicator` is a shortcut for `setNullHandling(ctx, NULL_ATTR)`.
PROCEDURE `setBindValue( ctx IN ctxHandle, bindVariableName IN VARCHAR2, bindValue IN VARCHAR2);` Introduced in Oracle Database 10*g* Release 1 (10.0.1).	Sets bind value for the bind variable appearing in the query string associated with the context handle. The query string with bind variables cannot be executed until all the bind variables are set values using `setBindValue()` call.
PROCEDURE `clearBindValue(ctx IN ctxHandle);` Introduced in Oracle Database 10*g* Release 1 (10.0.1).	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.

DBMS_XMLGEN Examples

Example 16–24 DBMS_XMLGEN: Generating Simple XML

This example creates an XML document by selecting employee data from an object-relational table and putting the resulting `CLOB` value into a table.

```
CREATE TABLE temp_clob_tab(result CLOB);

DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  result CLOB;
BEGIN
  qryCtx := DBMS_XMLGEN.newContext('SELECT * FROM hr.employees');
  -- Set the row header to be EMPLOYEE
  DBMS_XMLGEN.setRowTag(qryCtx, 'EMPLOYEE');
  -- Get the result
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES(result);
  --Close context
  DBMS_XMLGEN.closeContext(qryCtx);
END;
/
```

This query example generates the following XML (only part of the result is shown):

```
SELECT * FROM temp_clob_tab WHERE ROWNUM = 1;

RESULT
------------------------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <EMPLOYEE>
  <EMPLOYEE_ID>100</EMPLOYEE_ID>
  <FIRST_NAME>Steven</FIRST_NAME>
  <LAST_NAME>King</LAST_NAME>
```

```
  <EMAIL>SKING</EMAIL>
  <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
  <HIRE_DATE>17-JUN-87</HIRE_DATE>
  <JOB_ID>AD_PRES</JOB_ID>
  <SALARY>24000</SALARY>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </EMPLOYEE>

...

1 row selected.
```

Example 16–25 DBMS_XMLGEN: Generating Simple XML with Pagination (fetch)

Instead of generating all the XML data for all rows, you can use the `fetch` interface of `DBMS_XMLGEN` to retrieve a fixed number of rows each time. This speeds up response time and can help in scaling applications that need a Document Object Model (DOM) Application Program Interface (API) on the resulting XML, particularly if the number of rows is large.

The following example uses package `DBMS_XMLGEN` to retrieve results from table `hr.employees`:

```
-- Create a table to hold the results
CREATE TABLE temp_clob_tab(result clob);
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  result CLOB;
BEGIN
  -- Get the query context;
  qryCtx := DBMS_XMLGEN.newContext('SELECT * FROM hr.employees');
  -- Set the maximum number of rows to be 2
  DBMS_XMLGEN.setMaxRows(qryCtx, 2);
  LOOP
    -- Get the result
    result := DBMS_XMLGEN.getXML(qryCtx);
    -- If no rows were processed, then quit
    EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed(qryCtx) = 0;

    -- Do some processing with the lob data
    --   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;
/

SELECT * FROM temp_clob_tab WHERE rownum <3;

RESULT
--------------------------------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>100</EMPLOYEE_ID>
  <FIRST_NAME>Steven</FIRST_NAME>
  <LAST_NAME>King</LAST_NAME>
  <EMAIL>SKING</EMAIL>
```

```
  <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
  <HIRE_DATE>17-JUN-87</HIRE_DATE>
  <JOB_ID>AD_PRES</JOB_ID>
  <SALARY>24000</SALARY>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>101</EMPLOYEE_ID>
  <FIRST_NAME>Neena</FIRST_NAME>
  <LAST_NAME>Kochhar</LAST_NAME>
  <EMAIL>NKOCHHAR</EMAIL>
  <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
  <HIRE_DATE>21-SEP-89</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
</ROWSET>

<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>102</EMPLOYEE_ID>
  <FIRST_NAME>Lex</FIRST_NAME>
  <LAST_NAME>De Haan</LAST_NAME>
  <EMAIL>LDEHAAN</EMAIL>
  <PHONE_NUMBER>515.123.4569</PHONE_NUMBER>
  <HIRE_DATE>13-JAN-93</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>103</EMPLOYEE_ID>
  <FIRST_NAME>Alexander</FIRST_NAME>
  <LAST_NAME>Hunold</LAST_NAME>
  <EMAIL>AHUNOLD</EMAIL>
  <PHONE_NUMBER>590.423.4567</PHONE_NUMBER>
  <HIRE_DATE>03-JAN-90</HIRE_DATE>
  <JOB_ID>IT_PROG</JOB_ID>
  <SALARY>9000</SALARY>
  <MANAGER_ID>102</MANAGER_ID>
  <DEPARTMENT_ID>60</DEPARTMENT_ID>
 </ROW>
</ROWSET>

2 rows selected.
```

Example 16–26 DBMS_XMLGEN: Generating Nested XML With Object Types

This example uses 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, 'Mary',  10);
INSERT INTO new_employees   VALUES(40, 'John',  20);
INSERT INTO new_employees   VALUES(41, 'Jerry', 20);
COMMIT;
CREATE TYPE emplist_t AS TABLE OF emp_t;
/
CREATE TYPE dept_t AS OBJECT("@department_id" NUMBER,
                             department_name  VARCHAR2(20),
                             emplist          emplist_t);
/
CREATE TABLE temp_clob_tab(result CLOB);
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  result CLOB;
BEGIN
  DBMS_XMLGEN.setRowTag(qryCtx, NULL);
  qryCtx := DBMS_XMLGEN.newContext
    ('SELECT dept_t(department_id,
                    department_name,
                    CAST(MULTISET
                           (SELECT e.employee_id, e.last_name
                              FROM new_employees e
                              WHERE e.department_id = d.department_id)
                         AS emplist_t))
        AS deptxml
        FROM new_departments d');
  -- now get the result
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES (result);
  -- close context
  DBMS_XMLGEN.closeContext(qryCtx);
END;
/
SELECT * FROM temp_clob_tab;
```

Here is the resulting XML:

```
RESULT
--------------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <DEPTXML department_id="10">
   <DEPARTMENT_NAME>SALES</DEPARTMENT_NAME>
   <EMPLIST>
    <EMP_T employee_id="30">
     <LAST_NAME>Scott</LAST_NAME>
    </EMP_T>
    <EMP_T employee_id="31">
     <LAST_NAME>Mary</LAST_NAME>
    </EMP_T>
   </EMPLIST>
  </DEPTXML>
 </ROW>
 <ROW>
  <DEPTXML department_id="20">
   <DEPARTMENT_NAME>ACCOUNTING</DEPARTMENT_NAME>
```

```
   <EMPLIST>
    <EMP_T employee_id="40">
     <LAST_NAME>John</LAST_NAME>
    </EMP_T>
    <EMP_T employee_id="41">
     <LAST_NAME>Jerry</LAST_NAME>
    </EMP_T>
   </EMPLIST>
  </DEPTXML>
 </ROW>
</ROWSET>

1 row selected.
```

With relational data, the result is an XML document without nested elements. To obtain nested XML structures, you can use object-relational data, where the mapping is as follows:

- ***Object types*** map as an XML element – see Chapter 5, "XML Schema Storage and Query: Basic".
- ***Attributes of the type***, map to sub-elements of the parent element

> **Note:** Complex structures can be obtained by using object types and creating object views or object tables. A canonical mapping is used to map object instances to XML.
>
> When used in column names or attribute names, the at-sign (`@`) is translated into an attribute of the enclosing XML element in the mapping.

Example 16–27 DBMS_XMLGEN: Generating Nested XML With User-Defined Datatype Instances

When you provide a user-defined datatype instance to `DBMS_XMLGEN` functions, the user-defined datatype instance is mapped to an XML document using canonical mapping: the *attributes* of the user-defined datatype are mapped to XML *elements*. Attributes with names starting with an at-sign (`@`) are mapped to attributes of the preceding element.

User-defined datatype instances 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 preceded by an at-sign (@) to indicate that it must
-- be mapped as an attribute of the enclosing Employee element.
CREATE TYPE emp_t AS OBJECT("@empno" NUMBER,  -- empno defined as attribute
                            ename   VARCHAR2(20));
/
INSERT INTO DEPT VALUES(10, 'Sports');
INSERT INTO DEPT VALUES(20, 'Accounting');
INSERT INTO EMP VALUES(200, 'John',  10);
INSERT INTO EMP VALUES(300, 'Jack',  10);
```

```
INSERT INTO EMP VALUES(400, 'Mary',  20);
INSERT INTO EMP VALUES(500, 'Jerry', 20);
COMMIT;
CREATE TYPE emplist_t AS TABLE OF emp_t;
/
CREATE TYPE dept_t AS OBJECT("@deptno" NUMBER,
                             dname     VARCHAR2(20),
                             emplist   emplist_t);
/
-- Department type dept_t contains a list of employees.
-- We can now query the employee and department tables and get
-- the result as an XML document, as follows:
CREATE TABLE temp_clob_tab(result CLOB);
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  RESULT CLOB;
BEGIN
  -- get query context
  qryCtx := DBMS_XMLGEN.newContext(
    'SELECT dept_t(deptno,
                   dname,
                   CAST(MULTISET(SELECT empno, ename
                                 FROM emp e
                                 WHERE e.deptno = d.deptno)
                        AS emplist_t))
       AS deptxml
       FROM dept d');
  -- set maximum number of rows to 5
  DBMS_XMLGEN.setMaxRows(qryCtx, 5);
  -- set no row tag for this result, since there is a single ADT column
  DBMS_XMLGEN.setRowTag(qryCtx, NULL);
  LOOP
    -- get result
    result := DBMS_XMLGEN.getXML(qryCtx);
    -- if there were no rows processed, then quit
    EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed(qryCtx) = 0;
    -- do something with the result
    INSERT INTO temp_clob_tab VALUES (result);
  END LOOP;
END;
/
```

Function `MULTISET` treats the employees working in the department as a list, and function `CAST` assigns this list to the appropriate collection type. A department instance is created, and `DBMS_XMLGEN` routines create the XML for the object instance.

```
SELECT * FROM temp_clob_tab;

RESULT
--------------------------------
<?xml version="1.0"?>
<ROWSET>
 <DEPTXML deptno="10">
  <DNAME>Sports</DNAME>
  <EMPLIST>
   <EMP_T empno="200">
    <ENAME>John</ENAME>
   </EMP_T>
   <EMP_T empno="300">
    <ENAME>Jack</ENAME>
```

```
   </EMP_T>
  </EMPLIST>
 </DEPTXML>
 <DEPTXML deptno="20">
  <DNAME>Accounting</DNAME>
  <EMPLIST>
   <EMP_T empno="400">
    <ENAME>Mary</ENAME>
   </EMP_T>
   <EMP_T empno="500">
    <ENAME>Jerry</ENAME>
   </EMP_T>
  </EMPLIST>
 </DEPTXML>
</ROWSET>

1 row selected.
```

The default name ROW is not present because we set that to NULL. The deptno and empno have become attributes of the enclosing element.

Example 16–28 DBMS_XMLGEN: Generating an XML Purchase Order

This example uses DBMS_XMLGEN.getXMLType() to generate a purchase order in XML format using object views.

```
-- Create relational schema and define object views
-- DBMS_XMLGEN maps user-defined datatype attribute names that start
--    with an at-sign (@) to XML attributes

-- Purchase Order Object View Model

-- PhoneList varray object type
CREATE TYPE phonelist_vartyp AS VARRAY(10) OF VARCHAR2(20)
/
-- Address object type
CREATE TYPE address_typ AS OBJECT(Street VARCHAR2(200),
                                  City   VARCHAR2(200),
                                  State  CHAR(2),
                                  Zip    VARCHAR2(20))
/
-- Customer object type
CREATE TYPE customer_typ AS OBJECT(CustNo    NUMBER,
                                   CustName  VARCHAR2(200),
                                   Address   address_typ,
                                   PhoneList phonelist_vartyp)
/
-- StockItem object type
CREATE TYPE stockitem_typ AS OBJECT("@StockNo" NUMBER,
                                    Price      NUMBER,
                                    TaxRate    NUMBER)
/
-- LineItems object type
CREATE TYPE lineitem_typ AS OBJECT("@LineItemNo" NUMBER,
                                   Item          stockitem_typ,
                                   Quantity      NUMBER,
                                   Discount      NUMBER)
/
-- LineItems 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)
/
-- Populate data
-------------------
-- Establish Inventory
INSERT INTO stock_tab VALUES(1004, 6750.00, 2);
INSERT INTO stock_tab VALUES(1011, 4500.23, 2);
INSERT INTO stock_tab VALUES(1534, 2234.00, 2);
INSERT INTO stock_tab VALUES(1535, 3456.23, 2);
-- Register Customers
INSERT INTO customer_tab
  VALUES (1, 'Jean Nance', '2 Avocet Drive',
          'Redwood Shores', 'CA', '95054',
          '415-555-1212', NULL, NULL);
INSERT INTO customer_tab
  VALUES (2, 'John Nike', '323 College Drive',
          'Edison', 'NJ', '08820',
          '609-555-1212', '201-555-1212', NULL);
-- Place orders
INSERT INTO po_tab
  VALUES (1001, 1, '10-APR-1997', '10-MAY-1997',
          NULL, NULL, NULL, NULL);
INSERT INTO po_tab
  VALUES (2001, 2, '20-APR-1997', '20-MAY-1997',
          '55 Madison Ave', 'Madison', 'WI', '53715');
-- Detail line items
INSERT INTO lineitems_tab VALUES(01, 1001, 1534, 12,  0);
INSERT INTO lineitems_tab VALUES(02, 1001, 1535, 10, 10);
INSERT INTO lineitems_tab VALUES(01, 2001, 1004,  1,  0);
INSERT INTO lineitems_tab VALUES(02, 2001, 1011,  2,  1);

-- Use package DBMS_XMLGEN to generate purchase order in XML format
--   and store XMLType in table po_xml
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  pxml XMLType;
  cxml CLOB;
BEGIN
  -- get query context;
  qryCtx := DBMS_XMLGEN.newContext('SELECT pono,deref(cust_ref) customer,
                                           p.orderdate,
                                           p.shipdate,
                                           lineitems_ntab lineitems,
                                           shiptoaddr
                                      FROM po p');
  -- set maximum number of rows to be 1,
  DBMS_XMLGEN.setMaxRows(qryCtx, 1);
  -- set ROWSET tag to NULL and ROW tag to PurchaseOrder
  DBMS_XMLGEN.setRowSetTag(qryCtx, NULL);
  DBMS_XMLGEN.setRowTag(qryCtx, 'PurchaseOrder');
  LOOP
    -- get purchase order in XML format
```

```
    pxml := DBMS_XMLGEN.getXMLType(qryCtx);
    -- if there were no rows processed, then quit
    EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed(qryCtx) = 0;
    -- Store XMLType po in po_xml table (get the pono out)
    INSERT INTO po_xml_tab(poid, poDoc)
      VALUES(pxml.extract('//PONO/text()').getNumberVal(), pxml);
  END LOOP;
END;
/
```

This query then produces two XML purchase-order documents:

```
SELECT x.podoc.getClobVal() xpo FROM po_xml_tab x;

XPO
--------------------------------------------------
 <PurchaseOrder>
  <PONO>1001</PONO>
  <CUSTOMER>
   <CUSTNO>1</CUSTNO>
   <CUSTNAME>Jean Nance</CUSTNAME>
   <ADDRESS>
    <STREET>2 Avocet Drive</STREET>
    <CITY>Redwood Shores</CITY>
    <STATE>CA</STATE>
    <ZIP>95054</ZIP>
   </ADDRESS>
   <PHONELIST>
    <VARCHAR2>415-555-1212</VARCHAR2>
   </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>10-APR-97</ORDERDATE>
  <SHIPDATE>10-MAY-97 12.00.00.000000 AM</SHIPDATE>
  <LINEITEMS>
   <LINEITEM_TYP LineItemNo="1">
    <ITEM StockNo="1534">
     <PRICE>2234</PRICE>
     <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>12</QUANTITY>
    <DISCOUNT>0</DISCOUNT>
   </LINEITEM_TYP>
   <LINEITEM_TYP LineItemNo="2">
    <ITEM StockNo="1535">
     <PRICE>3456.23</PRICE>
     <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>10</QUANTITY>
    <DISCOUNT>10</DISCOUNT>
   </LINEITEM_TYP>
  </LINEITEMS>
  <SHIPTOADDR/>
 </PurchaseOrder>

 <PurchaseOrder>
  <PONO>2001</PONO>
  <CUSTOMER>
   <CUSTNO>2</CUSTNO>
   <CUSTNAME>John Nike</CUSTNAME>
   <ADDRESS>
    <STREET>323 College Drive</STREET>
```

```
    <CITY>Edison</CITY>
    <STATE>NJ</STATE>
    <ZIP>08820</ZIP>
   </ADDRESS>
   <PHONELIST>
    <VARCHAR2>609-555-1212</VARCHAR2>
    <VARCHAR2>201-555-1212</VARCHAR2>
   </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>20-APR-97</ORDERDATE>
  <SHIPDATE>20-MAY-97 12.00.00.000000 AM</SHIPDATE>
  <LINEITEMS>
   <LINEITEM_TYP LineItemNo="1">
    <ITEM StockNo="1004">
     <PRICE>6750</PRICE>
     <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>1</QUANTITY>
    <DISCOUNT>0</DISCOUNT>
   </LINEITEM_TYP>
   <LINEITEM_TYP LineItemNo="2">
    <ITEM StockNo="1011">
     <PRICE>4500.23</PRICE>
     <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>2</QUANTITY>
    <DISCOUNT>1</DISCOUNT>
   </LINEITEM_TYP>
  </LINEITEMS>
  <SHIPTOADDR>
   <STREET>55 Madison Ave</STREET>
   <CITY>Madison</CITY>
   <STATE>WI</STATE>
   <ZIP>53715</ZIP>
  </SHIPTOADDR>
 </PurchaseOrder>

2 rows selected.
```

Example 16–29 DBMS_XMLGEN: Generating a New Context Handle from a Ref Cursor

This example shows how to open a cursor variable for a query and use that cursor variable to create a new context handle for DBMS_XMLGEN.

```
CREATE TABLE emp_tab(emp_id       NUMBER PRIMARY KEY,
                     name         VARCHAR2(20),
                     dept_id      NUMBER);
Table created.
INSERT INTO emp_tab VALUES(122, 'Scott',  301);
1 row created.
INSERT INTO emp_tab VALUES(123, 'Mary',   472);
1 row created.
INSERT INTO emp_tab VALUES(124, 'John',   93);
1 row created.
INSERT INTO emp_tab VALUES(125, 'Howard', 488);
1 row created.
INSERT INTO emp_tab VALUES(126, 'Sue',    16);
1 row created.
COMMIT;
```

```
DECLARE
  ctx     NUMBER;
  maxrow  NUMBER;
  xmldoc  CLOB;
  refcur  SYS_REFCURSOR;
BEGIN
  DBMS_LOB.createtemporary(xmldoc, TRUE);
  maxrow := 3;
  OPEN refcur FOR 'SELECT * FROM emp_tab WHERE ROWNUM <= :1' USING maxrow;
  ctx := DBMS_XMLGEN.newContext(refcur);
   -- xmldoc will have 3 rows
  DBMS_XMLGEN.getXML(ctx, xmldoc, DBMS_XMLGEN.NONE);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_LOB.freetemporary(xmldoc);
  CLOSE refcur;
  DBMS_XMLGEN.closeContext(ctx);
END;
/
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMP_ID>122</EMP_ID>
  <NAME>Scott</NAME>
  <DEPT_ID>301</DEPT_ID>
 </ROW>
 <ROW>
  <EMP_ID>123</EMP_ID>
  <NAME>Mary</NAME>
  <DEPT_ID>472</DEPT_ID>
 </ROW>
 <ROW>
  <EMP_ID>124</EMP_ID>
  <NAME>John</NAME>
  <DEPT_ID>93</DEPT_ID>
 </ROW>
</ROWSET>

PL/SQL procedure successfully completed.
```

See Also: *Oracle Database PL/SQL User's Guide and Reference* for more information on cursor variables (`REF CURSOR`)

Example 16–30 DBMS_XMLGEN: Specifying NULL Handling

```
CREATE TABLE emp_tab(emp_id       NUMBER PRIMARY KEY,
                     name         VARCHAR2(20),
                     dept_id      NUMBER);
Table created.
INSERT INTO emp_tab VALUES(30, 'Scott', NULL);
1 row created.
INSERT INTO emp_tab VALUES(31, 'Marry', NULL);
1 row created.
INSERT INTO emp_tab VALUES(40, 'John', NULL);
1 row created.
COMMIT;
CREATE TABLE temp_clob_tab(result CLOB);
Table created.

DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  result CLOB;
```

```
BEGIN
  qryCtx := DBMS_XMLGEN.newContext('SELECT * FROM emp_tab where name = :NAME');
  -- Set the row header to be EMPLOYEE
  DBMS_XMLGEN.setRowTag(qryCtx, 'EMPLOYEE');
  -- Drop nulls
  DBMS_XMLGEN.setBindValue(qryCtx, 'NAME', 'Scott');
  DBMS_XMLGEN.setNullHandling(qryCtx, DBMS_XMLGEN.DROP_NULLS);
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES(result);
  -- Null attribute
  DBMS_XMLGEN.setBindValue(qryCtx, 'NAME', 'Marry');
  DBMS_XMLGEN.setNullHandling(qryCtx, DBMS_XMLGEN.NULL_ATTR);
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES(result);
  -- Empty tag
  DBMS_XMLGEN.setBindValue(qryCtx, 'NAME', 'John');
  DBMS_XMLGEN.setNullHandling(qryCtx, DBMS_XMLGEN.EMPTY_TAG);
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES(result);
  --Close context
  DBMS_XMLGEN.closeContext(qryCtx);
END;
/

PL/SQL procedure successfully completed.

SELECT * FROM temp_clob_tab;

RESULT
-----------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <EMPLOYEE>
  <EMP_ID>30</EMP_ID>
  <NAME>Scott</NAME>
 </EMPLOYEE>
</ROWSET>

<?xml version="1.0"?>
<ROWSET xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
 <EMPLOYEE>
  <EMP_ID>31</EMP_ID>
  <NAME>Marry</NAME>
  <DEPT_ID xsi:nil = "true"/>
 </EMPLOYEE>
</ROWSET>

<?xml version="1.0"?>
<ROWSET>
 <EMPLOYEE>
  <EMP_ID>40</EMP_ID>
  <NAME>John</NAME>
  <DEPT_ID/>
 </EMPLOYEE>
</ROWSET>

3 rows selected.
```

Example 16–31 DBMS_XMLGEN : Generating Recursive XML with a Hierarchical Query

Function `DBMS_XMLGEN.newContextFromHierarchy` takes as argument a hierarchical query string, which is typically formulated with a `CONNECT BY` clause. It returns a context that can be used to generate a hierarchical XML document with recursive elements.

The hierarchical query returns two columns, the level number (a pseudocolumn generated by `CONNECT BY` query) and an `XMLType`. The level is used to determine the position of the `XMLType` value within the hierarchy of the result XML document.

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()`.

```
CREATE TABLE sqlx_display(id NUMBER, xmldoc XMLType);
Table created.
DECLARE
  qryctx DBMS_XMLGEN.ctxhandle;
  result XMLType;
BEGIN
  qryctx :=
    DBMS_XMLGEN.newcontextFromHierarchy(
      'SELECT level,
              XMLElement("employees",
                         XMLElement("enumber", employee_id),
                         XMLElement("name", last_name),
                         XMLElement("Salary", salary),
                         XMLElement("Hiredate", hire_date))
        FROM hr.employees
        START WITH last_name=''De Haan'' CONNECT BY PRIOR employee_id=manager_id
        ORDER SIBLINGS BY hire_date');
  result := DBMS_XMLGEN.getxmltype(qryctx);
  DBMS_OUTPUT.put_line('<result num rows>');
  DBMS_OUTPUT.put_line(to_char(DBMS_XMLGEN.getNumRowsProcessed(qryctx)));
  DBMS_OUTPUT.put_line('</result num rows>');
  INSERT INTO sqlx_display VALUES (2, result);
  COMMIT;
  DBMS_XMLGEN.closecontext(qryctx);
END;
/
<result num rows>
6
</result num rows>
PL/SQL procedure successfully completed.

SELECT xmldoc FROM sqlx_display WHERE id = 2;

XMLDOC
--------------------------------------------------
<?xml version="1.0"?>
<employees>
  <enumber>102</enumber>
  <name>De Haan</name>
  <Salary>17000</Salary>
  <Hiredate>1993-01-13</Hiredate>
  <employees>
    <enumber>103</enumber>
    <name>Hunold</name>
    <Salary>9000</Salary>
```

```
    <Hiredate>1990-01-03</Hiredate>
    <employees>
      <enumber>104</enumber>
      <name>Ernst</name>
      <Salary>6000</Salary>
      <Hiredate>1991-05-21</Hiredate>
    </employees>
    <employees>
      <enumber>105</enumber>
      <name>Austin</name>
      <Salary>4800</Salary>
      <Hiredate>1997-06-25</Hiredate>
    </employees>
    <employees>
      <enumber>106</enumber>
      <name>Pataballa</name>
      <Salary>4800</Salary>
      <Hiredate>1998-02-05</Hiredate>
    </employees>
    <employees>
      <enumber>107</enumber>
      <name>Lorentz</name>
      <Salary>4200</Salary>
      <Hiredate>1999-02-07</Hiredate>
    </employees>
  </employees>
</employees>

1 row selected.
```

By default, the `ROWSET` tag is `NULL`: there is no default `ROWSET` tag used to enclose the XML result. However, you can explicitly set the `ROWSET` tag by using procedure `setRowSetTag()`, as follows:

```
CREATE TABLE gg(x XMLType);
Table created.
DECLARE
  qryctx DBMS_XMLGEN.ctxhandle;
  result CLOB;
BEGIN
  qryctx := DBMS_XMLGEN.newcontextFromHierarchy(
              'SELECT level,
                      XMLElement("NAME", last_name) AS myname FROM hr.employees
               CONNECT BY PRIOR employee_id=manager_id
               START WITH employee_id = 102');
  DBMS_XMLGEN.setRowSetTag(qryctx, 'mynum_hierarchy');
  result:=DBMS_XMLGEN.getxml(qryctx);
  DBMS_OUTPUT.put_line('<result num rows>');
  DBMS_OUTPUT.put_line(to_char(DBMS_XMLGEN.getNumRowsProcessed(qryctx)));
  DBMS_OUTPUT.put_line('</result num rows>');
  INSERT INTO gg VALUES(XMLType(result));
  COMMIT;
  DBMS_XMLGEN.closecontext(qryctx);
END;
/
<result num rows>
6
</result num rows>
PL/SQL procedure successfully completed.
```

```
SELECT * FROM gg;

X
------------------------------------------------------------
<?xml version="1.0"?>
<mynum_hierarchy>
  <NAME>De Haan
    <NAME>Hunold
      <NAME>Ernst</NAME>
      <NAME>Austin</NAME>
      <NAME>Pataballa</NAME>
      <NAME>Lorentz</NAME>
    </NAME>
  </NAME>
</mynum_hierarchy>

1 row selected.
```

Example 16–32 DBMS_XMLGEN : Binding Query Variables with setBindValue()

If the query string used to create a context contains host variables, you can use `setBindValue()` to give the variables values before query execution.

```
-- Bind one variable
DECLARE
  ctx NUMBER;
  xmldoc CLOB;
BEGIN
  ctx := DBMS_XMLGEN.newContext(
           'SELECT * FROM employees WHERE employee_id = :NO');
  DBMS_XMLGEN.setBindValue(ctx, 'NO', '145');
  xmldoc := DBMS_XMLGEN.getXML(ctx);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_XMLGEN.closeContext(ctx);
EXCEPTION
  WHEN OTHERS THEN DBMS_XMLGEN.closeContext(ctx);
  RAISE;
END;
/
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>145</EMPLOYEE_ID>
  <FIRST_NAME>John</FIRST_NAME>
  <LAST_NAME>Russell</LAST_NAME>
  <EMAIL>JRUSSEL</EMAIL>
  <PHONE_NUMBER>011.44.1344.429268</PHONE_NUMBER>
  <HIRE_DATE>01-OCT-96</HIRE_DATE>
  <JOB_ID>SA_MAN</JOB_ID>
  <SALARY>14000</SALARY>
  <COMMISSION_PCT>.4</COMMISSION_PCT>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
</ROWSET>

PL/SQL procedure successfully completed.

--Bind one variable twice with different values
DECLARE
  ctx NUMBER;
```

```
  xmldoc CLOB;
BEGIN
  ctx := DBMS_XMLGEN.newContext('SELECT * FROM employees
                                   WHERE hire_date = :MDATE');
  DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '01-OCT-96');
  xmldoc := DBMS_XMLGEN.getXML(ctx);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '10-MAR-97');
  xmldoc := DBMS_XMLGEN.getXML(ctx);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_XMLGEN.closeContext(ctx);
EXCEPTION
  WHEN OTHERS THEN DBMS_XMLGEN.closeContext(ctx);
  RAISE;
END;
/
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>145</EMPLOYEE_ID>
  <FIRST_NAME>John</FIRST_NAME>
  <LAST_NAME>Russell</LAST_NAME>
  <EMAIL>JRUSSEL</EMAIL>
  <PHONE_NUMBER>011.44.1344.429268</PHONE_NUMBER>
  <HIRE_DATE>01-OCT-96</HIRE_DATE>
  <JOB_ID>SA_MAN</JOB_ID>
  <SALARY>14000</SALARY>
  <COMMISSION_PCT>.4</COMMISSION_PCT>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
</ROWSET>

<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>147</EMPLOYEE_ID>
  <FIRST_NAME>Alberto</FIRST_NAME>
  <LAST_NAME>Errazuriz</LAST_NAME>
  <EMAIL>AERRAZUR</EMAIL>
  <PHONE_NUMBER>011.44.1344.429278</PHONE_NUMBER>
  <HIRE_DATE>10-MAR-97</HIRE_DATE>
  <JOB_ID>SA_MAN</JOB_ID>
  <SALARY>12000</SALARY>
  <COMMISSION_PCT>.3</COMMISSION_PCT>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>159</EMPLOYEE_ID>
  <FIRST_NAME>Lindsey</FIRST_NAME>
  <LAST_NAME>Smith</LAST_NAME>
  <EMAIL>LSMITH</EMAIL>
  <PHONE_NUMBER>011.44.1345.729268</PHONE_NUMBER>
  <HIRE_DATE>10-MAR-97</HIRE_DATE>
  <JOB_ID>SA_REP</JOB_ID>
  <SALARY>8000</SALARY>
  <COMMISSION_PCT>.3</COMMISSION_PCT>
  <MANAGER_ID>146</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
```

```
 </ROW>
</ROWSET>
PL/SQL procedure successfully completed.
-- Bind two variables
DECLARE
  ctx NUMBER;
  xmldoc CLOB;
BEGIN
  ctx := DBMS_XMLGEN.newContext('SELECT * FROM employees
                                   WHERE employee_id = :NO
                                     AND hire_date = :MDATE');
  DBMS_XMLGEN.setBindValue(ctx, 'NO', '145');
  DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '01-OCT-96');
  xmldoc := DBMS_XMLGEN.getXML(ctx);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_XMLGEN.closeContext(ctx);
EXCEPTION
  WHEN OTHERS THEN DBMS_XMLGEN.closeContext(ctx);
  RAISE;
END;
/
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>145</EMPLOYEE_ID>
  <FIRST_NAME>John</FIRST_NAME>
  <LAST_NAME>Russell</LAST_NAME>
  <EMAIL>JRUSSEL</EMAIL>
  <PHONE_NUMBER>011.44.1344.429268</PHONE_NUMBER>
  <HIRE_DATE>01-OCT-96</HIRE_DATE>
  <JOB_ID>SA_MAN</JOB_ID>
  <SALARY>14000</SALARY>
  <COMMISSION_PCT>.4</COMMISSION_PCT>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
</ROWSET>
PL/SQL procedure successfully completed.
```

Generating XML Using SQL Function SYS_XMLGEN

This Oracle Database-specific SQL function is similar to the SQL/XML standard function `XMLElement`, except that it takes a single argument and converts the result to an `XMLType` instance. Unlike the other XML generation functions, `sys_XMLGen` always returns a well-formed XML *document*. Unlike package `DBMS_XMLGEN`, which operates at a query level, `sys_XMLGen` operates at the row level returning a XML *document for each row*.

Example 16–33 Using SYS_XMLGEN to Create XML

In this query, SQL function `sys_XMLGen` queries XML instances and returns an XML document for each row of relational data:

```
SELECT sys_XMLGen(employee_id) AS "result"
  FROM employees WHERE first_name LIKE 'John%';
```

The resulting XML documents are as follows:

```
result
---------------
```

```
<?xml version="1.0"?>
<EMPLOYEE_ID>110</EMPLOYEE_ID>

<?xml version="1.0"?>
<EMPLOYEE_ID>139</EMPLOYEE_ID>

<?xml version="1.0"?>
<EMPLOYEE_ID>145</EMPLOYEE_ID>

3 rows selected.
```

SYS_XMLGEN Syntax

SQL function `sys_XMLGen` takes as argument a scalar value, object type, or `XMLType` instance to be converted to an XML document. It also takes an optional `XMLFormat` object (previously called `XMLGenFormatType)`, which you can use to specify formatting options for the resulting XML document. The syntax is shown in Figure 16–15.

Figure 16–15 SYS_XMLGEN Syntax

Expression *expr* evaluates to a particular row and column of the database. It can be a scalar value, a user-defined datatype instance, or an `XMLType` instance.

- If *expr* evaluates to a scalar value, then the function returns an XML element containing the scalar value.
- If *expr* evaluates to a user-defined datatype instance, then the function maps the user-defined datatype attributes to XML elements.
- If *expr* evaluates to an `XMLType` instance, then the function encloses the document in an XML element whose default tag name is `ROW`.

By default, the elements of the XML document match the *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.

You can use a `WHERE` clause in a query to suppress `<ROW/>` tags with `sys_XMLGen`, if you do not want `NULL` values represented:

```
SELECT sys_XMLGen(x) FROM table_name WHERE x IS NOT NULL;
```

Example 16–34 SYS_XMLGEN: Generating an XML Element from a Database Column

The following example retrieves the employee `first_name` from sample-schema table `hr.employees`, where the `employee_id` value is `110`, and generates an `XMLType` instance containing an XML document with an `FIRST_NAME` element.

```
SELECT sys_XMLGen(first_name).getStringVal()
  FROM employees
  WHERE employee_id = 110;

SYS_XMLGEN(FIRST_NAME).GETSTRINGVAL()
--------------------------------------
<?xml version="1.0"?>
<FIRST_NAME>John</FIRST_NAME>
```

```
1 row selected.
```

Advantages of Using SYS_XMLGEN

SQL function sys_XMLGen has the following advantages:

- You can create and query XML instances *within* SQL queries.
- Using the object-relational infrastructure, you can create complex and nested XML instances from simple relational tables. For example, when you use an XMLType view that uses sys_XMLGen on top of an object type, Oracle XML DB rewrites these queries when possible. See also Chapter 6, "XPath Rewrite".

sys_XMLGen creates an XML document from a user-defined datatype instance, a scalar value, or an XMLType instance. It returns an XMLType instance.

sys_XMLGen also accepts an optional XMLFormat object as argument, which you can use to customize the result. A NULL format object implies that the default mapping action is to be used.

Using XMLFormat Object Type

You can use the XMLFormat object to specify formatting arguments for SQL functions sys_XMLGen and sys_XMLAgg.

Function sys_XMLGen returns an XMLType instance containing an XML document. Oracle Database provides the XMLFormat object to format the output of sys_XMLGen.

Table 16–2 lists the attributes of object XMLFormat.

Table 16–2 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 column name is used as the default value. Otherwise, the default value is ROWSET. When schemaType is set to USE_GIVEN_SCHEMA, this attribute also provides the name of the XML schema element.
schemaType	VARCHAR2(100)	The type of schema generation for the output document. Valid values are 'NO_SCHEMA' and 'USE_GIVEN_SCHEMA'. The default value is 'NO_SCHEMA'.
schemaName	VARCHAR2(4000)	The name of the target schema used if schemaType is 'USE_GIVEN_SCHEMA'. If you specify schemaName, then the enclosing tag is used as the element name.
targetNameSpace	VARCHAR2(4000)	The target namespace if the schema is specified (that is, schemaType is GEN_SCHEMA_*, or USE_GIVEN_SCHEMA)
dburl	VARCHAR2(2000)	The URL to the database to be used if WITH_SCHEMA is specified. If this attribute is not specified, then a relative URL reference is used for the URL to the types.
processingIns	VARCHAR2(4000)	User-provided processing instructions. They are appended to the top of the function output, before the element.

You can use method createFormat() to implement the XMLFormat object. Method createFormat() of object XMLFormat accepts as arguments the enclosing element name, the XML-schema type, and the XML-schema name. Default values are provided for the other XMLFormat attributes.

See Also:

- Example 16–37 for an example of using `createFormat()` to name the root element that is output by `sys_XMLGen`
- *Oracle Database SQL Reference* for more information on `sys_XMLGen` and the `XMLFormat` object

Example 16–35 SYS_XMLGEN: Converting a Scalar Value to XML Element Contents

SQL function `sys_XMLGen` converts a scalar value to an element that contains the scalar value. For example, the following query returns an XML document that contains the `employee_id` value as an element containing that value:

```
SELECT sys_XMLGen(employee_id) FROM hr.employees WHERE ROWNUM < 2;

SYS_XMLGEN(EMPLOYEE_ID)
--------------------------
<?xml version="1.0"?>
<EMPLOYEE_ID>100</EMPLOYEE_ID>

1 row selected.
```

The enclosing element name, in this case `EMPLOYEE_ID`, is derived from the column name passed to `sys_XMLGen`. The query result is a single row containing an `XMLType` instance that corresponds to a complete XML document.

Example 16–36 SYS_XMLGEN: Default Element Name ROW

In Example 16–35, the column name `EMPLOYEE_ID` is used by default for the XML element name. If the column name cannot be derived directly, then the default name `ROW` is used instead:

```
SELECT sys_XMLGen(employee_id*2) FROM hr.employees WHERE ROWNUM < 2;

SYS_XMLGEN(EMPLOYEE_ID*2)
-------------------------
<?xml version="1.0"?>
<ROW>200</ROW>

1 row selected.
```

In this example, the argument to `sys_XMLGen` is not a simple column name, so the name of the output element tag cannot simply be a column name – the default element name, `ROW`, is used.

You can override the default `ROW` tag by supplying an `XMLFormat` object as the second `sys_XMLGen` argument – see Example 16–37 for an example.

Example 16–37 Overriding the Default Element Name: Using SYS_XMLGEN with XMLFormat

In this example, a formatting argument is supplied to `sys_XMLGen`, to name the element explicitly:

```
SELECT sys_XMLGen(employee_id*2,
                  XMLFormat.createformat('DOUBLE_ID')).getclobval()
 FROM hr.employees WHERE ROWNUM < 2;

SYS_XMLGEN(EMPLOYEE_ID*2,XMLFORMAT.CREATEFORMAT('EMPLOYEE_ID')).GETCLOBVAL()
--------------------------------------------------------------------------------
<?xml version="1.0"?>
```

```
<DOUBLE_ID>200</DOUBLE_ID>

1 row selected.
```

Example 16–38 SYS_XMLGEN: Converting a User-Defined Datatype Instance to XML

When you provide a user-defined datatype instance as an argument to `sys_XMLGen`, the instance is canonically mapped to an XML document. In this mapping, the user-defined datatype attributes are mapped to XML elements.

Any datatype attributes with names that start with an at sign (`@`) are mapped to attributes of the preceding XML element. User-defined datatype instances can be used to obtain nesting in the resulting XML document.

You can generate hierarchical XML for the employee-and-department example (see "Generating XML Using DBMS_XMLGEN" on page 16-24) as follows:

```
CREATE OR REPLACE TYPE hr.emp_t AS OBJECT(empno    NUMBER(6),
                                          ename    VARCHAR2(25),
                                          job      VARCHAR2(10),
                                          mgr      NUMBER(6),
                                          hiredate DATE,
                                          sal      NUMBER(8,2),
                                          comm     NUMBER(2,2));
/
Type created.
CREATE OR REPLACE TYPE hr.emplist_t AS TABLE OF emp_t;
/
Type created.
CREATE OR REPLACE TYPE hr.dept_t AS OBJECT(deptno  NUMBER(4),
                                           dname   VARCHAR2(30),
                                           loc     VARCHAR2(4),
                                           emplist emplist_t);
/
Type created.

SELECT sys_XMLGen(
         dept_t(department_id,
                department_name,
                d.location_id,
                CAST(MULTISET(SELECT emp_t(e.employee_id, e.last_name, e.job_id,
                                           e.manager_id, e.hire_date, e.salary,
                                           e.commission_pct)
                              FROM hr.employees e
                              WHERE e.department_id = d.department_id)
                    AS emplist_t))).getClobVal()
  AS deptxml
  FROM hr.departments d WHERE department_id = 10 OR department_id = 20;
```

SQL function `MULTISET` treats the result of the subset of employees working in the department as a list, and the `CAST` then assigns this to the appropriate collection type. A department-type (`dept_t`) element is wrapped around this to create the XML data for the object instance.

The result is as follows. The default name `ROW` is present because the function cannot deduce the name of the input operand directly.

```
DEPTXML
-------------------------------------
<?xml version="1.0"?>
<ROW>
```

```
 <DEPTNO>10</DEPTNO>
 <DNAME>Administration</DNAME>
 <LOC>1700</LOC>
 <EMPLIST>
  <EMP_T>
   <EMPNO>200</EMPNO>
   <ENAME>Whalen</ENAME>
   <JOB>AD_ASST</JOB>
   <MGR>101</MGR>
   <HIREDATE>17-SEP-87</HIREDATE>
   <SAL>4400</SAL>
  </EMP_T>
 </EMPLIST>
</ROW>

<?xml version="1.0"?>
<ROW>
 <DEPTNO>20</DEPTNO>
 <DNAME>Marketing</DNAME>
 <LOC>1800</LOC>
 <EMPLIST>
  <EMP_T>
   <EMPNO>201</EMPNO>
   <ENAME>Hartstein</ENAME>
   <JOB>MK_MAN</JOB>
   <MGR>100</MGR>
   <HIREDATE>17-FEB-96</HIREDATE>
   <SAL>13000</SAL>
  </EMP_T>
  <EMP_T>
   <EMPNO>202</EMPNO>
   <ENAME>Fay</ENAME>
   <JOB>MK_REP</JOB>
   <MGR>201</MGR>
   <HIREDATE>17-AUG-97</HIREDATE>
   <SAL>6000</SAL>
  </EMP_T>
 </EMPLIST>
</ROW>

2 rows selected.
```

> **Note:** The difference between using SQL function `sys_XMLGen` and PL/SQL package `DBMS_XMLGEN` is apparent from the preceding example. Function `sys_XMLGen` works inside SQL queries, and operates on the expressions and columns within the rows; package `DBMS_XMLGEN` works on the entire result set.

Example 16–39 SYS_XMLGEN: Converting an XMLType Instance

If you pass an XML document to function `sys_XMLGen`, this function encloses the document (or fragment) with an element, whose tag name is the default `ROW`, or the name passed in through the `XMLFormat` formatting object. This functionality can be used to turn XML fragments into well-formed documents. Consider this XML data:

```
CREATE TABLE po_xml_tab(podoc XMLType);
Table created.
INSERT INTO po_xml_tab VALUES(XMLType('<DOCUMENT>
                                        <EMPLOYEE>
```

```
    <ENAME>John</ENAME>
    <EMPNO>200</EMPNO>
  </EMPLOYEE>
  <EMPLOYEE>
    <ENAME>Jack</ENAME>
    <EMPNO>400</EMPNO>
  </EMPLOYEE>
  <EMPLOYEE>
    <ENAME>Joseph</ENAME>
    <EMPNO>300</EMPNO>
  </EMPLOYEE>
</DOCUMENT>'));
1 row created.
COMMIT;
```

This query extracts `ENAME` elements:

```
SELECT e.podoc.extract('/DOCUMENT/EMPLOYEE/ENAME') FROM po_xml_tab e;
```

The query result is an XML document fragment:

```
<ENAME>John</ENAME>
<ENAME>Jack</ENAME>
<ENAME>Joseph</ENAME>
```

You can make such a fragment into a valid XML document by calling `sys_XMLGen` to wrap a root element around the fragment, as follows:

```
SELECT sys_XMLGen(e.podoc.extract('/DOCUMENT/EMPLOYEE/ENAME')).getClobVal()
  FROM po_xml_tab e;
```

This places a `ROW` element around the fragment, as follows:

```
<?xml version="1.0"?>
<ROW>
  <ENAME>John</ENAME>
  <ENAME>Jack</ENAME>
  <ENAME>Joseph</ENAME>
</ROW>
```

> **Note:** If the input to `sys_XMLGen` is a column, then the column name is used as the default element name. You can override the element name using the `XMLFormat` formatting object as a second argument to `sys_XMLGen`. See "Using XMLFormat Object Type" on page 16-50.

Example 16–40 Using SYS_XMLGEN with Object Views

For any undefined entities here, refer to the code in Example 16–28 on page 16-37.

```
-- Create purchase order object type
CREATE OR REPLACE TYPE po_typ AUTHID CURRENT_USER
  AS OBJECT(pono            NUMBER,
            customer        customer_typ,
            orderdate       DATE,
            shipdate        TIMESTAMP,
            lineitems_ntab  lineitems_ntabtyp,
            shiptoaddr      address_typ)
/
--Purchase order view
```

```
CREATE OR REPLACE VIEW po OF po_typ
  WITH OBJECT IDENTIFIER (PONO)
  AS SELECT p.pono, customer_typ(p.custno, c.custname, c.address, c.phonelist),
            p.orderdate, p.shipdate,
            CAST(MULTISET(
                   SELECT
                     lineitem_typ(l.lineitemno,
                                  stockitem_typ(l.stockno, s.price, s.taxrate),
                                                l.quantity, l.discount)
                     FROM lineitems_tab l, stock_tab s
                     WHERE l.pono = p.pono AND s.stockno=l.stockno)
                 AS lineitems_ntabtyp),
            address_typ(p.tostreet, p.tocity, p.tostate, p.tozip)
       FROM po_tab p, customer c
       WHERE p.custno=c.custno;

-- Use sys_XMLGen to generate PO in XML format
SELECT  sys_XMLGen(OBJECT_VALUE,
                   XMLFormat.createFormat('PurchaseOrder')).getClobVal() PO
  FROM po p
  WHERE p.pono=1001;
```

The query returns the purchase order in XML format:

```
PO
--------------------------------------------
<?xml version="1.0"?>
<PurchaseOrder>
 <PONO>1001</PONO>
 <CUSTOMER>
  <CUSTNO>1</CUSTNO>
  <CUSTNAME>Jean Nance</CUSTNAME>
  <ADDRESS>
   <STREET>2 Avocet Drive</STREET>
   <CITY>Redwood Shores</CITY>
   <STATE>CA</STATE>
   <ZIP>95054</ZIP>
  </ADDRESS>
  <PHONELIST>
   <VARCHAR2>415-555-1212</VARCHAR2>
  </PHONELIST>
 </CUSTOMER>
 <ORDERDATE>10-APR-97</ORDERDATE>
 <SHIPDATE>10-MAY-97 12.00.00.000000 AM</SHIPDATE>
 <LINEITEMS_NTAB>
  <LINEITEM_TYP LineItemNo="1">
   <ITEM StockNo="1534">
    <PRICE>2234</PRICE>
    <TAXRATE>2</TAXRATE>
   </ITEM>
   <QUANTITY>12</QUANTITY>
   <DISCOUNT>0</DISCOUNT>
  </LINEITEM_TYP>
  <LINEITEM_TYP LineItemNo="2">
   <ITEM StockNo="1535">
    <PRICE>3456.23</PRICE>
    <TAXRATE>2</TAXRATE>
   </ITEM>
   <QUANTITY>10</QUANTITY>
   <DISCOUNT>10</DISCOUNT>
  </LINEITEM_TYP>
```

```
 </LINEITEMS_NTAB>
 <SHIPTOADDR/>
</PurchaseOrder>

1 row selected.
```

Generating XML Using SQL Function SYS_XMLAGG

SQL function `sys_XMLAgg` 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 16–16 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 a `<xsql:include-xml>` element should return a *single row* containing a *single column*. The column can be either a `CLOB` instance or a `VARCHAR2` value. It must contain a well-formed XML document. The XML document is parsed and included in your XSQL page.

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide* for information on element `<xsql:include-xml>` and XSQL pages

Example 16–41 Using XSQL Servlet <xsql:include-xml> with Nested XMLAgg Functions

This example uses nested calls to function `XMLAgg` to aggregate the results of a dynamically-constructed XML document containing departments and their employees into a single XML result document, which is wrapped in a `DepartmentList` element. The call to method `getClobVal()` provides XSQL Servlet with a `CLOB` value instead of an `XMLType` instance. To display the results, XSQL Servlet needs a special environment, such as the XSQL Command-Line Utility, XSQL Servlet installed in a Web server, Java Server Pages (JSP), or a Java XSQLRequest object.

```
<xsql:include-xml connection="orcl92" xmlns:xsql="urn:oracle-xsql">
  SELECT
    XMLElement("DepartmentList",
               XMLAgg(XMLElement(
                        "Department",
                        XMLAttributes(department_id as "Id"),
                        XMLForest(department_name as "Name"),
                        (SELECT XMLElement("Employees",
                                           XMLAgg(XMLElement(
                                                    "Employee",
                                                    XMLAttributes(
                                                      employee_id as "Id"),
                                                    XMLForest(
```

```
                                                    last_name as "Name",
                                                    salary as "Salary",
                                                    job_id as "Job"))))
                            FROM employees e
                            WHERE e.department_id=d.department_id)))).getClobVal()
    FROM departments d
    ORDER BY department_name
</xsql:include-xml>
```

The query itself produces the following result:

```
XMLELEMENT("DEPARTMENTLIST",XMLAGG(XMLELEMENT("DEPARTMENT",XMLATTRIBUTES(DEPARTM
--------------------------------------------------------------------------------
<DepartmentList><Department Id="10"><Name>Administration</Name><Employees><Emplo
yee Id="200"><Name>Whalen</Name><Salary>4400</Salary><Job>AD_ASST</Job></Employe
e></Employees></Department><Department Id="20"><Name>Marketing</Name><Employees>
<Employee Id="201"><Name>Hartstein</Name><Salary>13000</Salary><Job>MK_MAN</Job>
</Employee><Employee Id="202"><Name>Fay</Name><Salary>6000</Salary><Job>MK_REP</
Job></Employee></Employees></Department>
...
</DepartmentList>

1 row selected.
```

Example 16–42 Using XSQL Servlet <xsql:include-xml> with XMLElement and XMLAgg

It is more efficient for the database to aggregate XML fragments into a single result document. Element `<xsql:include-xml>` encourages this approach by retrieving only the first row from the query you provide.

You can use the built-in Oracle Database XPath query features to extract an aggregate list of all purchase orders of the film *Grand Illusion*. This example uses the `purchaseorder` table in sample schema `OE`.

```
CONNECT OE/OE;
Connected.

SELECT
  XMLElement(
    "GrandIllusionOrders",
    XMLAgg(extract(OBJECT_VALUE,
                   '/PurchaseOrder/LineItems/*[Part[@Id="37429121924"]]')))
  FROM purchaseorder;
```

This produces the following result (the actual result is not pretty-printed):

```
XMLELEMENT("GRANDILLUSIONORDERS",XMLAGG(EXTRACT(OBJECT_VALUE,'/PURCHASEORDER/LIN
--------------------------------------------------------------------------------
<GrandIllusionOrders>
  <LineItem ItemNumber="14">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="14">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="6">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="19">
```

```
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
  <LineItem ItemNumber="21">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="15">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="8">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="1"/>
  </LineItem>
  <LineItem ItemNumber="17">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
</GrandIllusionOrders>

1 row selected.
```

To include this `XMLType` query result in your XSQL page, paste the query inside an `<xsql:include-xml>` element, and call method `getClobVal()`, so that the result is returned to the client as a `CLOB` value instead of as an `XMLType` instance:

```
<xsql:include-xml connection="orcl92" xmlns:xsql="urn:oracle-xsql">
  SELECT
    XMLElement(
      "GrandIllusionOrders",
      XMLAgg(
        extract(
          OBJECT_VALUE,
          '/PurchaseOrder/LineItems/*[Part[@Id="37429121924"]]'))).getClobval()
    FROM purchaseorder;
</xsql:include-xml>
```

SQL functions `XMLElement` and `XMLAgg` are used together here to aggregate all of the XML fragments identified by the query into a single, well-formed XML document. Failing to do this results in an attempt by the XSQL page processor to parse a `CLOB` value that looks like this:

```
<LineItem>...</LineItem>
<LineItem>...</LineItem>
...
```

This is not well-formed XML because it does not have a single root element as required by the XML 1.0 recommendation. Functions `XMLElement` and `XMLAgg` work together to produce a well-formed result with single root element `GrandIllusionOrders`. This well-formed XML is then parsed and included in your XSQL page.

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide*, the chapter, 'XSQL Page Publishing Framework'

Using XSLT and XSQL

With XSQL Pages, you have control over where XSLT is executed: in the database, the middle-tier, or the client. For database execution, use SQL function `XMLtransform` (or the equivalent) in your query. For middle-tier execution, add `<?xml-stylesheet?>` at the top of your template page. For client execution, add attribute `client="yes"` to PI `<?xml-stylesheet?>`.

With XSQL Pages, you can build pages that conditionally off-load style-sheet processing to the client, depending, for example, on what browser is used.

To improve performance and throughput, XSQL caches and pools XSLT style sheets (as well as database connections) in the middle tier. 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.

XSQL Pages can include a mix of static XML and dynamically produced XML. You can take advantage of this by using the database to create only the dynamic part of the page.

Generating XML Using XML SQL Utility (XSU)

Oracle XML SQL Utility (XSU) can be used with Oracle Database to generate XML. You can use XSU l to generate XML on either the middle tier or the client. XSU also supports generating XML on tables with `XMLType` columns.

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide* for 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 16–43 Using XMLAGG ORDER BY Clause

Consider this table:

```
CREATE TABLE dev_tab (dev         NUMBER,
                      dev_total   NUMBER,
                      devname     VARCHAR2(20));
Table created.
INSERT INTO dev_tab VALUES (16, 5,  'Alexis');
1 row created.
INSERT INTO dev_tab VALUES (2,  14, 'Han');
1 row created.
INSERT INTO dev_tab VALUES (1,  2,  'Jess');
1 row created.
INSERT INTO dev_tab VALUES (9,  88, 'Kurt');
1 row created.
COMMIT;
```

In this example, the result is aggregated according to the order of the `dev` column (the actual result is not pretty-printed):

```
SELECT XMLAgg(XMLElement("Dev",
                         XMLAttributes(dev AS "id", dev_total AS "total"),
                         devname)
              ORDER BY dev)
  FROM tab1 dev_total;

XMLAGG(XMLELEMENT("DEV",XMLATTRIBUTES(DEVAS"ID",DEV_TOTALAS"TOTAL"),DEVNAME)ORDE
--------------------------------------------------------------------------------
<Dev id="1" total="2">Jess</Dev>
<Dev id="2" total="14">Han</Dev>
<Dev id="9" total="88">Kurt</Dev>
<Dev id="16" total="5">Alexis</Dev>

1 row selected.
```

Using XMLSEQUENCE, EXTRACT, and TABLE to Return a Rowset

To use SQL function `XMLSequence` with `extract` to return a rowset with relevant portions of a document extracted as multiple rows, use `XMLSequence` as an argument to function `TABLE`, as shown in Example 16–44.

Example 16–44 Returning a Rowset using XMLSEQUENCE, EXTRACT, and TABLE

This example uses the `purchaseorder` table in sample schema `OE`.

```
CONNECT OE/OE;
Connected.

SELECT extractValue(value(item), '/*/Description') AS descr,
       extractValue(value(item), '/*/Part/@Id') AS partid
  FROM purchaseorder p,
       table(XMLSequence(extract(value(p),
                                 '/PurchaseOrder/LineItems/LineItem'))) item
  WHERE extractValue(value(item), '/*/Part/@Id') = '715515012027'
     OR extractValue(value(item), '/*/Part/@Id') = '715515011921'
  ORDER BY partid;
```

This returns a rowset with just the descriptions and part IDs, ordered by part ID.

```
DESCR
--------------
PARTID
--------------
My Man Godfrey
715515011921

My Man Godfrey
715515011921

My Man Godfrey
715515011921

My Man Godfrey
715515011921

My Man Godfrey
715515011921
```

```
My Man Godfrey
715515011921

My Man Godfrey
715515011921

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

16 rows selected.
```

17

Using XQuery with Oracle XML DB

This chapter describes how to use the XQuery language with Oracle XML DB. It covers Oracle XML DB support for the language, including SQL functions `XMLQuery` and `XMLTable` and the SQL*Plus `XQUERY` command.

This chapter contains these topics:

Overview of XQuery in Oracle XML DB

> **Note:** At the time of release of Oracle Database 10g Release 2, the W3C XQuery working group had not yet published the XQuery recommendation. Oracle will continue to track the evolution of the XQuery standard, until such time as it becomes a recommendation. During this period, in order to follow the evolution of the XQuery standard, Oracle may release updates to the XQuery implementation which are not backwards compatible with previous releases or patch sets.

Oracle XML DB support for the XQuery language is provided through a native implementation of SQL/XML functions `XMLQuery` and `XMLTable`. As a convenience, SQL*Plus command `XQUERY` is also provided, which lets you enter XQuery expressions directly—in effect, this command turns SQL*Plus into an XQuery command-line interpreter.

Oracle XML DB generally evaluates XQuery expressions by *compiling* them into the same underlying structures as relational queries. Queries are optimized, leveraging both relational-database and XQuery-specific optimization technologies, so that Oracle XML DB serves as a native XQuery engine.

There are a few XQuery expressions that cannot be rewritten to relational expressions. To provide you the full power of XQuery, Oracle XML DB evaluates these XQuery expressions using a functional XQuery interpreter, or evaluation engine (which itself has been compiled into the database). The treatment of *all* XQuery expressions, whether natively compiled or evaluated functionally, is *transparent* to you: you will never need to change your code in any way to take advantage of available XQuery optimizations.

See Also:

- SQL Functions XMLQuery and XMLTable and SQL*Plus XQUERY Command
- Oracle XQuery Extension Functions for Oracle-specific XQuery functions that extend the language
- Oracle XML DB Support for XQuery for details on Oracle XML DB support for XQuery

Overview of the XQuery Language

Oracle XML DB supports the latest version of the XQuery language specifications. This section presents a brief overview of the language. For more information, consult a recent book on the language or refer to the standards documents that define it, which are available at `http://www.3.org`.

Functional Language Based on Sequences

XQuery 1.0 is the W3C language designed for querying XML data. It is similar to SQL in many ways, but just as SQL is designed for querying structured, relational data, XQuery is designed especially for querying semistructured, XML data from a variety of data sources. You can use XQuery to query XML data wherever it is found, whether it is stored in database tables, available through Web Services, or otherwise created on the fly. In addition to querying XML data, XQuery can be used to *construct* XML data. In this regard, XQuery can serve as an alternative or a complement to both XSLT and the other SQL/XML publishing functions, such as `XMLElement`.

XQuery builds on the Post-Schema-Validation Infoset (PSVI) data model, which unites the XML Information Set (Infoset) data model and the XML Schema type system. XQuery defines a new data model based on *sequences*: the result of *each* XQuery expression is a sequence. XQuery is all about manipulating sequences. This makes XQuery similar to a set-manipulation language, except that sequences are ordered and can contain duplicate items. XQuery sequences differ from the sequences in some other languages in that nested XQuery sequences are always *flattened* in their effect.

In many cases, sequences can be treated as unordered, to maximize optimization – where this is available, it is under your control. This **unordered mode** can be applied to join order in the treatment of nested iterations (`for`), and it can be applied to the treatment of XPath expressions (for example, in `/a/b`, the matching `b` elements can be processed without regard to document order).

An XQuery **sequence** consists of zero or more **items**, which can be either *atomic* (scalar) values or XML *nodes*. Items are typed using a rich type system that is based

upon the types of XML Schema. This type system is a major change from that of XPath 1.0, which is limited to simple scalar types such as Boolean, number, and string.

XQuery is a *functional* language. As such, it consists of a set of possible *expressions* that are *evaluated* and return *values* (which, in the case of XQuery, are sequences). As a functional language, XQuery is also **referentially transparent**, generally: the *same expression* evaluated in the *same context* returns the *same value*.

Exceptions to this desirable mathematical property include the following:

- XQuery expressions that derive their value from interaction with the external environment. For example, an expression such as `fn:current-time(...)` or `fn:doc(...)` does not necessarily always return the same value, since it depends on external conditions that can change (the time changes; the content of the target document might change).

  In some cases, like that of `fn:doc`, XQuery is defined to be referentially transparent within the execution of a single query: within a query, each invocation of `fn:doc` with the same argument results in the same document.

- XQuery expressions that are defined to be dependent on the particular XQuery language implementation. The result of evaluating such expressions might vary between implementations. Function `fn:doc` is an example of a function that is essentially implementation-defined.

Referential transparency applies also to XQuery *variables*: the same variable in the same context has the same value. Functional languages are like mathematics formalisms in this respect and unlike procedural, or imperative, programming languages. A variable in a procedural language is really a name for a memory location; it has a *current* value, or state, as represented by its content at any time. A variable in a declarative language such as XQuery is really a name for a *static* value.

XQuery Expressions

XQuery expressions are case-sensitive. The expressions include the following:

- **primary expression –** literal, variable, or function application. A variable name starts with a dollar-sign (`$`) – for example, `$foo`. Literals include numerals, strings, and character or entity references.
- **XPath expression –** Any XPath expression. The developing XPath 2.0 standard will be a subset of XQuery. XPath 1.0 is currently a subset, although XQuery uses a richer type system.
- **FLWOR expression –** The most important XQuery expression, composed of the following, in order, from which FLWOR takes its name: `for`, `let`, `where` , `order by`, `return`.
- **XQuery sequence** – The comma (`,`) constructor creates sequences. Sequence-manipulating functions such as `union` and `intersect` are also available. All XQuery sequences are effectively **flat**: a nested sequence is treated as its flattened equivalent. Thus, for instance, `(1, 2, (3, 4, (5), 6), 7)` is treated as `(1, 2, 3, 4, 5, 6, 7)`. A singleton sequence, such as `(42)`, acts the same in most XQuery contexts as does its single item, `42`. Remember that the result of any XQuery expression is a sequence.
- **Direct (literal) constructions** – XML element and attribute syntax automatically constructs elements and attributes: what you see is what you get. For example, the XQuery expression `<a>33</a>` constructs the XML element `<a>33</a>`.

- **Computed (dynamic) constructions –** You can construct XML data at runtime using computed values. For example, the following XQuery expression constructs this XML data: `<foo toto="5"><bar>tata titi</bar> why? </foo>`.

  ```
  <foo>{attribute toto {2+3}, element bar {"tata", "titi"}, text {" why? "}</foo>
  ```

  In this example, element `foo` is a direct construction; the other constructions are computed. In practice, the arguments to computed constructors are not literals (such as `toto` and `"tata"`), but expressions to be evaluated (such as `2+3`). Both the name and the value arguments of an element or attribute constructor can be computed. Braces (`{, }`) are used to mark off an XQuery expression to be evaluated.

- **Conditional expression –** As usual, but remember that each part of the expression is itself an arbitrary expression. For instance, in this conditional expression, each of these subexpressions can be any XQuery expression: `something`, `somethingElse`, `expression1`, and `expression2`.

  ```
  if (something < somethingElse) then expression1 else expression2
  ```

- **Arithmetic, relational expression –** As usual, but remember that each relational expression returns a (Boolean[1]) value. Examples:

  ```
  2 + 3
  42 < $a + 5
  (1, 4) = (1, 2)
  5 > 3 eq true()
  ```

- **Quantifier expression –** Universal (`every`) and existential (`some`) quantifier functions provide shortcuts to using a FLWOR expression in some cases. Examples:

  ```
  every $foo in doc("bar.xml")//Whatever satisfies $foo/@bar > 42
  some $toto in (42, 5), $titi in ("xyz12", "abc", 5) satisfies $toto = $titi
  ```

- **Regular expression –** XQuery regexes are based on XML Schema 1.0 and Perl. (See Support for XQuery Functions and Operators on page 17-35.)

- **Type expression –** An XQuery expression that represents an XQuery type. Examples: `item()`, `node()`, `attribute()`, `element()`, `document-node()`, `namespace()`, `text()`, `xs:integer`, `xs:string`.[2]

  Type expressions can have **occurrence indicators**: **`?`** (optional: zero or one), **`*`** (zero or more), **`+`** (one or more). Examples: `document-node(element())*`, `item()+`, `attribute()?`.

  XQuery also provides operators for working with types. These include `cast as`, `castable as`, `treat as`, `instance of`, `typeswitch`, and `validate`. For example, `"42" cast as xs:integer` is an expression whose value is the integer 2. (It is not, strictly speaking, a type expression, because its value does not represent a type.)

FLWOR Expressions

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

[1] The value returned is in fact a sequence, as always. However, in XQuery, a sequence of one item is equivalent to that item itself. In this case, the single item is a Boolean value.

[2] Namespace prefix `xs` is predefined for the XML Schema namespace, `http://www.w3.org/2001/XMLSchema`.

FLWOR is the most general expression syntax in XQuery. FLWOR (pronounced "flower") stands for `for`, `let`, `where`, `order by`, and `return`. A FLWOR expression has at least one `for` or `let` clause and a `return` clause; single `where` and `order by` clauses are optional.

- **`for`** – Bind one or more variables each to any number of values, in turn. That is, for each variable, iterate, binding the variable to a different value for each iteration.

  At each iteration, the variables are bound in the order they appear, so that the value of a variable `$earlier` that is listed before a variable `$later` in the `for` list, can be used in the binding of variable `$later`. For example, during its second iteration, this expression binds `$i` to `4` and `$j` to `6` (2+4):

  ```
  for $i in (3, 4), $j in ($i, 2+$i)
  ```

- **`let`** – Bind one or more variables.

  Just as with `for`, a variable can be bound by `let` to a value computed using another variable that is listed previously in the binding list of the `let` (or an enclosing `for` or `let`). For example, this expression binds `$j` to `5` (3+2):

  ```
  let $i := 3, $j := $i + 2
  ```

- **`where`** – Filter the `for` and `let` variable bindings according to some condition. This is similar to a SQL `WHERE` clause.
- **`order by`** – Sort the result of `where` filtering.
- **`return`** – Construct a result from the ordered, filtered values. This is the result of the FLWOR expression as a whole. It is a flattened sequence.

Expressions `for` and `let` function similarly to a SQL `FROM` clause; `where` acts like a SQL `WHERE` clause; `order by` is similar to `ORDER BY` in SQL; and `return` is like `SELECT` in SQL. In other words, except for the two keywords whose names are the same in both languages (`where`, `order by`), FLWOR clause order is more or less opposite to the SQL clause order, but the meanings of the corresponding clauses are quite similar.

Note that using a FLWOR expression (with `order by`) is the *only* way to construct a sequence in any order other than document order.

SQL Functions XMLQuery and XMLTable

SQL functions `XMLQuery` and `XMLTable` are defined by the SQL/XML standard as a general interface between the SQL and XQuery languages. As is the case for the other SQL/XML functions, `XMLQuery` and `XMLTable` let you take advantage of the power and flexibility of both SQL and XML. Using these functions, you can construct XML data using relational data, query relational data as if it were XML, and construct relational data from XML data.

The SQL/XML standard is ISO/IEC 9075–14:2005(E), Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML). As part of the SQL standard, it is aligned with SQL:2003. It is being developed under the auspices of these two standards bodies:

- ISO/IEC JTC1/SC32 ("International Organization for Standardization and International Electrotechnical Committee Joint Technical Committee 1, Information technology, Subcommittee 32, Data Management and Interchange").

- INCITS Technical Committee H2 ("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. Committee H2 is the committee responsible for SQL and SQL/MM.

This SQL/XML standardization process is ongoing. Please refer to `http://www.sqlx.org` for the latest information about `XMLQuery` and `XMLTable`.

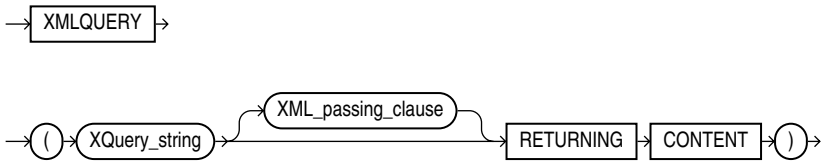
See Also:

- `http://www.sqlx.org` for information on SQL functions `XMLQuery` and `XMLTable`
- `http://www.w3.org` for information on the XQuery language
- "Generating XML Using SQL Functions" on page 16-3 for information on using other SQL/XML functions with Oracle XML DB

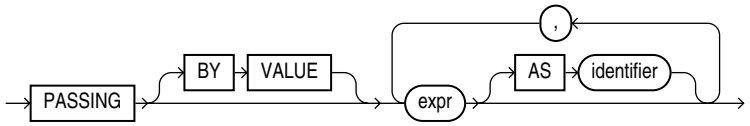
XMLQUERY SQL Function in Oracle XML DB

You use SQL function `XMLQuery` to construct or query XML data. This function takes as arguments an *XQuery expression*, as a string literal, and an optional XQuery *context item*, as a SQL expression. The context item establishes the XPath context in which the XQuery expression is evaluated. Additionally, `XMLQuery` accepts as arguments any number of SQL expressions whose values are bound to XQuery variables during the XQuery expression evaluation. The function returns the result of evaluating the XQuery expression, as an `XMLType` instance.

Figure 17–1 XMLQUERY Syntax



XML_passing_clause::=



- *`XQuery_string`* is a complete XQuery expression, possibly including a prolog, as a literal string.
- The *`XML_passing_clause`* is the keyword `PASSING` followed by one or more SQL expressions (*`expr`*) that each return an `XMLType` instance. All but possibly one of the expressions must each be followed by the keyword `AS` and an XQuery *`identifier`*. The result of evaluating each *`expr`* is bound to the corresponding *`identifier`* for the evaluation of *`XQuery_string`*. If there is an *`expr`* that is not followed by an `AS` clause, then the result of evaluating that *`expr`* is used as the *context* item for evaluating *`XQuery_string`*. Oracle XML DB supports only passing `BY VALUE`, not passing `BY REFERENCE`, so the clause `BY VALUE` is implicit and can be omitted.

- `RETURNING CONTENT` indicates that the value returned by an application of `XMLQuery` is an instance of parameterized XML type `XML(CONTENT)`, not parameterized type `XML(SEQUENCE)`. It is a document fragment that conforms to the *extended* Infoset data model. As such, it is a single document node with any number of children. The children can each be of any XML node type; in particular, they can be text nodes.

  Oracle XML DB supports only the `RETURNING CONTENT` clause of SQL/XML function `XMLQuery`; it does *not* support the `RETURNING SEQUENCE` clause.

You can pass an `XMLType` column, table, or view as the context-item argument to function `XMLQuery`—see, for example, Example 17–8. To query a relational table or view as if it were XML, without having to first create a SQL/XML view on top of it, use XQuery function `ora:view` within an XQuery expression—see, for example, Example 17–6.

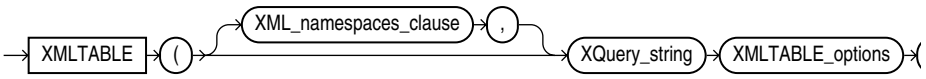
See Also:

- `http://www.sqlx.org` for information on the definition of SQL function `XMLQuery`
- *Oracle Database SQL Reference* for reference information on SQL function `XMLQuery` in Oracle Database
- "ora:view XQuery Function" on page 17-11

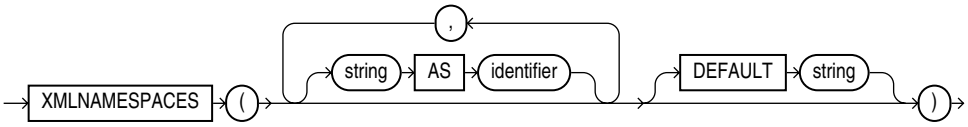
XMLTABLE SQL Function in Oracle XML DB

You use SQL function `XMLTable` to *shred* the result of an XQuery-expression evaluation into the relational rows and columns of a new, virtual table. You can then insert the virtual table into a pre-existing database table, or you can query it using SQL—in a join expression, for example (see Example 17–9). You use `XMLTable` in a SQL `FROM` clause.

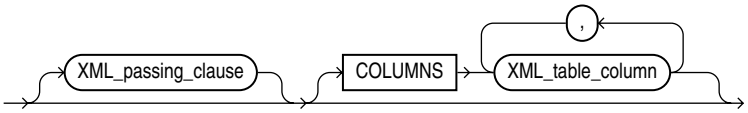
Figure 17–2 XMLTABLE Syntax



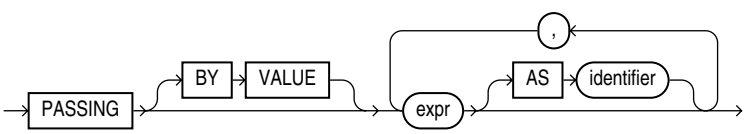
XML_namespaces_clause::=



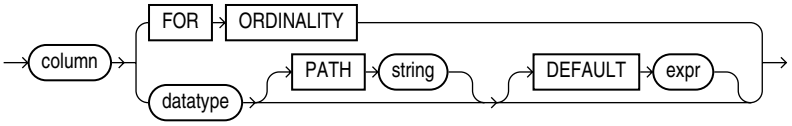
XMLTABLE_options::=



XML_passing_clause::=



XML_table_column::=



- *XQuery_string* is a complete XQuery expression, possibly including a prolog, as a literal string. The value of the expression serves as input to the XMLTable function; it is this XQuery result that is shredded into relational data.
- The optional XMLNAMESPACES clause contains XML namespace declarations that are referenced by *XQuery_string* and by the XPath expression in the PATH clause of *XML_table_column*.
- The *XML_passing_clause* is the keyword PASSING followed by one or more SQL expressions (*expr*) that each return an XMLType instance. All but possibly one of the expressions must each be followed by the keyword AS and an XQuery *identifier*. The result of evaluating each *expr* is bound to the corresponding *identifier* for the evaluation of *XQuery_string*. If there is an *expr* that is not followed by an AS clause, then the result of evaluating that *expr* is used as the *context* item for evaluating *XQuery_string*. Oracle XML DB supports only passing BY VALUE, not passing BY REFERENCE, so the clause BY VALUE is implicit and can be omitted.
- The optional COLUMNS clause defines the columns of the virtual table to be created by XMLTable.
  - If you omit the COLUMNS clause, then XMLTable returns a row with a single XMLType pseudo-column, named COLUMN_VALUE.
  - FOR ORDINALITY specifies that *column* is to be a column of generated row numbers (SQL datatype NUMBER). There must be at most one FOR ORDINALITY clause.
  - You must specify the *datatype* of each resulting *column* except the FOR ORDINALITY column.
  - The optional PATH clause specifies that the portion of the XQuery result that is addressed by XPath expression *string* is to be used as the *column* content. You can use multiple PATH clauses to split the XQuery result into different virtual-table columns.

    If you omit PATH, then the XPath expression *column* is assumed. For example, these two expressions are equivalent:

    ```
    XMLTable(... COLUMNS foo)
    XMLTable(... COLUMNS foo PATH 'FOO')
    ```

  - The optional DEFAULT clause specifies the value to use when the PATH expression results in an empty sequence (or NULL). Its *expr* is an XQuery expression that is evaluated to produce the default value.

    See Also:

    - http://www.sqlx.org for information on the definition of SQL function XMLTable
    - *Oracle Database SQL Reference* for reference information on SQL function XMLTable in Oracle Database

Predefined Namespaces and Prefixes

The following namespaces and prefixes are predefined for use with XQuery in Oracle XML DB:

Table 17–1 Predefined Namespaces and Prefixes

Prefix	Namespace	Description
`ora`	`http://xmlns.oracle.com/xdb`	Oracle XML DB namespace
`local`	`http://www.w3.org/2003/11/xpath-local-functions`	XPath local function declaration namespace
`fn`	`http://www.w3.org/2003/11/xpath-functions`	XPath function namespace
`xdt`	`http://www.w3.org/2003/11/xpath-datatypes`	XPath datatype namespace
`xml`	`http://www.w3.org/XML/1998/namespace`	XML namespace
`xs`	`http://www.w3.org/2001/XMLSchema`	XML Schema namespace
`xsi`	`http://www.w3.org/2001/XMLSchema-instance`	XML Schema instance namespace

You can use these prefixes in XQuery expressions without first declaring them in the XQuery-expression prolog. You can redefine any of them *except* `xml` in the prolog. All of these prefixes except `ora` are predefined in the XQuery standard.

Oracle XQuery Extension Functions

Oracle XML DB adds some XQuery functions to those provided in the W3C standard. These additional functions are in the Oracle XML DB namespace, `http://xmlns.oracle.com/xdb`, which uses the predefined prefix **`ora`**. This section describes these Oracle extension functions.

ora:contains XQuery Function

Syntax

```
ora:contains (input_text, text_query [, policy_name] [, policy_owner])
```

XPath function `ora:contains` can be used in an XPath expression inside an XQuery expression or in a call to SQL function `existsNode`, `extract`, or `extractValue`. It is used to restrict a structural search with a full-text predicate. Function `ora:contains` returns a positive integer when the *`input_text`* matches *`text_query`* (the higher the number, the more relevant the match), and zero otherwise. When used in an XQuery expression, the XQuery return type is `xs:integer()`; when used in an XPath expression outside of an XQuery expression, the XPath return type is `number`.

Argument *`input_text`* must evaluate to a single text node or an attribute. The syntax and semantics of *`text_query`* in `ora:contains` are the same as *`text_query`* in `contains`, with a few restrictions.

See Also: "ora:contains XPath Function" on page 10-18

ora:matches XQuery Function

Syntax

```
ora:matches (target_string, match_pattern [, match_parameter])
```

XQuery function `ora:match` lets you use a regular expression to match text in a string. It returns `true()` if its *`target_string`* argument matches its regular-expression *`match_pattern`* argument and `false()` otherwise. If *`target_string`* is the empty sequence, `false()` is returned. Optional argument *`match_parameter`* is a code that qualifies matching: case-sensitivity and so on.

The behavior of XQuery function `ora:matches` is the same as that of SQL condition `REGEXP_LIKE`, but the types of its arguments are XQuery types instead of SQL datatypes. The argument types are as follows:

- *`target_string`* – `xs:string?`[3]
- *`match_pattern`* – `xs:string`
- *`match_parameter`* – `xs:string`

> **See Also:** *Oracle Database SQL Reference* for information on SQL condition `REGEXP_LIKE`

ora:replace XQuery Function

Syntax

```
ora:replace (target_string, match_pattern, replace_string [, match_parameter])
```

XQuery function `ora:replace` lets you use a regular expression to replace matching text in a string. *Each* occurrence in *`target_string`* that matches regular-expression *`match_pattern`* is replaced by *`replace_string`*. It returns the new string that results from the replacement. If *`target_string`* is the empty sequence, then the empty string (`""`) is returned. Optional argument *`match_parameter`* is a code that qualifies matching: case-sensitivity and so on.

The behavior of XQuery function `ora:matches` is the same as that of SQL function `regexp_replace`, but the types of its arguments are XQuery types instead of SQL datatypes. The argument types are as follows:

- *`target_string`* – `xs:string?`[4]
- *`match_pattern`* – `xs:string`
- *`replace_string`* – `xs:string`
- *`match_parameter`* – `xs:string`

In addition, `ora:replace` requires argument *`replace_string`* (it is optional in `regexp_replace`) and it does not use arguments for position and number of occurences – search starts with the first character and all occurrences are replaced.

> **See Also:** *Oracle Database SQL Reference* for information on SQL function `regexp_replace`

[3] The question mark (`?`) here is a zero-or-one occurrence indicator that indicates that the argument can be the empty sequence. See "XQuery Expressions" on page 17-3.

[4] The question mark (`?`) here is a zero-or-one occurrence indicator that indicates that the argument can be the empty sequence. See "XQuery Expressions" on page 17-3.

ora:sqrt XQuery Function

Syntax

```
ora:sqrt (number)
```

XQuery function `ora:sqrt` returns the square root of its numerical argument, which can be of XQuery type `xs:decimal`, `xs:float`, or `xs:double`. The returned value is of the same XQuery type as the argument.

ora:view XQuery Function

Syntax

```
ora:view ([db-schema STRING,] db-table STRING)
RETURNS document-node(element())*
```
[5]

XQuery function `ora:view` lets you query existing database tables or views inside an XQuery expression, as if they were XML documents. In effect, `ora:view` creates XML views over the relational data, on the fly. You can thus use `ora:view` to avoid explicitly creating XML views on top of relational data.

The input parameters are as follows:

- *db-schema* – An optional string literal that names a database schema.
- *db-table* – A string literal naming a database table or view. If *db-schema* is present, then *db-table* is in database schema *db-schema*.

Function `ora:view` returns an unordered sequence of `document` nodes, one for each row of *db-table*. The SQL/XML standard is used to map each input row to the output XML document: relational column names become XML element names. Unless *db-table* is of type `XMLType`, the column elements derived from a given table row are wrapped together in a `ROW` element. In that case, the return type is, more precisely, `document-node(element(ROW))*`.

XMLQuery and XMLTable Examples

XQuery is a very general and expressive language, and SQL functions `XMLQuery` and `XMLTable` combine that power of expression and computation with the similar strengths of SQL. This section illustrates some of what you can do with these two SQL/XML functions.

You will typically use XQuery with Oracle XML DB in the following ways. The examples here are organized to reflect these different uses.

- Query XML data in Oracle XML DB Repository.

  See "Using XQuery to Query XML Data in Oracle XML DB Repository".

- Query a relational table or view as if it were XML data. To do this, you use Oracle XQuery function `ora:view` to create an XML view over the relational data, on the fly.

  See "Using ora:view to Query Relational Data in XQuery Expressions".

[5] The asterisk (*) here is a zero-or-more occurrence indicator that indicates that the argument can be a possibly empty sequence of document nodes of type element. See "XQuery Expressions" on page 17-3.

- Query `XMLType` relational data, possibly shredding the resulting XML into relational data using function `XMLTable`.

  See "Using XQuery with XMLType Data".

Example 17–1 Creating Resources for Examples

This example creates repository resources that are used in some of the other examples.

```
DECLARE
  res BOOLEAN;
  empsxmlstring VARCHAR2(300):=
    '<?xml version="1.0"?>
     <emps>
       <emp empno="1" deptno="10" ename="John" salary="21000"/>
       <emp empno="2" deptno="10" ename="Jack" salary="310000"/>
       <emp empno="3" deptno="20" ename="Jill" salary="100001"/>
     </emps>';
  empsxmlnsstring VARCHAR2(300):=
    '<?xml version="1.0"?>
     <emps xmlns="http://emp.com">
       <emp empno="1" deptno="10" ename="John" salary="21000"/>
       <emp empno="2" deptno="10" ename="Jack" salary="310000"/>
       <emp empno="3" deptno="20" ename="Jill" salary="100001"/>
     </emps>';
  deptsxmlstring VARCHAR2(300):=
    '<?xml version="1.0"?>
     <depts>
       <dept deptno="10" dname="Administration"/>
       <dept deptno="20" dname="Marketing"/>
       <dept deptno="30" dname="Purchasing"/>
     </depts>';
BEGIN
  res := DBMS_XDB.createResource('/public/emps.xml',   empsxmlstring);
  res := DBMS_XDB.createResource('/public/empsns.xml', empsxmlnsstring);
  res := DBMS_XDB.createResource('/public/depts.xml',  deptsxmlstring);
END;
/
```

XQuery Is About Sequences

It is important to keep in mind that XQuery is a general *sequence*-manipulation language. Its expressions and their results are not necessarily XML data. An XQuery sequence can contain items of any XQuery type, which includes numbers, strings, Boolean values, dates, as well as various types of XML node (`document-node()`, `element()`, `attribute()`, `text()`, `namespace()`, and so on). Example 17–2 provides a sampling.

Example 17–2 XMLQuery Applied to a Sequence of Items of Different Types

This example applies SQL function `XMLQuery` to an XQuery sequence that contains items of several different kinds:

- an integer literal: `1`
- a arithmetic expression: `2 + 3`
- a string literal: `"a"`
- a sequence of integers: `100 to 102`
- a constructed XML element node: `<A>33</A>`

This example also shows construction of a sequence using the comma operator (`,`) and parentheses (`(,)`) for grouping.

```
SELECT XMLQuery('(1, 2 + 3, "a", 100 to 102, <A>33</A>)'
                RETURNING CONTENT) AS output
  FROM DUAL;

OUTPUT
-------------------------
1 5 a 100 101 102<A>33</A>

1 row selected.
```

The sequence expression `100 to 102` evaluates to the sequence `(100, 101, 102)`, so the argument to `XMLQuery` is actually a sequence that contains a nested sequence. The sequence argument is automatically flattened, as is always the case for XQuery sequences. The actual argument is, in effect, `(1, 5, "a", 100, 101, 102, <A>33</A>)`.

Using XQuery to Query XML Data in Oracle XML DB Repository

This section presents examples of using XQuery with XML data in Oracle XML DB Repository. In Oracle XML DB, functions `fn:doc` and `fn:collection` return file and folder resources in the repository, respectively. Each example in this section uses XQuery function `fn:doc` to obtain a repository file that contains XML data, and then binds XQuery variables to parts of that data using `for` and `let` FLWOR-expression clauses.

> **See Also:** XQuery Functions doc and collection

Example 17–3 FLOWR Expression Using For, Let, Order By, Where, and Return

This example queries two XML-document resources in Oracle XML DB Repository: `/public/emps.xml` and `/public/depts.xml`. It illustrates the use of *each* of the possible FLWOR-expression clauses, as well as the use of `fn:doc`.

```
SELECT XMLQuery('for $e in doc("/public/emps.xml")/emps/emp
                 let $d :=
                   doc("/public/depts.xml")//dept[@deptno = $e/@deptno]/@dname
                 where $e/@salary > 100000
                 order by $e/@empno
                 return <emp ename="{$e/@ename}" dept="{$d}"/>'
                RETURNING CONTENT) FROM DUAL;

XMLQUERY('FOR$EINDOC("/PUBLIC/EMPS.XML")/EMPS/EMPLET$D:=DOC("/PUBLIC/DEPTS.XML")
--------------------------------------------------------------------------------
<emp ename="Jack" dept="Administration"></emp><emp ename="Jill" dept="Marketing"
></emp>

1 row selected.
```

In Example 17–3, the various FLWOR clauses perform these operations:

- **`for`** iterates over the `emp` elements in `/public/emps.xml`, binding variable `$e` to the value of each such element, in turn. That is, it iterates over a general list of employees, binding `$e` to each employee.
- **`let`** binds variable `$d` to a *sequence* consisting of all of the values of `dname` attributes of those `dept` elements in `/public/emps.xml` whose `deptno` attributes have the same value as the `deptno` attribute of element `$e` (this is a join

operation). That is, it binds `$d` to the names of all of the departments that have the same department number as the department of employee `$e`. (It so happens that the `dname` value is unique for each `deptno` value in `depts.xml`.) Note that, unlike `for`, `let` never iterates over values; `$d` is bound only once in this example.

- Together, `for` and `let` produce a stream of tuples (`$e`, `$d`), where `$e` represents an employee and `$d` represents the names of all of the departments to which that employee belongs—in this case, the unique name of the employee's unique department.
- **`where`** filters this tuple stream, keeping only tuples with employees whose salary is greater than 100,000.
- **`order by`** sorts the filtered tuple stream by employee number, `empno` (in ascending order, by default).
- **`return`** constructs `emp` elements, one for each tuple. Attributes `ename` and `dept` of these elements are constructed using attribute `ename` from the input and `$d`, respectively. Note that the element and attribute names `emp` and `ename` in the output have no necessary connection with the same names in the input document `emps.xml`.

Example 17–4 also uses each of the FLWOR-expression clauses. In addition, it demonstrates the use of other XQuery functions, besides `fn:doc`.

Example 17–4 FLOWR Expression Using Built-In Functions

This example shows the use of XQuery functions `doc`, `count`, `avg`, and `integer`, which are in the namespace for built-in XQuery functions, `http://www.w3.org/2003/11/xpath-functions`. This namespace is bound to the prefix `fn`.

```
SELECT XMLQuery('for $d in fn:doc("/public/depts.xml")/depts/dept/@deptno
                 let $e := fn:doc("/public/emps.xml")/emps/emp[@deptno = $d]
                 where fn:count($e) > 1
                 order by fn:avg($e/@salary) descending
                 return
                   <big-dept>{$d,
                              <headcount>{fn:count($e)}</headcount>,
                              <avgsal>{xs:integer(fn:avg($e/@salary))}</avgsal>}
                   </big-dept>'
               RETURNING CONTENT) FROM DUAL;

XMLQUERY('FOR$DINFN:DOC("/PUBLIC/DEPTS.XML")/DEPTS/DEPT/@DEPTNOLET$E:=FN:DOC("/P
--------------------------------------------------------------------------------
<big-dept>10<headcount>2</headcount><avgsal>165500</avgsal></big-dept>

1 row selected.
```

In Example 17–4, the various FLWOR clauses perform these operations:

- **`for`** iterates over `deptno` attributes in input document `/public/depts.xml`, binding variable `$d` to the value of each such attribute, in turn.
- **`let`** binds variable `$e` to a sequence consisting of all of the `emp` elements in input document `/public/emps.xml` whose `deptno` attributes have value `$d` (this is a join operation).
- Together, `for` and `let` produce a stream of tuples (`$d`, `$e`), where `$d` represents a department number and `$e` represents the set of employees in that department.
- **`where`** filters this tuple stream, keeping only tuples with more than one employee.

- **`order by`** sorts the filtered tuple stream by average salary in descending order. The average is computed by applying XQuery function `avg` (in namespace `fn`) to the values of attribute `salary`, which is attached to the `emp` elements of `$e`.
- **`return`** constructs `big-dept` elements, one for each tuple produced by `order by`. The `text()` node of `big-dept` contains the department number, bound to `$d`. A `headcount` child element contains the number of employees, bound to `$e`, as determined by XQuery function `count`. An `avgsal` child element contains the computed average salary.

Using ora:view to Query Relational Data in XQuery Expressions

This section presents examples of using Oracle XQuery function `ora:view` to query relational data as if it were XML data, from within an XQuery expression.

See Also: "ora:view XQuery Function" on page 17-11

Example 17–5 Using ora:view to Query Relational Tables as XML Views

This example uses Oracle XQuery function `ora:view` in a FLWOR expression to query two relational tables, `regions` and `countries` joining. Both tables belong to sample database schema `hr`.

```
SELECT XMLQuery('for $i in ora:view("REGIONS"), $j in  ora:view("COUNTRIES")
                 where $i/ROW/REGION_ID = $j/ROW/REGION_ID
                   and $i/ROW/REGION_NAME = "Asia"
                 return $j'
                RETURNING CONTENT) AS asian_countries
  FROM DUAL;
```

This produces the following result (the actual result is *not* pretty-printed).

```
ASIAN_COUNTRIES
--------------------------------------------------------------------------------
<ROW>
  <COUNTRY_ID>AU</COUNTRY_ID>
  <COUNTRY_NAME>Australia</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>CN</COUNTRY_ID>
  <COUNTRY_NAME>China</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>HK</COUNTRY_ID>
  <COUNTRY_NAME>HongKong</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>IN</COUNTRY_ID>
  <COUNTRY_NAME>India</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>JP</COUNTRY_ID>
  <COUNTRY_NAME>Japan</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
```

```
  <COUNTRY_ID>SG</COUNTRY_ID>
  <COUNTRY_NAME>Singapore</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>

1 row selected.
```

In Example 17–5, the various FLWOR clauses perform these operations:

- **for** iterates over sequences of XML elements returned by calls to `ora:view`. In the first call, each element corresponds to a row of relational table `regions` and is bound to variable `$i`. Similarly, in the second call to `ora:view`, `$j` is bound to successive rows of table `countries`. Since `regions` and `countries` are not `XMLType` tables, the top-level element corresponding to a row in each table is `ROW` (a wrapper element). Iteration over the row elements is unordered.
- **where** filters the rows from both tables, keeping only those pairs of rows whose `region_id` is the same for each table (it performs a join on `region_id`) and whose `region_name` is `Asia`.
- **return** returns the filtered rows from the `countries` table as an XML document containing XML fragments with `ROW` as their top-level element.

Example 17–6 uses `ora:view` within nested FLWOR expressions.

Example 17–6 Using ora:view in a Nested FLWOR Query

This query is an example of using nested FLWOR expressions. It accesses relational table `warehouses`, which is in sample database schema `oe`, and relational table `locations`, which is in sample database schema `hr`. To run this example as user `oe`, you must first connect as user `hr` and grant permission to user `oe` to perform `SELECT` operations on table `locations`. The two-argument form of `ora:view` is used here, to specify the database schema (first argument) in addition to the table (second argument).

```
CONNECT HR/HR
GRANT SELECT ON LOCATIONS TO OE
/
CONNECT OE/OE

SELECT XMLQuery(
         'for $i in ora:view("OE", "WAREHOUSES")/ROW
          return <Warehouse id="{$i/WAREHOUSE_ID}">
                 <Location>
                   {for $j in ora:view("HR", "LOCATIONS")/ROW
                    where $j/LOCATION_ID eq $i/LOCATION_ID
                    return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                 </Location>
                 </Warehouse>'
        RETURNING CONTENT) FROM DUAL;
```

This produces the following result (the actual result is *not* pretty-printed).

```
XMLQUERY('FOR$IINORA:VIEW("OE","WAREHOUSES")/ROWRETURN<WAREHOUSEID="{$I/WAREHOUS
--------------------------------------------------------------------------------
<Warehouse id="1">
  <Location>
    <STREET_ADDRESS>2014 Jabberwocky Rd</STREET_ADDRESS>
    <CITY>Southlake</CITY>
    <STATE_PROVINCE>Texas</STATE_PROVINCE>
  </Location>
```

```
</Warehouse>
<Warehouse id="2">
  <Location>
    <STREET_ADDRESS>2011 Interiors Blvd</STREET_ADDRESS>
    <CITY>South San Francisco</CITY>
    <STATE_PROVINCE>California</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="3">
  <Location>
    <STREET_ADDRESS>2007 Zagora St</STREET_ADDRESS>
    <CITY>South Brunswick</CITY>
    <STATE_PROVINCE>New Jersey</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="4">
  <Location>
    <STREET_ADDRESS>2004 Charade Rd</STREET_ADDRESS>
    <CITY>Seattle</CITY>
    <STATE_PROVINCE>Washington</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="5">
  <Location>
    <STREET_ADDRESS>147 Spadina Ave</STREET_ADDRESS>
    <CITY>Toronto</CITY>
    <STATE_PROVINCE>Ontario</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="6">
  <Location>
    <STREET_ADDRESS>12-98 Victoria Street</STREET_ADDRESS>
    <CITY>Sydney</CITY>
    <STATE_PROVINCE>New South Wales</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="7">
  <Location>
    <STREET_ADDRESS>Mariano Escobedo 9991</STREET_ADDRESS>
    <CITY>Mexico City</CITY>
    <STATE_PROVINCE>Distrito Federal,</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="8">
  <Location>
    <STREET_ADDRESS>40-5-12 Laogianggen</STREET_ADDRESS>
    <CITY>Beijing</CITY>
  </Location>
</Warehouse>
<Warehouse id="9">
  <Location>
    <STREET_ADDRESS>1298 Vileparle (E)</STREET_ADDRESS>
    <CITY>Bombay</CITY>
    <STATE_PROVINCE>Maharashtra</STATE_PROVINCE>
  </Location>
</Warehouse>

1 row selected.
```

In Example 17–6, the various FLWOR clauses perform these operations:

- The outer **for** iterates over the sequence of XML elements returned by `ora:view`: each element corresponds to a row of relational table `warehouses` and is bound to variable `$i`. Since `warehouses` is not an `XMLType` table, the top-level element corresponding to a row is `ROW`. The iteration over the row elements is unordered.
- The inner **for** iterates, similarly, over a sequence of XML elements returned by `ora:view`: each element corresponds to a row of relational table `locations` and is bound to variable `$j`.
- **where** filters the tuples (`$i`, `$j`), keeping only those whose `location_id` child is the same for `$i` and `$j` (it performs a join on `location_id`).
- The inner **return** constructs an XQuery sequence of elements `STREET_ADDRESS`, `CITY`, and `STATE_PROVINCE`, all of which are children of locations-table `ROW` element `$j`; that is, they are the values of the locations-table columns of the same name.
- The outer **return** wraps the result of the inner `return` in a `Location` element, and wraps that in a `Warehouse` element. It provides the `Warehouse` element with an `id` attribute whose value comes from the `warehouse_id` column of table `warehouses`.

See Also: Example 17–14 for the `EXPLAIN PLAN` of Example 17–6

Example 17–7 Using ora:view with XMLTable to Query a Relational Table as XML

In this example, SQL function `XMLTable` is used to shred the result of an XQuery query to virtual relational data. The XQuery expression used in this example is identical to the one used in Example 17–6; the result of evaluating the XQuery expression is a sequence of `Warehouse` elements. Function `XMLTable` produces a virtual relational table whose rows are those `Warehouse` elements. More precisely, the value of pseudocolumn `COLUMN_VALUE` for each virtual-table row is an XML fragment (of type `XMLType`) with a single `Warehouse` element.

```
SELECT *
  FROM XMLTable(
         'for $i in ora:view("OE", "WAREHOUSES")/ROW
          return <Warehouse id="{$i/WAREHOUSE_ID}">
                   <Location>
                     {for $j in ora:view("HR", "LOCATIONS")/ROW
                      where $j/LOCATION_ID eq $i/LOCATION_ID
                      return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                   </Location>
                 </Warehouse>');
```

This produces the same result as Example 17–6, except that each `Warehouse` element is output as a separate row, instead of all `Warehouse` elements being output together in a single row.

```
COLUMN_VALUE
------------------------------------------------------------
<Warehouse id="1">
  <Location>
    <STREET_ADDRESS>2014 Jabberwocky Rd</STREET_ADDRESS>
    <CITY>Southlake</CITY>
    <STATE_PROVINCE>Texas</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="2">
  <Location>
    <STREET_ADDRESS>2011 Interiors Blvd</STREET_ADDRESS>
```

```
    <CITY>South San Francisco</CITY>
    <STATE_PROVINCE>California</STATE_PROVINCE>
  </Location>
</Warehouse>
. . .

9 rows selected.
```

See Also: Example 17–15 for the `EXPLAIN PLAN` of Example 17–7

Using XQuery with XMLType Data

This section presents examples of using XQuery with `XMLType` relational data.

Example 17–8 Using XMLQuery with PASSING Clause, to Query an XMLType Column

This example passes an `XMLType` column, `oe.warehouse_spec`, as *context* item to XQuery, using function `XMLQuery` with the `PASSING` clause. It constructs a `Details` element for each of the warehouses whose area is greater than 80,000: `/Warehouse/Area > 80000`.

```
SELECT warehouse_name,
       XMLQuery(
         'for $i in /Warehouse
          where  $i/Area > 80000
          return <Details>
                   <Docks num="{$i/Docks}"/>
                   <Rail>{if ($i/RailAccess = "Y") then "true" else "false"}
                   </Rail>
                 </Details>'
        PASSING warehouse_spec RETURNING CONTENT) big_warehouses
  FROM warehouses;
```

This produces the following output:

```
WAREHOUSE_NAME
--------------
BIG_WAREHOUSES
--------------
Southlake, Texas


San Francisco


New Jersey
<Details><Docks></Docks><Rail>false</Rail></Details>

Seattle, Washington
<Details><Docks num="3"></Docks><Rail>true</Rail></Details>

Toronto


Sydney


Mexico City


Beijing
```

```
Bombay


9 rows selected.
```

In Example 17–8, function XMLQuery is applied to the warehouse_spec column in each row of table warehouses. The various FLWOR clauses perform these operations:

- **for** iterates over the Warehouse elements in each row of column warehouse_spec (the passed context item): each such element is bound to variable $i, in turn. The iteration is unordered.
- **where** filters the Warehouse elements, keeping only those whose Area child has a value greater than 80,000.
- **return** constructs an XQuery sequence of Details elements, each of which contains a Docks and a Rail child elements. The num attribute of the constructed Docks element is set to the text() value of the Docks child of Warehouse. The text() content of Rail is set to true or false, depending on the value of the RailAccess attribute of element Warehouse.

The SELECT statement applies to each row in table warehouses. The XMLQuery expression returns the *empty sequence* for those rows that do not match the XQuery expression. Only the warehouses in New Jersey and Seattle satisfy the XQuery query, so they are the only warehouses for which <Details>...</Details> is returned.

Example 17–9 Using XMLTable with XML Schema-Based Data

This example uses SQL function XMLTable to query an XMLType table, hr.purchaseorder, which contains XML Schema-based data. It uses the PASSING clause to provide the purchaseorder table as the context item for the XQuery-expression argument to XMLTable. Pseudocolumn COLUMN_VALUE of the resulting virtual table holds a constructed element, A10po, which contains the Reference information for those purchase orders whose CostCenter element has value A10 and whose User element has value SMCCAIN. The query performs a join between the virtual table and database table purchaseorder.

```
SELECT xtab.COLUMN_VALUE
  FROM purchaseorder, XMLTable('for $i in /PurchaseOrder
                                where $i/CostCenter eq "A10"
                                  and $i/User eq "SMCCAIN"
                                return <A10po pono="{$i/Reference}"/>'
                               PASSING OBJECT_VALUE) xtab;

COLUMN_VALUE
--------------------------------------------------
<A10po pono="SMCCAIN-20021009123336151PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336341PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337173PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335681PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335470PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336972PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336842PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336512PDT"></A10po>
<A10po pono="SMCCAIN-20021009123333894PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337403PDT"></A10po>

10 rows selected.
```

Example 17–10 Using XMLQuery with Schema-Based Data

This example is similar to Example 17–9 in its effect. It uses `XMLQuery`, instead of `XMLTable`, to query `hr.purchaseorder`. These two examples differ in their treatment of the empty sequences returned by the XQuery expression. In Example 17–9, these empty sequences are not joined with the `purchaseorder` table, so the overall SQL-query result set has only ten rows. In Example 17–10, these empty sequences are part of the overall result set of the SQL query, which contains 132 rows, one for each of the rows in table `purchaseorder`. All but ten of those rows are empty, and show up in the output as empty lines. To save space here, those empty lines have been removed.

```
SELECT XMLQuery('for $i in /PurchaseOrder
                 where $i/CostCenter eq "A10"
                   and $i/User eq "SMCCAIN"
                 return <A10po pono="{$i/Reference}"/>'
                PASSING OBJECT_VALUE
                RETURNING CONTENT)
  FROM purchaseorder;

XMLQUERY('FOR$IIN/PURCHASEORDERWHERE$I/COSTCENTEREQ"A10"AND$I/USEREQ"SMCCAIN"RET
--------------------------------------------------------------------------------
<A10po pono="SMCCAIN-20021009123336151PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336341PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337173PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335681PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335470PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336972PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336842PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336512PDT"></A10po>
<A10po pono="SMCCAIN-20021009123333894PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337403PDT"></A10po>

132 rows selected.
```

See Also: Example 17–16 for the `EXPLAIN PLAN` of Example 17–10

Example 17–11 Using XMLTable with PASSING and COLUMNS Clauses

This example uses `XMLTable` clauses `PASSING` and `COLUMNS`. The XQuery expression iterates over top-level `PurchaseOrder` elements, constructing a `PO` element for each purchase order with cost center `A10`. The resulting `PO` elements are then passed to `XMLTable` for processing.

Data from the children of `PurchaseOrder` is used to construct the children of `PO`, which are `Ref`, `Type`, and `Name`. The content of `Type` is taken from the content of `/PurchaseOrder/SpecialInstructions`, but the classes of `SpecialInstructions` are divided up differently for `Type`.

Function `XMLTable` shreds the result of XQuery evaluation, returning it as three `VARCHAR2` columns of a virtual table: `poref`, `priority`, and `contact`. The `DEFAULT` clause is used to supply a default `priority` of `Regular`.

```
SELECT xtab.poref, xtab.priority, xtab.contact
  FROM purchaseorder,
       XMLTable('for $i in /PurchaseOrder
                 let $spl := $i/SpecialInstructions
                 where $i/CostCenter eq "A10"
                 return <PO>
                          <Ref>{$i/Reference}</Ref>
                          {if ($spl eq "Next Day Air" or $spl eq "Expedite") then
```

```
                          <Type>Fastest</Type>
                        else if ($spl eq "Air Mail") then
                          <Type>Fast</Type>
                        else ()}
                       <Name>{$i/Requestor}</Name>
                     </PO>'
                PASSING OBJECT_VALUE
                COLUMNS poref    VARCHAR2(20) PATH '/PO/Ref',
                        priority VARCHAR2(8)  PATH '/PO/Type' DEFAULT 'Regular',
                        contact  VARCHAR2(20) PATH '/PO/Name') xtab;

POREF                PRIORITY CONTACT
-------------------- -------- --------------------
SKING-20021009123336 Fastest  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
SMCCAIN-200210091233 Fastest  Samuel B. McCain
JCHEN-20021009123337 Fastest  John Z. Chen
JCHEN-20021009123337 Regular  John Z. Chen
SKING-20021009123337 Regular  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
JCHEN-20021009123338 Regular  John Z. Chen
SMCCAIN-200210091233 Regular  Samuel B. McCain
SKING-20021009123335 Regular  Steven X. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
SKING-20021009123336 Regular  Steven A. King
SMCCAIN-200210091233 Fast     Samuel B. McCain
SKING-20021009123336 Fastest  Steven A. King
SKING-20021009123336 Fastest  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
JCHEN-20021009123335 Regular  John Z. Chen
SKING-20021009123336 Regular  Steven A. King
JCHEN-20021009123336 Regular  John Z. Chen
SKING-20021009123336 Regular  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
SKING-20021009123337 Regular  Steven A. King
SKING-20021009123338 Fastest  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
JCHEN-20021009123337 Regular  John Z. Chen
JCHEN-20021009123337 Regular  John Z. Chen
JCHEN-20021009123337 Regular  John Z. Chen
SKING-20021009123337 Regular  Steven A. King
JCHEN-20021009123337 Regular  John Z. Chen
SKING-20021009123337 Regular  Steven A. King
SKING-20021009123337 Regular  Steven A. King
SMCCAIN-200210091233 Fast     Samuel B. McCain

32 rows selected.
```

Example 17–12 Using XMLTable to Shred XML Collection Elements into Relational Data

In this example, SQL function `XMLTable` is used to shred the XML data in an `XMLType` collection element, `LineItem`, into separate columns of a virtual table.

```
SELECT lines.lineitem, lines.description, lines.partid,
       lines.unitprice, lines.quantity
  FROM purchaseorder,
       XMLTable('for $i in /PurchaseOrder/LineItems/LineItem
                 where $i/@ItemNumber >= 8
                  and $i/Part/@UnitPrice > 50
                  and $i/Part/@Quantity > 2
                 return $i'
```

```
                PASSING OBJECT_VALUE
                COLUMNS lineitem    NUMBER       PATH '@ItemNumber',
                        description VARCHAR2(30) PATH 'Description',
                        partid      NUMBER       PATH 'Part/@Id',
                        unitprice   NUMBER       PATH 'Part/@UnitPrice',
                        quantity    NUMBER       PATH 'Part/@Quantity') lines;

LINEITEM DESCRIPTION                         PARTID UNITPRICE QUANTITY
-------- ------------------------------ ----------- --------- --------
      11 Orphic Trilogy                 37429148327        80        3
      22 Dreyer Box Set                 37429158425        80        4
      11 Dreyer Box Set                 37429158425        80        3
      16 Dreyer Box Set                 37429158425        80        3
       8 Dreyer Box Set                 37429158425        80        3
      12 Brazil                         37429138526        60        3
      18 Eisenstein: The Sound Years    37429149126        80        4
      24 Dreyer Box Set                 37429158425        80        3
      14 Dreyer Box Set                 37429158425        80        4
      10 Brazil                         37429138526        60        3
      17 Eisenstein: The Sound Years    37429149126        80        3
      16 Orphic Trilogy                 37429148327        80        4
      13 Orphic Trilogy                 37429148327        80        4
      10 Brazil                         37429138526        60        4
      12 Eisenstein: The Sound Years    37429149126        80        3
      12 Dreyer Box Set                 37429158425        80        4
      13 Dreyer Box Set                 37429158425        80        4

17 rows selected.
```

See Also: Example 17–17 for the `EXPLAIN PLAN` of Example 17–12

Using Namespaces with XQuery

You can use the XQuery `declare namespace` declaration in the prolog of an XQuery expression to define a namespace prefix. You can use `declare` **`default`** `namespace` to establish the namespace as the default namespace for the expression.

An XQuery namespace declaration has no effect outside of its XQuery expression, however. To declare a namespace prefix for use in an `XMLTable` expression outside of the XQuery expression, use the `XMLNAMESPACES` clause. This clause also covers the XQuery expression argument to `XMLTable`, eliminating the need for a separate declaration in the XQuery prolog.

In Example 17–13, `XMLNAMESPACES` is used to define the prefix `e` for the namespace `http://emp.com`. This namespace is used in the `COLUMNS` clause as well as the XQuery expression of the `XMLTable` expression.

Example 17–13 Using XMLTable with the NAMESPACES Clause

```
SELECT * FROM XMLTable(XMLNAMESPACES('http://emp.com' AS "e"),
                       'for $i in doc("/public/empsns.xml")
                        return $i/e:emps/e:emp'
                       COLUMNS name VARCHAR2(6) PATH '@ename',
                               id   NUMBER      PATH '@empno');
```

This produces the following result:

```
NAME           ID
------ ----------
John            1
Jack            2
```

```
Jill            3

3 rows selected.
```

It is the presence of qualified names `e:ename` and `e:empno` in the `COLUMNS` clause that necessitates using the `XMLNAMESPACES` clause. Otherwise, a prolog namespace declaration (`declare namespace e = "http://emp.com"`) would suffice for the XQuery expression itself.

Because the same namespace is used throughout the `XMLTable` expression, a default namespace could be used: `XMLNAMESPACES (DEFAULT 'http://emp.com')`. The qualified name `$i/e:emps/e:emp` could then be written without an explicit prefix: `$i/emps/emp`.

Performance Tuning for XQuery

As mentioned, Oracle XML DB generally evaluates XQuery expressions by executing equivalent relational expressions. This optimization uses the same mechanism as XPath rewrite, and it provides the same benefits. To tune the execution performance of XQuery expressions:

1. Print and examine the associated `EXPLAIN PLAN`s.
2. If execution is not optimal, perform XPath tuning as discussed in Chapter 3 and Chapter 6.

This section presents the `EXPLAIN PLAN`s for some of the examples shown earlier, to indicate how they are executed. The examples are organized into the following groups according to the target of the XQuery expression:

- a SQL/XML view created on the fly using `ora:view`
- an XML schema-based `XMLType` table

See Also:

- "Understanding and Optimizing XPath Rewrite" on page 3-57
- "Diagnosing XPath Rewrite" on page 6-13

XQuery Optimization over a SQL/XML View Created by ora:view

Example 17–14 shows the optimization of `XMLQuery` over a SQL/XML view created by `ora:view`. Example 17–15 shows the optimization of `XMLTable` in the same context.

Example 17–14 Optimization of XMLQuery with ora:view

Here, again, is Example 17–6:

```
SELECT XMLQuery(
         'for $i in ora:view("OE", "WAREHOUSES")/ROW
          return <Warehouse id="{$i/WAREHOUSE_ID}">
                 <Location>
                   {for $j in ora:view("HR", "LOCATIONS")/ROW
                    where $j/LOCATION_ID eq $i/LOCATION_ID
                    return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                 </Location>
                 </Warehouse>'
        RETURNING CONTENT) FROM DUAL;
```

The EXPLAIN PLAN for this example shows that the query has been optimized:

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------
Plan hash value: 2976528487

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |     1 |       |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE               |            |     1 |    41 |            |          |
|   2 |   NESTED LOOPS                |            |     1 |    41 |     3   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| LOCATIONS  |     1 |    41 |     1   (0)| 00:00:01 |
|*  4 |     INDEX UNIQUE SCAN         | LOC_ID_PK  |     1 |       |     0   (0)| 00:00:01 |
|   5 |    FAST DUAL                  |            |     1 |       |     2   (0)| 00:00:01 |
|   6 |     SORT AGGREGATE            |            |     1 |   185 |            |          |
|   7 |      NESTED LOOPS             |            |     9 |  1665 |     4   (0)| 00:00:01 |
|   8 |       FAST DUAL               |            |     1 |       |     2   (0)| 00:00:01 |
|   9 |       TABLE ACCESS FULL       | WAREHOUSES |     9 |  1665 |     2   (0)| 00:00:01 |
|  10 |        FAST DUAL              |            |     1 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

 4 - access("LOCATION_ID"=:B1)

22 rows selected.
```

Example 17–15 Optimization of XMLTable with ora:view

Here, again, is Example 17–7:

```
SELECT *
  FROM XMLTable(
         'for $i in ora:view("OE", "WAREHOUSES")/ROW
          return <Warehouse id="{$i/WAREHOUSE_ID}">
                   <Location>
                     {for $j in ora:view("HR", "LOCATIONS")/ROW
                      where $j/LOCATION_ID eq $i/LOCATION_ID
                      return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                   </Location>
                 </Warehouse>');
```

The EXPLAIN PLAN for this example shows that the query has been optimized:

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------
Plan hash value: 2573750906

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |     9 |  1665 |     4   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE               |            |     1 |    41 |            |          |
|   2 |   NESTED LOOPS                |            |     1 |    41 |     3   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| LOCATIONS  |     1 |    41 |     1   (0)| 00:00:01 |
|*  4 |     INDEX UNIQUE SCAN         | LOC_ID_PK  |     1 |       |     0   (0)| 00:00:01 |
|   5 |    FAST DUAL                  |            |     1 |       |     2   (0)| 00:00:01 |
|   6 |     NESTED LOOPS              |            |     9 |  1665 |     4   (0)| 00:00:01 |
|   7 |      FAST DUAL                |            |     1 |       |     2   (0)| 00:00:01 |
|   8 |      TABLE ACCESS FULL        | WAREHOUSES |     9 |  1665 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

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

20 rows selected.
```

XQuery Optimization over XML Schema-Based XMLType Data

Example 17–16 shows the optimization of `XMLQuery` over an XML schema-based `XMLType` table. Example 17–17 shows the optimization of `XMLTable` in the same context.

Example 17–16 Optimization of XMLQuery with Schema-Based XMLType Data

Here, again, is Example 17–10:

```
SELECT XMLQuery('for $i in /PurchaseOrder
                 where $i/CostCenter eq "A10"
                   and $i/User eq "SMCCAIN"
                 return <A10po pono="{$i/Reference}"/>'
                PASSING OBJECT_VALUE
                RETURNING CONTENT)
  FROM purchaseorder;
```

The `EXPLAIN PLAN` for this example shows that the query has been optimized.

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------
Plan hash value: 3611789148

-------------------------------------------------------------------------------------
| Id  | Operation           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |               |     1 |   530 |     5   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE     |               |     1 |       |            |          |
|*  2 |   FILTER            |               |       |       |            |          |
|   3 |    FAST DUAL        |               |     1 |       |     2   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| PURCHASEORDER |     1 |   530 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   2 - filter(:B1='SMCCAIN' AND :B2='A10')
   4 - filter(SYS_CHECKACL("ACLOID","OWNERID",xmltype('<privilege
             xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
             http://xmlns.oracle.com/xdb/acl.xsd DAV:http://xmlns.oracle.com/xdb/dav.xsd">
             <read-properties/><read-contents/></privilege>'))=1)

22 rows selected.
```

Example 17–17 Optimization of XMLTable with Schema-Based XMLType Data

Here, again, is Example 17–12:

```
SELECT lines.lineitem, lines.description, lines.partid,
       lines.unitprice, lines.quantity
  FROM purchaseorder,
       XMLTable('for $i in /PurchaseOrder/LineItems/LineItem
                 where $i/@ItemNumber >= 8
                  and $i/Part/@UnitPrice > 50
```

```
  and $i/Part/@Quantity > 2
 return $i'
PASSING OBJECT_VALUE
COLUMNS lineitem    NUMBER       PATH '@ItemNumber',
        description VARCHAR2(30) PATH 'Description',
        partid      NUMBER       PATH 'Part/@Id',
        unitprice   NUMBER       PATH 'Part/@UnitPrice',
        quantity    NUMBER       PATH 'Part/@Quantity') lines;
```

The `EXPLAIN PLAN` for this example shows that the query has been optimized. The XQuery result is never materialized. Instead, the underlying storage columns for the XML collection element `LineItem` are used to generate the overall result set.

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 3113556559

------------------------------------------------------------------------------------------
| Id  | Operation          | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                   |     4 |   376 |     6   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |                   |     4 |   376 |     6   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| PURCHASEORDER     |     1 |    37 |     5   (0)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN | SYS_IOT_TOP_48748 |     3 |   171 |     1   (0)| 00:00:01 |
------------------------------------------------------------------------------------------

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

   2 - 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"><re
              ad-properties/><read-contents/></privilege>'))=1)
   3 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")
       filter("SYS_NC00013$">50 AND "SYS_NC00012$">2 AND "ITEMNUMBER">=8)

22 rows selected.
```

In this example, table `hr.purchaseorder` is traversed completely; the `XMLTable` expression is evaluated for each purchase-order document. It is more efficient, however, to have the `XMLTable` expression, not the `purchaseorder` table, drive the SQL-query execution. That is, although the XQuery expression has been rewritten to relational expressions, you can improve this optimization by creating an *index* on the underlying relational data—you can optimize this query in the same way that you would optimize a purely SQL query.

That is always the case with XQuery in Oracle XML DB: the optimization techniques you use are the same that you use in SQL.

The `UnitPrice` attribute of collection element `LineItem` is an appropriate index target. The governing XML schema specifies that a nested table is used to store the `LineItem` elements.

However, the name of this nested table was system-generated when the XML purchase-order documents were shredded as XML schema-based data. Instead of using table `purchaseorder` from sample database-schema `hr`, for illustration we will manually create a new `purchaseorder` table (in a different database schema) with the same properties and same data, but having nested tables with user-friendly names. Refer to Example 3–11 on page 3-27 for how to do this.

Assuming that a purchaseorder table has been created as in Example 3–11, the following statement creates the appropriate index:

```
CREATE INDEX unitprice_index ON lineitem_table("PART"."UNITPRICE");
```

With this index defined, Example 17–12 results in the following EXPLAIN PLAN, which shows that the XMLTable expression has driven the overall evaluation.

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------
Plan hash value: 1578014525

-------------------------------------------------------------------------------------------
| Id  | Operation          | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                   |     3 |   624 |     8   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |                   |     3 |   624 |     8   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN| SYS_IOT_TOP_49323 |     3 |   564 |     5   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN| UNITPRICE_INDEX   |    20 |       |     2   (0)| 00:00:01 |
|*  4 |   INDEX UNIQUE SCAN| SYS_C004411       |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   2 - access("SYS_NC00013$">50)
       filter("ITEMNUMBER">=8 AND "SYS_NC00012$">2)
   3 - access("SYS_NC00013$">50)
   4 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")

Note
-----
   - dynamic sampling used for this statement

23 rows selected.
```

XQuery Static Type-Checking in Oracle XML DB

Oracle XML DB performs static (that is, compile-time) type-checking of XQuery expressions. It also performs dynamic (runtime) type-checking. This section presents examples that demonstrate the utility of static type-checking.

Example 17–18 Static Type-Checking of XQuery Expressions: ora:view

The XML view produced on the fly by Oracle XQuery function ora:view has ROW as its top-level element, but this example incorrectly lacks that ROW wrapper element. This omission raises a compile-time error. Forgetting that ora:view wraps relational data in this way is an easy mistake to make, and one that could be difficult to diagnose without static type-checking. Example 17–5 shows the correct code.

```
-- This produces a static-type-check error, because "ROW" is missing.
SELECT XMLQuery('for $i in ora:view("REGIONS"), $j in ora:view("COUNTRIES")
                 where $i/REGION_ID = $j/REGION_ID and $i/REGION_NAME = "Asia"
                 return $j'
                RETURNING CONTENT) AS asian_countries
  FROM DUAL;
SELECT XMLQuery('for $i in ora:view("REGIONS"), $j in ora:view("COUNTRIES")
*
ERROR at line 1:
ORA-19276: XP0005 - XPath step specifies an invalid element/attribute name:
(REGION_ID)
```

Example 17–19 Static Type-Checking of XQuery Expressions: Schema-Based XML

In this example, XQuery static type-checking finds a mismatch between an XPath expression and its target XML schema-based data. Element `CostCenter` is misspelled here as `costcenter` (XQuery and XPath are case-sensitive). Example 17–11 shows the correct code.

```
-- This results in a static-type-check error: CostCenter is not the right case.
SELECT xtab.poref, xtab.usr, xtab.requestor
  FROM purchaseorder,
       XMLTable('for $i in /PurchaseOrder where $i/costcenter eq "A10" return $i'
                PASSING OBJECT_VALUE
                COLUMNS poref     VARCHAR2(20) PATH 'Reference',
                        usr       VARCHAR2(20) PATH 'User' DEFAULT 'Unknown',
                        requestor VARCHAR2(20) PATH 'Requestor') xtab;
  FROM purchaseorder,
       *
ERROR at line 2:
ORA-19276: XP0005 - XPath step specifies an invalid element/attribute name:
(costcenter)
```

SQL*Plus XQUERY Command

Example 17–20 shows how you can enter an XQuery expression directly at the SQL*Plus command line, by preceding the expression with the SQL*Plus command **`XQUERY`** and following it with a slash (`/`) on a line by itself. Oracle Database treats XQuery expressions submitted with this command the same way it treats XQuery expressions in SQL functions `XMLQuery` and `XMLTable`. Execution is identical, with the same optimizations.

Example 17–20 Using the SQL*Plus XQUERY Command

```
SQL> XQUERY for $i in ora:view("departments")
  2  where $i/ROW/DEPARTMENT_ID < 50
  3  return $i
  4  /

Result Sequence
--------------------------------------------------------------------------------
<ROW><DEPARTMENT_ID>10</DEPARTMENT_ID><DEPARTMENT_NAME>Administration</DEPARTMEN
T_NAME><MANAGER_ID>200</MANAGER_ID><LOCATION_ID>1700</LOCATION_ID></ROW>

<ROW><DEPARTMENT_ID>20</DEPARTMENT_ID><DEPARTMENT_NAME>Marketing</DEPARTMENT_NAM
E><MANAGER_ID>201</MANAGER_ID><LOCATION_ID>1800</LOCATION_ID></ROW>

<ROW><DEPARTMENT_ID>30</DEPARTMENT_ID><DEPARTMENT_NAME>Purchasing</DEPARTMENT_NA
ME><MANAGER_ID>114</MANAGER_ID><LOCATION_ID>1700</LOCATION_ID></ROW>

<ROW><DEPARTMENT_ID>40</DEPARTMENT_ID><DEPARTMENT_NAME>Human Resources</DEPARTME
NT_NAME><MANAGER_ID>203</MANAGER_ID><LOCATION_ID>2400</LOCATION_ID></ROW>
```

There are also a few SQL*Plus `SET` commands that you can use for settings that are specific to XQuery. Use `SHOW XQUERY` to see the current settings.

- **`SET XQUERY BASEURI`** – Set the base URI for `XQUERY`. URIs in XQuery expressions are relative to this URI.
- **`SET XQUERY ORDERING`** – Set the `XQUERY` execution mode to `ORDERED` or `UNORDERED` (the default value). See "Functional Language Based on Sequences" on page 17-2.

- **SET XQUERY NODE** – Set the node-identity preservation mode to `BY REFERENCE` or `BY VALUE` (the default value). `BY REFERENCE` preserves XML node identity for subsequent operations. `BY VALUE` means that node identity need not be preserved.
- **SET XQUERY CONTEXT** – Specify a context item for subsequent `XQUERY` evaluations.

See Also: *SQL*Plus User's Guide and Reference*

Using XQuery with PL/SQL, JDBC, and ODP.NET

Previous sections in this chapter have shown how to invoke XQuery from SQL. This section provides examples of using XQuery with the Oracle APIs for PL/SQL, JDBC, and Oracle Data Provider for .NET (ODP.NET).

Example 17–21 Using XQuery with PL/SQL

This example shows how to use XQuery with PL/SQL, in particular, how to bind *dynamic variables* to an XQuery expression using the `XMLQuery PASSING` clause. The bind variables `:1` and `:2` are bound to the PL/SQL bind arguments `nbitems` and `partid`, respectively. These are then passed to XQuery as XQuery variables `itemno` and `id`, respectively.

```
DECLARE
  sql_stmt VARCHAR2(2000); -- Dynamic SQL statement to execute
  nbitems  NUMBER := 3; -- Number of items
  partid   VARCHAR2(20):= '715515009058'; -- Part ID
  result   XMLType;
  doc      DBMS_XMLDOM.DOMDocument;
  ndoc     DBMS_XMLDOM.DOMNode;
  buf      VARCHAR2(20000);
BEGIN
  sql_stmt :=
    'SELECT XMLQuery(
               ''for $i in ora:view("PURCHASEORDER") ' ||
                'where count($i/PurchaseOrder/LineItems/LineItem)
                       = $itemno/itemNode ' ||
                  'and $i/PurchaseOrder/LineItems/LineItem/Part/@Id
                       = $id/idNode ' ||
                'return $i/PurchaseOrder/LineItems'' ' ||
               'PASSING XMLElement("itemNode", :1) AS "itemno",
                        XMLElement("idNode", :2) AS "id"' ||
               'RETURNING CONTENT) FROM DUAL';

  EXECUTE IMMEDIATE sql_stmt INTO result USING nbitems, partid;
  doc  := DBMS_XMLDOM.newDOMDocument(result);
  ndoc := DBMS_XMLDOM.makeNode(doc);
  DBMS_XMLDOM.writeToBuffer(ndoc, buf);
  DBMS_OUTPUT.put_line(buf);
END;
/
```

This produces the following output:

```
<LineItems>
  <LineItem ItemNumber="1">
    <Description>Samurai 2: Duel at Ichijoji Temple</Description>
    <Part Id="37429125526" UnitPrice="29.95" Quantity="3"/>
  </LineItem>
```

```
    <LineItem ItemNumber="2">
      <Description>The Red Shoes</Description>
      <Part Id="37429128220" UnitPrice="39.95" Quantity="4"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="1"/>
    </LineItem>
  </LineItems>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="2">
      <Description>The Unbearable Lightness Of Being</Description>
      <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>Sisters</Description>
      <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
    </LineItem>
  </LineItems>

  PL/SQL procedure successfully completed.
```

Example 17–22 Using XQuery with JDBC

This example shows how to use XQuery with JDBC, binding variables by position with the `PASSING` clause of SQL function `XMLTable`.

```
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.driver.*;
import oracle.xdb.XMLType;
import java.util.*;

public class QueryBindByPos
{
  public static void main(String[] args) throws Exception, SQLException
  {
    System.out.println("*** JDBC Access of XQuery using Bind Variables ***");
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    OracleConnection conn
      = (OracleConnection)
        DriverManager.getConnection("jdbc:oracle:oci8:@localhost:1521:ora10gR2", "oe", "oe");
    String xqString
      = "SELECT column_value" +
          "FROM XMLTable('for $i in ora:view(\"PURCHASEORDER\") " +
                         "where $i/PurchaseOrder/Reference= $ref/refNode " +
                         "return $i/PurchaseOrder/LineItems/LineItem/Description/text()' " +
                        "PASSING XMLElement(\"refNode\", ?) AS \"ref\")";
    OraclePreparedStatement stmt = (OraclePreparedStatement)conn.prepareStatement(xqString);
    String refString = "EABEL-20021009123336251PDT"; // Set the filter value
    stmt.setString(1, refString); // Bind the string
    ResultSet rs = stmt.executeQuery();
    while (rs.next())
    {
       XMLType desc = (XMLType) rs.getObject(1);
       System.out.println("LineItem Description: " + desc.getStringVal());
    }
```

```
    rs.close();
    stmt.close();
  }
}
```

This produces the following output:

```
*** JDBC Access of Database XQuery with Bind Variables ***
LineItem Description: Samurai 2: Duel at Ichijoji Temple
LineItem Description: The Red Shoes
LineItem Description: A Night to Remember
```

Example 17–23 Using XQuery with ODP.NET and C#

This example shows how to use XQuery with ODP.NET and the C# language. The C# input parameters `:nbitems` and `:partid` are passed to XQuery as XQuery variables `itemno` and `id`, respectively.

```
using System;
using System.Data;
using System.Text;
using System.IO;
using System.Xml;
using Oracle.DataAccess.Client;
using Oracle.DataAccess.Types;

namespace XQuery
{
  /// <summary>
  /// Demonstrates how to bind variables for XQuery calls
  /// </summary>
  class XQuery
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    static void Main(string[] args)
    {
      int rows = 0;
      StreamReader sr = null;

      // Create the connection.
      string constr = "User Id=oe;Password=oe;Data Source=ora10gr2";
      OracleConnection con = new OracleConnection(constr);
      con.Open();

      // Create the command.
      OracleCommand cmd = new OracleCommand("", con);

      // Set the XML command type to query.
      cmd.CommandType   = CommandType.Text;

      // Create the SQL query with the XQuery expression.
      StringBuilder blr = new StringBuilder();
      blr.Append("SELECT column_value FROM XMLTable");
      blr.Append("(\'for $i in ora:view(\"PURCHASEORDER\") ");
      blr.Append("   where count($i/PurchaseOrder/LineItems/LineItem) = $itemno/itemNode ");
      blr.Append("      and $i/PurchaseOrder/LineItems/LineItem/Part/@Id = $id/idNode ");
      blr.Append("   return $i/PurchaseOrder/LineItems\' ");
      blr.Append("  PASSING XMLElement(\"itemNode\", :nbitems) AS \"itemno\",
                            XMLElement(\"idNode\", :partid) AS \"id\")");

      cmd.CommandText = blr.ToString();
      cmd.Parameters.Add(":nbitems", OracleDbType.Int16, 3, ParameterDirection.Input);
      cmd.Parameters.Add(":partid", OracleDbType.Varchar2, "715515009058", ParameterDirection.Input);
```

```
      // Get the XML document as an XmlReader.
      OracleDataReader dr = cmd.ExecuteReader();
      dr.Read();

      // Get the XMLType column as an OracleXmlType
      OracleXmlType xml = dr.GetOracleXmlType(0);

      // Print out the XML data in the OracleXmlType object
      Console.WriteLine(xml.Value);
      xml.Dispose();

      // Clean up.
      cmd.Dispose();
      con.Close();
      con.Dispose();
    }
  }
}
```

This produces the following output:

```
<LineItems>
  <LineItem ItemNumber="1">
    <Description>Samurai 2: Duel at Ichijoji Temple</Description>
    <Part Id="37429125526" UnitPrice="29.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>The Red Shoes</Description>
    <Part Id="37429128220" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>A Night to Remember</Description>
    <Part Id="715515009058" UnitPrice="39.95" Quantity="1"/>
  </LineItem>
</LineItems>
```

See Also:

- Chapter 11, "PL/SQL API for XMLType"
- Chapter 13, "Java API for XMLType"
- Chapter 15, "Using Oracle Data Provider for .NET with Oracle XML DB"

Oracle XML DB Support for XQuery

Oracle XML DB supports the latest definition of the XQuery language. Because the language definition is an ongoing process, some areas of the language that are not yet firmly established are unsupported by Oracle XML DB or supported in a limited manner. This limits any impact on early adopters when the language definition evolves. Oracle participates vigorously in the definition of the XQuery language, is committed to full XQuery support, and will continue to remain at the forefront of XQuery development. For the latest status of the Oracle XML DB XQuery implementation, please consult the current version of *Oracle Database Read Me*.

Support for XQuery and SQL

Support for the XQuery language in Oracle XML DB is designed to provide the best fit between the worlds of relational storage and querying XML data. That is, Oracle XML DB is a general XQuery implementation, but it is in addition specifically designed to make relational and XQuery queries work well together.

The specific properties of the Oracle XML DB XQuery implementation are described in this section. The XQuery standard explicitly calls out certain aspects of the language processing as implementation-defined or implementation-dependent. There are also areas where the Oracle XML DB XQuery implementation departs from the standard.

Implementation Choices Specified in the XQuery Standard

The XQuery specification specifies that each of the following aspects of language processing is to be defined by the implementation.

- **Implicit time zone support –** In Oracle XML DB, the implicit time zone is always assumed to be `Z`, and timestamps with missing time zones are automatically converted to UTC.
- **Invalid XPath expressions –** Whenever it can determine at query compile time that an XPath expression is invalid, Oracle XML DB raises an *error*; it does *not* return the empty sequence.
- **Ordering mode –** In Oracle XML DB, the default ordering mode is `unordered`. You can use an ordering mode declaration in the prolog of an XQuery expression to set the ordering mode for the expression.

Implementation Departures from the XQuery Standard

- **Collations –** Oracle XML DB uses the SQL collation rules, based on the current settings for parameters `NLS_LANG`, `NLS_COMP`, and `NLS_SORT`. You can add a `default collation` clause to a query to force the standard XQuery collation rules.
- **Boundary condition differences –** The SQL behavior is generally used for XQuery in Oracle XML DB, if the SQL behavior differs from the standard XQuery behavior. Examples include:
  - **`doc`**, **`collection`** – When XQuery functions `fn:doc` and `fn:collection` are passed an invalid file or collection name, the XQuery standard requires and *error* to be raised, but Oracle XML DB returns an *empty sequence* instead. In particular, this means that a name that does not point to an Oracle XML DB Repository resource results in an empty sequence.
  - **`contains`** – The XQuery standard requires the result of `fn:contains` to be *true* when the first argument is a zero-length string, but Oracle XML DB returns *false* in this case.
  - **`mod`** – Evaluation of *x* `mod 0`, for any number *x*, is required by the the XQuery standard to return `NaN` (not a number), but Oracle XML DB returns *x* instead.
  - **`+0`** and **`-0`** – The XQuery standard distinguishes positive zero from negative zero, but Oracle XML DB does not: both are represented as `0`, and they are treated equally.
  - **Empty string and `NULL` –** The XQuery standard distinguishes an empty string from `NULL`, but Oracle XML DB does not: they are treated equally.

XQuery Optional Features

There is currently no support for the following optional XQuery features defined by the W3C:

- Schema Import Feature
- Schema Validation Feature
- Module Feature

In addition to these defined optional features, the W3C specification allows an implementation to provide implementation-defined pragmas and extensions. These include the following:

- Pragmas
- Must-understand extensions
- Static-typing extensions

The Oracle implementation does not require any such pragmas or extensions.

Support for XQuery Functions and Operators

Oracle XML DB supports all of the XQuery functions and operators included in the latest *XQuery 1.0 and XPath 2.0 Functions and Operators* specification, with the following exceptions. There is *no* support for the following:

- XQuery regular-expression functions. Use the Oracle extensions for regular-expression operations, instead.
- Implicit time zones, when using functions that involve durations, dates, or times. (See "Support for XQuery and SQL" on page 17-33.)
- Values of type `xs:IDREF` or `xs:IDREFS`, in string functions that involve nodes.
- Function `trace()`.

XQuery Functions doc and collection

XQuery built-in functions `fn:doc` and `fn:collection` are essentially implementation-defined. Oracle XML DB supports these functions for all *resources* in Oracle XML DB Repository. Function `doc` returns the repository *file* resource that is targeted by its URI argument; it must be a file of well-formed XML data. Function `collection` is similar, but works on repository *folder* resources (each file in the folder must contain well-formed XML data). Each of these functions returns an empty sequence if the targeted resource is not found – it does *not* raise an error.

> **See Also:** `http://www.w3.org` for the definitions of XQuery functions and operators

18

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 a function such as extract with getNumberVal() applied to the XMLType result. It is recommended that you use SQL function extract rather than XMLType method extract() 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 18-20.

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 3-47
- Chapter 5, "XML Schema Storage and Query: Basic"
- Appendix A, "XML Schema Primer"

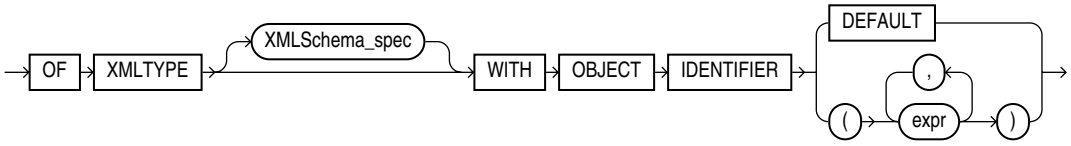
`XMLType` views can be constructed in the following ways:

- 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 SQL function `sys_XMLGen`. Non-schema-based `XMLType` views can be constructed using object types, object views, and function `sys_XMLGen` and XML schema-based `XMLType` view can be constructed using object types and object views. This enables the construction of the `XMLType` view from underlying relational or object relational tables directly without physically migrating the relational or object relational legacy data into XML. Creating non-schema-based `XMLType` view requires the use of `sys_XMLGen` over existing object types or object views. Creating XML-schema-based `XMLType` view requires to annotate the XML schema with a mapping to existing object types or to generate the XML schema from the existing object types.
- XML schema-based `XMLType` views can also be constructed directly from an `XMLType` table.

Creating XMLType Views: Syntax

Figure 18–1 shows the `CREATE VIEW` clause for creating `XMLType` views. See *Oracle Database SQL Reference* for details on the `CREATE VIEW` syntax.

Figure 18–1 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 16, "Generating XML Data from the Database", for details on SQL/XML generation functions

- Using object types and object views with SQL function `sys_XMLGen`. Here you create the `XMLType` view using object types with `sys_XMLGen`. This way of 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 to Create XML Schema-Based XMLType Views" on page 18-11

Using SQL/XML Generation Functions to Create Non-Schema-Based XMLType Views

Example 18–1 illustrates how to create an `XMLType` view using the SQL/XML function `XMLELement()`.

Example 18–1 Creating an XMLType View Using XMLELEMENT

The following statement creates an `XMLType` view using SQL function `XMLElement`:

```
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 operation, the XPath /Emp/@empno can refer directly to the empno column.

Existing data in relational tables or views can be exposed as XML using this mechanism. If a view is generated using a SQL/XML generation function, then queries that access the view with XPath expressions can often be optimized (rewritten). The optimized queries can then directly access the underlying relational columns. See "XPath Rewrite on XMLType Views" on page 18-20 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 to Create Non-Schema-Based XMLType Views

You can also create XMLType views using SQL function sys_XMLGen with object types. Function sys_XMLGen inputs object type and generates an XMLType. Here is an equivalent query that produces the same query results using sys_XMLGen:

Example 18–2 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>17-JUN-87</HIREDATE>
</EMP>

<?xml version="1.0"?>
<EMP empno="101">
  <FNAME>Neena</FNAME>
  <LNAME>Kochhar</LNAME>
  <HIREDATE>21-SEP-89</HIREDATE>
</EMP>

<?xml version="1.0"?>
<EMP empno="102">
  <FNAME>Lex</FNAME>
  <LNAME>De Haan</LNAME>
  <HIREDATE>13-JAN-93</HIREDATE>
</EMP>
```

Existing data in relational or object-relational tables or views can be exposed as XML using this mechanism. In addition, queries using SQL functions extract,

extractValue, and existsNode that involve simple XPath traversal over views generated by function 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 to Create XML Schema-Based XMLType Views" on page 18-5

- 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 to Create XML Schema-Based XMLType Views" on page 18-11

Using SQL/XML Generation Functions to Create XML Schema-Based XMLType Views

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.

These two steps are illustrated in Example 18–3 and Example 18–4, respectively.

Example 18–3 Registering XML Schema emp_simple.xsd

Assume that you have an XML schema emp_simple.xsd that contains XML structures defining an employee. This example shows how to 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
```

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.

Example 18–4 Creating an XMLType View Using SQL/XML Functions

```
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 Example 18–4, `XMLElement` creates the `Employee` XML element and the inner `XMLForest` function call creates the children of the `Employee` element. The `XMLAttributes` clause inside `XMLElement` constructs the required XML `namespace` and schema `location` attributes, so that the XML data that is generated conforms to the XML schema of the view. The innermost `XMLForest` function call creates the `department` XML element that is nested inside the `Employee` element.

The XML generation functions generate a non-schema-based XML instance, by default. However, when the schema location is specified, using attribute `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation`, Oracle XML DB generates XML 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, the XML is converted implicitly into a valid XML schema-based document. Any errors in the generated XML data are caught when further operations, such as validate or extract operations, are performed on the XML instance.

Example 18–5 Querying an XMLType View

This example queries the `XMLType` view, returning an XML result from the `employees` and `departments` tables. The result of the query is shown here pretty-printed here, for clarity.

```
SELECT OBJECT_VALUE AS RESULT FROM emp_simple_xml WHERE ROWNUM < 2;

RESULT
--------------------------------------------------------------------
<Employee xmlns="http://www.oracle.com/emp_simple.xsd"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.oracle.com/emp_simple.xsd
                              http://www.oracle.com/emp_simple.xsd">
  <EmployeeId>200</EmployeeId>
  <Name>Whalen</Name>
  <Job>AD_ASST</Job>
  <Manager>101</Manager>
  <HireDate>1987-09-17</HireDate>
  <Salary>4400</Salary>
  <Dept>
    <DeptNo>10</Deptno>
    <DeptName>Administration</DeptName>
    <Location>1700</Location>
  </Dept>
</Employee>
```

Using Namespaces With SQL/XML Functions

If you have complex 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 18–6 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 could 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 18–7 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 OBJECT_VALUE AS result FROM dept_xml WHERE ROWNUM < 2;

RESULT
--------------------------------------------------------------------
<Department
```

```
    xmlns="http://www.oracle.com/emp.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.oracle.com/dept.xsd
                        http://www.oracle.com/dept.xsd">
  <DeptNo>10</DeptNo>
  <DeptName>Administration</DeptName>
  <Location>1700</Location>
  <Employee>
    <EmployeeId>200</EmployeeId>
    <Name>Whalen</Name>
    <Job>AD_ASST</Job>
    <Manager>101</Manager>
    <Hiredate>1987-09-17</Hiredate>
    <Salary>4400</Salary>
  </Employee>
</Department>
```

Using Object Types and Views to Create XML Schema-Based XMLType Views

To wrap relational or object-relational data with strongly-typed XML using the object view approach, perform the following 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. 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: Basic" for more details on this process

   You can use PL/SQL functions `DBMS_XMLSchema .generateSchema` and `generateSchemas` 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 `VARCHAR2` or a Large Object (LOB) datatype.
   - ***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.

For examples, see the following sections, which are based on the employee and department relational tables and XML views of this data:

- "Creating Schema-Based XMLType Views Over Object Views"
- "Wrapping Relational Department Data with Nested Employee Data as XML"

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 Step 1 through Step 4b.

Step 1. Create Object Types Example 18–8 creates the object types for the views.

Example 18–8 Creating Object Types for Schema-Based XMLType Views

```
CREATE TYPE dept_t AS OBJECT
      (deptno        NUMBER(4),
       dname         VARCHAR2(30),
       loc           NUMBER(4));
/

CREATE TYPE emp_t AS OBJECT
      (empno         NUMBER(6),
       ename         VARCHAR2(25),
       job           VARCHAR2(10),
       mgr           NUMBER(6),
       hiredate      DATE,
       sal           NUMBER(8,2),
       comm          NUMBER(2,2),
       dept          dept_t );
/
```

Step 2. Create or Generate XMLSchema, emp.xsd You can create an XML schema manually or use package `DBMS_XMLSCHEMA` to generate it automatically from the existing object types, as shown in Example 18–9.

Example 18–9 Generating an XML Schema with DBMS_XMLSCHEMA.GENERATESCHEMA

```
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. Function `DBMS_XMLSCHEMA.generateSchemas` 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. Example 18–10 shows how to register XML schema `emp_complex.xsd`. See also the `xdb:SQLType` annotation Example 18–10.

Example 18–10 Registering XML Schema emp_complex.xsd

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/emp_complex.xsd', 4);
END;
```

```
/

COMMIT;

BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    '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 shown in Example 18–11.

Example 18–11 Creating an XMLType View

```
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 SQL function `extractValue` 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. It is recommended that you use SQL function `extractValue` rather than `XMLType` method `extractValue()`.

Step 4b. Using the Two-Step Process by First Creating an Object View In the two-step process, you first create an object view, then create an `XMLType` view on the object view, as shown in Example 18–12.

Example 18–12 Creating an Object View and an XMLType View on the Object View

```
CREATE OR REPLACE VIEW emp_v OF emp_t WITH OBJECT ID (empno) AS
  SELECT emp_t(e.employee_id, e.last_name, e.job_id, e.manager_id, e.hire_date,
               e.salary, e.commission_pct,
               dept_t(d.department_id, d.department_name, d.location_id))
    FROM employees e, departments d
    WHERE e.department_id = d.department_id;

CREATE OR REPLACE VIEW emp_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/emp_complex.xsd" ELEMENT "Employee"
  WITH OBJECT ID DEFAULT
  AS SELECT VALUE(p) FROM emp_v p;
```

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 Step 1 through Step 3b.

Step 1. Create Object Types The first step is to create the object types needed, as shown in Example 18–13.

Example 18–13 Creating Object Types

```
CREATE TYPE emp_t AS OBJECT (empno         NUMBER(6),
                             ename         VARCHAR2(25),
                             job           VARCHAR2(10),
                             mgr           NUMBER(6),
                             hiredate      DATE,
                             sal           NUMBER(8,2),
                             comm          NUMBER(2,2));
/

CREATE OR REPLACE TYPE emplist_t AS TABLE OF emp_t;
/

CREATE TYPE dept_t AS OBJECT (deptno        NUMBER(4),
                              dname         VARCHAR2(30),
                              loc           NUMBER(4),
                              emps          emplist_t);
/
```

Step 2. Register XML Schema, dept_complex.xsd You can either use a pre-existing XML schema or generate an XML schema from the object type with function `DBMS_XMLSCHEMA.generateSchema` or `DBMS_XMLSCHEMA.generateSchemas`. Example 18–14 shows how to register the XML schema `dept_complex.xsd`.

Example 18–14 Registering XML Schema dept_complex.xsd

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/dept_complex.xsd', 4);
END;
/

BEGIN
 DBMS_XMLSCHEMA.registerSchema('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 The next step is to create the `dept_xml` `XMLType` view from the department object type, as shown in Example 18–15.

Example 18–15 Creating XMLType Views on Relational Tables

```
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. Example 18–16 demonstrates this.

Example 18–16 Creating XMLType Views Using SQL/XML 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 18–17 Creating an XMLType View by Restricting Rows From an XMLType Table

This 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 section "Wrapping Relational Department Data with Nested Employee Data as XML", 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 OBJECT_VALUE FROM dept_xml_tab
           WHERE extractValue(OBJECT_VALUE, '/Department/Location') = 'DALLAS';
```

Here, `dallas_dept_view` restricts the `XMLType` table rows to those departments whose location is Dallas.

Example 18–18 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 9–2, "Using XMLTRANSFORM and DBURITYPE to Retrieve a Style Sheet" on page 9-4 for an example that uses XMLtransform:

```
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(OBJECT_VALUE,
                         XDBURIType('/home/SCOTT/xsl/po2.xsl').getxml())
       FROM po_tab;
```

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 18–19 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 XMLType tables or object types, object views, and SQL function sys_XMLGen are the same as for regular XMLType table columns. Hence, SQL functions extract, existsNode, and extractValue 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. Functions extract, existsNode, and extractValue on view columns get rewritten into underlying relational accesses for better performance.

See Also: Chapter 6, "XPath Rewrite"

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 18–20 illustrates XPath rewrites on non-schema-based XMLType views.

Example 18–20 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 SQL function extractValue to select from emp_view:

  ```
  SELECT extractValue(OBJECT_VALUE, '/Emp/name'),
         extractValue(OBJECT_VALUE, '/Emp/hiredate'
    FROM emp_view;
  ```

  This query becomes something like the following:

  ```
  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 expression is rewritten down to the relational column access as defined in view emp_view.

- Querying with SQL function extractValue followed by method getNumberVal() to select from emp_view:

  ```
  SELECT (extract(OBJECT_VALUE, '/Emp/@empno').getnumberval()) FROM emp_view;
  ```

  This query becomes something like the following:

  ```
  SELECT e.employee_id FROM employees e WHERE e.salary > 15000;
  ```

  The rewritten query is a simple relational query. The extract expression followed by getNumberVal() is rewritten down to the relational column access as defined in view emp_view.

- Querying with SQL function existsNode to select from view emp_view:

```
SELECT extractValue(OBJECT_VALUE, '/Emp/name'),
       extractValue(OBJECT_VALUE, '/Emp/hiredate')
  FROM emp_view WHERE existsNode(OBJECT_VALUE, '/Emp[@empno=101]') = 1;
```

This query becomes something like the following:

```
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 the `existsNode` expression is rewritten down to the predicate over relational columns as defined in view `emp_view`.

If there is an index created on column `employees.empno`, then the query optimizer can use the index to speed up the query.

Querying with `existsNode` to select from view `emp_view`:

```
SELECT extractValue(OBJECT_VALUE, '/Emp/name'),
       extractValue(OBJECT_VALUE, '/Emp/hiredate'),
       extractValue(OBJECT_VALUE, '/Emp/@empno')
  FROM emp_view
  WHERE existsNode(OBJECT_VALUE, '/Emp[name="Steven King" or @empno = 101] ')
        = 1;
```

This query becomes something like the following:

```
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 the `existsNode` expression is rewritten down to the predicate over relational columns as defined in view `emp_view`.

- Querying with `extract` to select from view `emp_view`:

```
SELECT extract(OBJECT_VALUE, '/Emp/name'),
       extract(OBJECT_VALUE, '/Emp/hiredate')
  FROM emp_view;
```

This query becomes something like the following:

```
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` expression is rewritten to expressions over relational columns.

Note: Since the view uses SQL function `XMLForest` to formulate `name` and `hiredate` elements, the rewritten query uses equivalent `CASE` expression to be consistent with `XMLForest` semantics.

XPath Rewrite on Schema-Based Views Constructed With SQL/XML

Example 18–21 illustrates an XPath rewrite on XML-schema-based `XMLType` view constructed with a SQL/XML function.

Example 18–21 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 SQL function `extractValue` to select from `emp_xml`:

```
SELECT
 extractValue(OBJECT_VALUE, '/Employee/EmployeeId') AS "a1",
 extractValue(OBJECT_VALUE, '/Employee/Name') AS "b1",
 extractValue(OBJECT_VALUE, '/Employee/Job') AS "c1",
 extractValue(OBJECT_VALUE, '/Employee/Manager') AS "d1",
 extractValue(OBJECT_VALUE, '/Employee/HireDate') AS "e1",
 extractValue(OBJECT_VALUE, '/Employee/Salary') AS "f1",
 extractValue(OBJECT_VALUE, '/Employee/Commission') AS "g1"
FROM emp_xml
WHERE existsNode(OBJECT_VALUE, '/Employee/Dept[Location = 1700]') = 1;
```

This query becomes something like the following:

```
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 the `existsNode` expression is rewritten down to the predicate over relational columns as defined in view `emp_view`:

Querying with SQL function `existsNode` to select from view `emp_xml`:

```
SELECT extractValue(OBJECT_VALUE, '/Employee/EmployeeId') as "a1",
       extractValue(OBJECT_VALUE, '/Employee/Dept/DeptNo') as "b1",
       extractValue(OBJECT_VALUE, '/Employee/Dept/DeptName') as "c1",
       extractValue(OBJECT_VALUE, '/Employee/Dept/Location') as "d1"
  FROM emp_xml
  WHERE existsNode(OBJECT_VALUE, '/Employee/Dept[Location = 1700
    AND DeptName="Finance"]') = 1;
```

This query becomes a simple relational query using the XPath rewrite mechanism. The XPath predicate in the `existsNode` expression is rewritten down to the predicate over relational columns as defined in view `emp_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;
```

Views Using Object Types, Object Views, and SYS_XMLGEN

The following sections describe XPath rewrite on `XMLType` views using object types, views, and SQL function `sys_XMLGen`.

Non-Schema-Based XMLType Views Using Object Types or Object 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 18–22 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(OBJECT_VALUE) FROM dept_ov;
```

Querying department numbers that have at least one employee making a salary more than $15000:

```
SELECT extractValue(OBJECT_VALUE, '/ROW/DEPTNO')
  FROM dept_xml
  WHERE existsNode(OBJECT_VALUE, '/ROW/EMPS/EMP_T[sal > 15000]') = 1;
```

This query becomes something like the following:

```
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 18–23 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 DDL statements. For example, consider a database that 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 view 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(OBJECT_VALUE) FROM employee_ov;
```

Queries on sys_XMLGen views are rewritten to access the object attributes directly if they meet certain conditions. Simple XPath traversals with SQL functions existsNode, extractValue, and extract are candidates for rewrite. See Chapter 6, "XPath Rewrite", for details on XPath rewrite. For example, a query such as the following:

```
SELECT extract(OBJECT_VALUE, '/ROW/EMPNO')
  FROM employee_ov_xml
  WHERE extractValue(OBJECT_VALUE, '/ROW/ENAME') = 'Smith';
```

This query is rewritten to something like the following:

```
SELECT sys_XMLGen(e.employee_id)
  FROM employees e
  WHERE e.last_name = 'Smith';
```

XML-Schema-Based Views Using Object Types or Object Views

Example 18–24 illustrates XPath rewrite on an XML-schema-based XMLType view using an object type.

Example 18–24 XML-Schema-Based Views Constructed Using Object Types

This example uses the same object types andXML schema (`emp_complex.xsd`) as in section "Creating Schema-Based XMLType Views Over Object Views".

```
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 SQL function `extractValue`:

```
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;
```

This query 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 SQL function `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-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 SQL functions to optionally accept an XML schema URL and element name:

- `createXML`
- `sys_XMLGen`
- `sys_XMLAgg`

See Also: Chapter 16, "Generating XML Data from the Database"

Example 18–25 Generating XML Schema-Based XML Without Creating Views

This example uses the same type and XML schema definitions as in section "Wrapping Relational Department Data with Nested Employee Data as XML". 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.

19

Accessing Data Through URIs

This chapter describes how to generate and store URLs in the database and how to retrieve data pointed to by those URLs. Three kinds of URIs are discussed:

- DBUris – addresses to relational data in the database
- XDBUris – addresses to data in Oracle XML DB Repository
- HTTPUris – Web addresses that use the Hyper Text Transfer Protocol (HTTP(S))

This chapter contains these topics:

Overview of Oracle XML DB URL Features

The two main features described in this chapter are these:

- ***Using paths as an indirection mechanism*** – You can store a path in the database and then access its target *indirectly* by referring to the path. The paths in question are various kinds of Uniform Resource Identifier (URI).
- ***Using paths that target database data to produce XML documents*** – One kind of URI that you can use for indirection in particular, a *DBUri*, provides a convenient XPath notation for addressing *database data*. You can use a DBUri to construct an *XML document* that contains database data and whose structure reflects the database structure.

URIs and URLs

In developing Web-based XML applications, you often refer to data located on a network using **Uniform Resource Identifiers**, or **URI**s. A **URL**, or **Uniform Resource Locator**, is a URI that accesses an object using an Internet protocol.

A URI has two parts, separated by a number sign (**#**):

- A URL part, that identifies a document.
- A fragment part, that identifies a fragment within the document. The notation for the fragment depends on the document type. For HTML documents, it is an anchor name. For XML documents, it is an XPath expression.

These are typical URIs:

- **For HTML –** `http://www.url.com/document1#some_anchor`, where `some_anchor` is a named anchor in the HTML document.
- **For XML –** `http://www.xml.com/xml_doc#/po/cust/custname`, where:
  - `http://www.xml.com/xml_doc` identifies the location of the XML document.
  - `/po/cust/custname` identifies a fragment within the document. This portion is defined by the W3C XPointer recommendation.

    See Also:

    - `http://www.w3.org/2002/ws/Activity.html` an explanation of HTTP(S) URL notation
    - `http://www.w3.org/TR/xpath` for an explanation of the XML XPath notation
    - `http://www.w3.org/TR/xptr/` for an explanation of the XML XPointer notation
    - `http://xml.coverpages.org/xmlMediaMIME.html` for a discussion of MIME types

URIType and its Subtypes

Oracle XML DB can represent paths of various kinds as database objects. These are the available path object types:

- **`HTTPURIType`** – An object of this type is called an **HTTPUri** and represents a URL that begins with **`http://`**. With `HTTPURIType`, you can create objects that represent links to remote *Web pages* (or files) and retrieve those Web pages by calling object methods. This type implements the Hyper Text Transfer Protocol (HTTP(S)) for accessing remote Web pages. `HTTPURIType` uses package `UTL_HTTP` to fetch data, so session settings for this package can also be used to influence HTTP fetches.

  See Also : "HTTPURIType Method getContentType()" on page 19-4

- **`DBURIType`** – An object of this type is called a **DBUri** and represents a URI that targets database data – a table, one or more rows, or a single column. With `DBURIType`, you can create objects that represent links to *database data*, and retrieve such data *as XML* by calling object methods. A DBUri uses a simple form of XPath expression as its URI syntax – for example, the following XPath expression is a DBUri reference to the row of database table `hr`, column `employees` where column `first_name` has value `Jack`:

  ```
  /HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]
  ```

  See Also : DBUris: Pointers to Database Data on page 19-12

- **`XDBURIType`** – An object of this type is called an **XDBUri** and represents a URI that targets a resource in Oracle XML DB Repository. With `XDBURIType`, you can create objects that represent links to *repository resources*, and retrieve all or part of any resource by calling object methods. The URI syntax for an XDBUri is a repository resource address optionally followed by an XPath expression. For example, `/public/hr/doc1.xml#/purchaseOrder/lineItem` is an XDBUri reference to the `lineItem` child element of the root element `purchaseOrder` in repository file `doc1.xml` in folder `/public/hr`.

  See Also : XDBUris: Pointers to Repository Resources on page 19-9

Each of these object types is derived from an *abstract* object type, **`URIType`**. As an abstract type, it has *no* instances (objects); only its subtypes have instances.

Type `URIType` provides the following features:

- ***Unified access to data stored inside and outside the server.*** Because you can use `URIType` values to store pointers to HTTP(S) and DBUris, you can create queries and indexes without worrying about where the data resides.
- ***Mapping of URIs in XML Documents to Database Columns.*** When an XML document is shredded to database tables and columns, any URIs contained in the document are mapped to database columns of the appropriate `URIType` subtype.

You can reference data stored in relational columns and expose it to the external world using URIs. Oracle Database provides a standard servlet, DBUriServlet, that interprets DBUris. It also provides PL/SQL package `UTL_HTTP` and Java class `java.net.URL`, which you can use to fetch URL references.

`URIType` columns can be indexed natively in Oracle Database using Oracle Text – no special datastore is needed.

See Also:

- "Creating New Subtypes of URIType using Package URIFACTORY" on page 19-19 for information on defining new `URIType` subtypes
- Chapter 4, "XMLType Operations", "Indexing XMLType Columns" on page 4-32 for information on indexing `XMLType` columns

DBUris and XDBUris – What For?

The following are typical uses of DBUris and XDBUris:

- You can reference XSLT style sheets from within database-generated Web pages. Package `DBMS_METADATA` uses DBUris to reference XSL style sheets. An XDBUri can be used to reference XSLT style sheets stored in Oracle XML DB Repository.
- You can reference HTML text, images and other data stored in the database. URLs can be used to point to data stored in database tables or in repository folders.
- You can improve performance by bypassing the Web server. Replace a global URL in your XML document with a reference to the database, and use a servlet, a DBUri, or a XDBUri to retrieve the targeted content. Using a DBUri or an XDBUri generally provides better performance than using a servlet, because you interact directly with the database rather than through a Web server.
- With a DBUri, you can access an XML document in the database without using SQL.

- Whenever a repository resource is stored in a database table to which you have access, you can use either an XDBUri or a DBUri to access its content.

  See Also: *Oracle Database PL/SQL Packages and Types Reference*, "DBMS_METADATA package"

URIType Methods

Abstract object type `URIType` includes methods that can be used with each of its subtypes. Each of these methods can be overridden by any of the subtypes. Table 19–1 lists the `URIType` methods. In addition, each of the subtypes has a constructor with the same name as the subtype.

Table 19–1 URIType Methods

URIType Method	Description
`getURL()`	Returns the URL of the `URIType` instance. Use this method instead of referencing a URL directly. `URIType` subtypes override this method to provide the correct URL. For example, `HTTPURIType` stores a URL without prefix `http://`. Method `getURL()` then prepends the prefix and returns the entire URL.
`getExternalURL()`	Similar to `getURL()`, but `getExternalURL()` escapes characters in the URL, to conform with the URL specification. For example, spaces are converted to the escaped value `%20`.
`getContentType()`	Returns the MIME content type for the URI. **HTTPUri:** The URL is followed and the MIME header examined, in order to return the actual content type. **DBUri:** The returned content type is either `text/plain` (for a scalar value) or `text/xml` (otherwise). **XDBUri:** The value of the `ContentType` metadata property of the repository resource is returned.
`getClob()`	Returns the target of the URI as a `CLOB` value. The database character set is used for encoding the data. **DBUri:** *XML* data is returned (unless node-test `text()` is used, in which case the targeted data is returned as is). When a `BLOB` column is targeted, the binary data in the column is *translated as hexadecimal character data*.
`getBlob()`	Returns the target of the URI as a `BLOB` value. No character conversion is performed, and the character encoding is that of the URI target. This method can also be used to fetch binary data. **DBUri:** When applied to a DBUri that targets a `BLOB` column, `getBlob()` returns the binary data *translated as hexadecimal character data*. When applied to a DBUri that targets *non*-binary data, the data is returned in the database character set.
`getXML()`	Returns the target of the URI as an `XMLType` instance. Using this, an application that performs operations other than `getClob()` and `getBlob()` can use `XMLType` methods to do those operations. This throws an exception if the URI does not target a well-formed XML document.
`createURI()`	Constructs an instance of one of the `URIType` subtypes.

HTTPURIType Method getContentType()

`HTTPURIType` method `getContentType()` returns the actual MIME information for its targeted document. You can use this information to decide whether to retrieve the document as a `BLOB` value or a `CLOB` value. For example, you might treat a Web page

with a MIME type of x/jpeg as a BLOB value, and one with a MIME type of text/plain or text/html as a CLOB value.

Example 19–1 Using HTTPURIType Method getContentType()

In this example, the HTTP content type is tested to determine whether to retrieve data as a CLOB or BLOB value. The content-type data is the HTTP header, for HTTPURIType, or the metadata of the database column, for DBURIType.

```
DECLARE
  httpuri HTTPURIType;
  y CLOB;
  x BLOB;
BEGIN
  httpuri := HTTPURIType('http://www.oracle.com/object1');
  DBMS_OUTPUT.put_line(httpuri.getContentType());
  IF httpuri.getContentType() = 'text/html'
  THEN
     y := httpuri.getClob();
  END IF;
  IF httpuri.getContentType() = 'application-x/bin'
  THEN
     x := httpuri.getBlob();
  END IF;
END;
/
text/html
```

DBURIType Method getContentType()

Method getContentType() returns the MIME information for a URL. If a DBUri targets a scalar value, then the MIME content type returned is text/plain; otherwise, it is text/xml. For example, consider table dbtab:

```
CREATE TABLE DBTAB(a VARCHAR2(20), b BLOB);
```

DBUris corresponding to the following XPath expressions have content type text/xml, because each targets a complete column of XML data.

- /HR/DBTAB/ROW/A
- /HR/DBTAB/ROW/B

DBUris corresponding to the following XPath expressions have content type text/plain, because each targets a scalar value.

- /HR/DBTAB/ROW/A/text()
- /HR/DBTAB/ROW/B/text()

DBURIType Method getClob()

When method getClob() is applied to a DBUri, the targeted data is returned as *XML* data, using the targeted column or table name as an XML element name. If the target XPath uses node-test text(), then the data is returned as text without an enclosing XML tag. In both cases, the returned data is in the database character set.

For example: If appliedto a DBUri with XPath /HR/DBTAB/ROW/A/text(), where A is a non-binary column, the data in column A is returned as is. Without XPath node-test text(), the result is the data wrapped in XML:

```
<HR><DBTAB><ROW><A>...data_in_column_A...</A></ROW></DBTAB></HR>
```

When applied toa DBUri that targets a *binary* (BLOB) column, the binary data in the column is *translated as hexadecimal character data*.

For example: If applied to a DBUri with XPath /HR/DBTAB/ROW/B/text(), where B is a BLOB column, the targeted binary data is translated to hexadecimal character data and returned. Without XPath node-test text(), the result is the translated data wrapped in XML:

```
<HR><DBTAB><ROW><B>...data_translated_to_hex...</B></ROW></DBTAB></HR>
```

DBURIType Method getBlob()

When applied to a DBUri that targets a BLOB column, getBlob() returns the binary data *translated as hexadecimal character data*. When applied to a DBUri that targets *non*-binary data, getBlob() returns the data (as a BLOB value) in the database character set.

For example, consider table dbtab:

```
CREATE TABLE DBTAB(a VARCHAR2(20), b BLOB);
```

When getBlob() is applied to a DBUri corresponding to XPath expression /HR/DBTAB/ROW/B, it returns a BLOB value containing an XML document with root element B whose content is the hexadecimal-character translation of the binary data of column B.

When getBlob() is applied to a DBUri corresponding to XPath expression /HR/DBTAB/ROW/B/text(), it returns a BLOB value containing only the hexadecimal-character translation of the binary data of column B.

When getBlob() is applied to a DBUri corresponding to XPath expression /HR/DBTAB/ROW/**A**/text(), which targets *non*-binary data, it returns a BLOB value containing the data of column A, in the database character set.

Accessing Data Using URIType Instances

To use instances of URIType subtypes for indirection, you generally store such instances in the database and then use them in queries with a method such as getClob() to retrieve the targeted data. This section illustrates how to do this.

You can create database columns using URIType or any of its subtypes, or you can store just the text of each URI as a string and then create the needed URIType instances on demand, when the URIs are accessed. You can store objects of different URIType subtypes in the same URIType database column.

You can also define your own object types that inherit from the URIType subtypes. Deriving new types lets you use custom techniques to retrieve, transform, or filter data.

See Also:

- "Creating New Subtypes of URIType using Package URIFACTORY" on page 19-19 for information on defining new URIType subtypes
- "XSL Transformation and Oracle XML DB" on page 3-70 for information on transforming XML data

Example 19–2 Creating and Querying a URI Column

This example stores an HTTPUri and a DBUri (instances of `URIType` subtypes `HTTPURIType` and `DBURIType`) in the same database column of type `URIType`. A query retrieves the data addressed by each of the URIs. The first URI is a Web-page URL; the second references data in the `employees` table of standard schema `hr`. (For brevity, only the beginning of the Web page is shown.)

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES (HTTPURIType.createURI('http://www.oracle.com'));
1 row created.

INSERT INTO uri_tab VALUES (DBURIType.createURI(
                              '/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]'));
1 row created.

SELECT e.url.getClob() FROM uri_tab e;

E.URL.GETCLOB()
--------------------------------------------------------------------
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Oracle Corporation</TITLE>
. . .

<?xml version="1.0"?>
 <ROW>
  <EMPLOYEE_ID>177</EMPLOYEE_ID>
  <FIRST_NAME>Jack</FIRST_NAME>
  <LAST_NAME>Livingston</LAST_NAME>
  <EMAIL>JLIVINGS</EMAIL>
  <PHONE_NUMBER>011.44.1644.429264</PHONE_NUMBER>
  <HIRE_DATE>23-APR-98</HIRE_DATE>
  <JOB_ID>SA_REP</JOB_ID>
  <SALARY>8400</SALARY>
  <COMMISSION_PCT>.2</COMMISSION_PCT>
  <MANAGER_ID>149</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>

2 rows selected.
```

In order to use `URIType` method `createURI()`, you must know the particular `URIType` subtype to use. Method `getURI()` of package `URIFACTORY` lets you instead use the flexibility of late binding, determining the particular type information at runtime.

`URIFACTORY.getURI()` takes as argument a URI string; it returns a `URIType` instance of the appropriate subtype (`HTTPURIType, DBURIType,` or `XDBURIType`), based on the form of the URI string:

- If the URI starts with `http://`, then `getURI()` creates and returns an HTTPUri.
- If the URI starts with either `/oradb/` or `/dburi/`, then `getURI()` creates and returns a DBUri.
- Otherwise, `getURI()` creates and returns an XDBUri.

Example 19–3 Using Different Kinds of URI, Created in Different Ways

This example is similar to Example 19–2. However, it uses two different ways to obtain documents targeted by URIs:

- Method `SYS.URIFACTORY.getURI()` with *absolute* URIs:
  - an HTTPUri that targets HTTP address `http://www.oracle.com`
  - a DBUri that targets database address `/oradb/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]`
- Constructor `SYS.HTTPURIType()` with a *relative* URL (no `http://`). The same HTTPUri is used as for the absolute URI: the Oracle home page.

In this example, the URI strings passed to `getURI()` are hard-coded, but they could just as easily be string values that are obtained by an application at runtime.

```
CREATE TABLE uri_tab (docUrl SYS.URIType, docName VARCHAR2(200));
Table created.

-- Insert an HTTPUri with absolute URL into SYS.URIType using URIFACTORY.
-- The target is Oracle home page.
INSERT INTO uri_tab VALUES
  (SYS.URIFACTORY.getURI('http://www.oracle.com'), 'AbsURL');
1 row created.

-- Insert an HTTPUri with relative URL using constructor SYS.HTTPURIType.
-- Note the absence of prefix http://. The target is the same.
INSERT INTO uri_tab VALUES (SYS.HTTPURIType('www.oracle.com'), 'RelURL');
1 row created.

-- Insert a DBUri that targets employee data from database table hr.employees.
INSERT INTO uri_tab VALUES
  (SYS.URIFACTORY.getURI('/oradb/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]'), 'Emp200');
1 row created.

-- Extract all of the documents.
SELECT e.docUrl.getClob(), docName FROM uri_tab e;

E.DOCURL.GETCLOB()
------------------
DOCNAME
------------------------------------
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Oracle Corporation</TITLE>
. . .
AbsURL

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Oracle Corporation</TITLE>
. . .
RelURL

<?xml version="1.0"?>
 <ROW>
  <EMPLOYEE_ID>200</EMPLOYEE_ID>
  <FIRST_NAME>Jennifer</FIRST_NAME>
  <LAST_NAME>Whalen</LAST_NAME>
```

```
  <EMAIL>JWHALEN</EMAIL>
  <PHONE_NUMBER>515.123.4444</PHONE_NUMBER>
  <HIRE_DATE>17-SEP-87</HIRE_DATE>
  <JOB_ID>AD_ASST</JOB_ID>
  <SALARY>4400</SALARY>
  <MANAGER_ID>101</MANAGER_ID>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
 </ROW>
Emp200

3 rows selected.

-- In PL/SQL
CREATE OR REPLACE FUNCTION returnclob
RETURN CLOB
IS
  a SYS.URIType;
BEGIN
  SELECT docUrl INTO a FROM uri_Tab WHERE docName LIKE 'Emp200%';
  RETURN a.getClob;
END;
/
Function created.

SELECT returnclob() FROM DUAL;

RETURNCLOB()
--------------------------------------------------------------
<?xml version="1.0"?>
 <ROW>
  <EMPLOYEE_ID>200</EMPLOYEE_ID>
  <FIRST_NAME>Jennifer</FIRST_NAME>
  <LAST_NAME>Whalen</LAST_NAME>
  <EMAIL>JWHALEN</EMAIL>
  <PHONE_NUMBER>515.123.4444</PHONE_NUMBER>
  <HIRE_DATE>17-SEP-87</HIRE_DATE>
  <JOB_ID>AD_ASST</JOB_ID>
  <SALARY>4400</SALARY>
  <MANAGER_ID>101</MANAGER_ID>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
 </ROW>

1 row selected.
```

XDBUris: Pointers to Repository Resources

`XDBURIType` is a subtype of `URIType` that provides a way to expose resources in Oracle XML DB Repository using URIs. Instances of type `XDBURIType` are called **XDBUris**.

XDBUri URI Syntax

The URL portion of an XDBUri URI is the hierarchical address of the targeted repository resource – it is a *repository* path (*not* an XPath expression).

The optional fragment portion of the URI uses the XPath syntax, and is separated from the URL part by a number-sign (#). It is appropriate only if the targeted resource is an *XML* document, in which case the fragment portion targets one or more parts of the

XML document. If the targeted resource is not an XML document, then omit the fragment and number-sign.

The following are examples of XDBUri URIs:

- `/public/hr/image27.jpg`
- `/public/hr/doc1.xml#/PurchaseOrder/LineItem`

Based on the form of these URIs, we can determine the following:

- `/public/hr` is a folder resource in Oracle XML DB Repository.
- `image27.jpg` and `doc1.xml` are resources in folder `/public/hr`.
- Resource `doc1.xml` is a file resource, and it contains an XML document.
- The XPath expression `/PurchaseOrder/LineItem` refers to the `LineItem` child element in element `PurchaseOrder` of XML document `doc1.xml`.

You can create an XDBUri using method `getURI()` of package `URIFACTORY`.

`XDBURIType` is the *default* `URIType` used when generating instances using `URIFACTORY` method `getURI()`, unless the URI has one of the recognized prefixes `http://`, `/dburi`, or `/oradb`.

For example, if resource `doc1.xml` is present in repository folder `/public/hr`, then the following query will return an XDBUri that targets that resource.

```
SELECT SYS.URIFACTORY.getURI('/public/hr/doc1.xml') FROM DUAL;
```

It is the lack of a special prefix that determines that the type is `XDBURIType`, not any particular resource file extension or the presence of # followed by an XPath expression; if the resource were named `foo.bar` instead of `doc1.xml`, the returned `URIType` instance would still be an XDBUri.

XDBUri Examples

Example 19–4 Using an XDBUri to Access a Repository Resource by URI

This example creates an XDBUri, inserts values into a purchase-order table, and then selects all of the purchase orders. Because there is no special prefix used in the URI passed to `URIFACTORY.getURI()`, the created `URIType` instance is an XDBUri.

```
DECLARE
res BOOLEAN;
postring VARCHAR2(100):= '<?xml version="1.0"?>
<ROW>
<PO>999</PO>
</ROW>';
BEGIN
res:=DBMS_XDB.createFolder('/public/orders/');
res:=DBMS_XDB.createResource('/public/orders/po1.xml', postring);
END;
/
PL/SQL procedure successfully completed.

CREATE TABLE uri_tab (poUrl SYS.URIType, poName VARCHAR2(1000));
Table created.

-- 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 is no prefix.
```

```
-- Here, po1.xml is an XML file that is stored in /public/orders/
-- of the XML repository.
INSERT INTO uri_tab VALUES
  (URIFACTORY.getURI('/public/orders/po1.xml'), 'SomePurchaseOrder');
1 row created.

-- Get all the purchase orders
SELECT e.poUrl.getClob(), poName FROM uri_tab e;

E.POURL.GETCLOB()
-----------------
PONAME
------
<?xml version="1.0"?>
<ROW>
<PO>999</PO>
</ROW>
SomePurchaseOrder

1 row selected.

-- Using PL/SQL, you can access table uri_tab as follows:
CREATE OR REPLACE FUNCTION returnclob
RETURN CLOB
IS
  a URIType;
BEGIN
  -- Get absolute URL for purchase order named like 'Some%'
   SELECT poUrl INTO a FROM uri_tab WHERE poName LIKE 'Some%';
   RETURN a.getClob();
END;
/
Function created.

SELECT returnclob() FROM DUAL;

RETURNCLOB()
--------------------
<?xml version="1.0"?>
<ROW>
<PO>999</PO>
</ROW>

1 row selected.
```

Example 19–5 Using getXML() with EXTRACTVALUE

Because method `getXML()` returns an `XMLType` instance, you can use it with SQL functions like `extractValue`. This query retrieves all purchase orders numbered 999:

```
SELECT e.poUrl.getClob() FROM uri_tab e
   WHERE extractValue(e.poUrl.getXML(), '/ROW/PO') = '999';

E.POURL.GETCLOB()
--------------------
<?xml version="1.0"?>
<ROW>
<PO>999</PO>
</ROW>

1 row selected.
```

DBUris: Pointers to Database Data

A DBUri is a URI that targets *database data*. As for all instances of `URIType` subtypes, a DBUri provides an indirection mechanism for accessing data. In addition, `DBURIType` lets you do the following:

- Address database data using XPath notation. This, in effect, lets you visualize and access the database as if it were XML data.

  For example, a DBUri can use an expression such as `/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]` to target the row of database table `hr`, column `employees` where column `first_name` has value `Jack`.

- *Construct an XML document* that contains database data targeted by a DBUri and whose structure reflects the database structure.

  For example: A DBUri with XPath `/HR/DBTAB/ROW/A` can be used to construct an XML document that wraps the data of column `A` in XML elements that reflect the database structure and are named accordingly:

  ```
  <HR><DBTAB><ROW><A>...data_in_column_A...</A></ROW></DBTAB></HR>
  ```

A DBUri does not reference a global location as does an HTTPUri. You can, however, also access objects addressed by a DBUri in a global manner, by appending the DBUri to an HTTPUri that identifies a servlet that handles DBUris – see "DBUriServlet" on page 19-25.

Viewing the Database as XML Data

You can only access those database schemas to which you have been granted access privileges. This portion of the database is, in effect, your own view of the database.

Using `DBURIType`, you can have corresponding *XML* views of the database, which are portions of the database to which you have access, presented *in the form of XML data*. This means all kinds database data, not just data that is stored as XML. When visualized this way, the database data is effectively wrapped in XML elements, resulting in one or more XML documents.

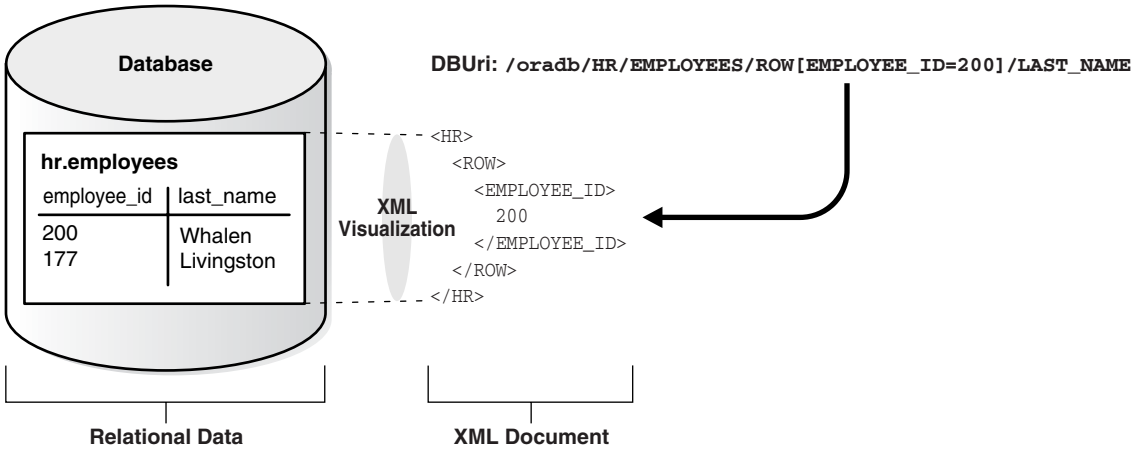
Such "XML views" are not database views, in the technical sense of the term; "view" here means only an abstract perspective that can be useful for understanding `DBURIType`. You can think of `DBURIType` as providing a way to visualize and access the database *as if it were* XML data.

However, `DBURIType` does not just provide an exercise in visualization and an additional means to access database data. Each "XML view" can be realized as an XML document – that is, you can use `DBURIType` to generate XML documents using database data.

All of this is another way of saying that `DBURIType` lets you use XPath notation to 1) address and access any database data to which you have access and 2) construct XML representations of that data.

Figure 19–1 illustrates the relation between a relational table, `hr.employees`, a corresponding "XML view" of a portion of that table, and the corresponding DBUri URI (a simple XPath expression). In this case, the portion of the data exposed as XML is the row where `employee_id` is `200`. The URI can be used to access the data and construct an XML document that reflects the "XML view".

Figure 19–1 A DBUri Corresponds to an XML Vizualization of Relational Data



The XML elements in the "XML view" and the steps in the URI XPath expression both reflect the database table and column names. Note the use of **`ROW`** to indicate a row in the database table – both in the "XML view" and in the URI XPath expression.

Note also that the XPath expression contains a root-element step, **`oradb`**. This is used to indicate that the URI corresponds to a DBUri, not an HTTPUri or an XDBUri. Whenever this correspondence is understood from context, this XPath step can be skipped. For example, if it is known that the path in question is a path to database data, the following URIs are equivalent:

- **`/oradb`**`/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME`
- `/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME`

Whenever the URI context is not clear, however, you must use the prefix `/oradb` to distinguish a URI as corresponding to a DBUri. In particular, you must supply the prefix to `URIFACTORY` methods and to DBUriServlet.

See Also:

- "Creating New Subtypes of URIType using Package URIFACTORY" on page 19-19
- "DBUriServlet" on page 19-25
- Chapter 16, "Generating XML Data from the Database" for other ways to generate XML from database data

DBUri URI Syntax

An XPath expression is a path into XML data that addresses one or more XML nodes. A DBUri exploits the notion of a virtual XML user visualization of the database to use a *simple form* of XPath expression as a URI to address database data. This is so, regardless of the type of data, in particular, whether or not the data is XML.

Thus, for `DBURIType`, Oracle Database does not support the full XPath or XPointer syntax; only a subset is allowed. There are no syntax restrictions for XDBUri XPath expressions. There is also an exception in the DBUri case: data in `XMLType` tables. For an `XMLType` table, the simple XPath form is used to address the table itself within the database. Then, to address particular XML data in the table, the remainder of the XPath expression can use the full XPath syntax. This exception applies only to `XMLType` *tables*, not to `XMLType` columns.

In any case, unlike an XDBUri, a DBUri URI does not use a number-sign (#) to separate the URL portion of a URI from a fragment (XPath) portion. DBURIType does not use URI fragments; the entire URI is treated as a (simple) XPath expression.

You can create DBUris to any database data to which you have access. XPaths such as the following are allowed:

- */database_schema/table*
- */database_schema/table*/ROW[*predicate_expression*]/*column*
- */database_schema/table*/ROW[*predicate_expression*]/*object_column/attribute*
- */database_schema/**XMLType_table***/ROW/*XPath_expression*

In the last case, *XMLType_table* is an XMLType table, and *XPath_expression* is *any* XPath expression. For tables that are *not* XMLType, a DBUri XPath expression must end at a column; it cannot address specific data inside a column. This restriction includes XMLType columns, LOB columns, and VARCHAR2 columns that contain XML data.

A DBUri XPath expression can do any of the following:

- Target an entire table.

  For example, /HR/EMPLOYEES targets table employees of database schema hr.

- Include XPath predicates at any step in the path, except the database schema and table steps.

  For example, /HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/EMAIL targets the email column of table hr.employees, where employee_id is 200.

- Use the text() XPath node test on data with scalar content. This is the *only* node test that can be used, and it cannot be used with the table or row step.

The following can be used in DBUri (XPath) *predicate* expressions:

- Boolean operators and, or, and not
- Relational operators <, >, <=, !=, >=, =, mod, div, * (multiply)

A DBUri XPath expression *must* do all of the following:

- Use only the *child* XPath axis – other axes, such as parent, are not allowed.
- Either specify a database schema or specify PUBLIC to resolve the table name without a specific schema.
- Specify a database view or table name.
- Include a ROW step, if a database column is targeted.
- Identify a *single* data value, which can be an object-type instance or a collection.
- Result in well-formed XML when it is used to generate XML data using database data.

  An example of a DBUri that does *not* result in well-formed XML is /HR/EMPLOYEES/ROW/LAST_NAME. It returns more than one <LAST_NAME> element fragment, with no single root element.

- Use *none* of the following:
  - * (wildcard)
  - . (self)

- `..` (parent)
- `//` (descendent or self)
- XPath functions, such as `count`

A DBUri XPath expression can optionally be prefixed by **`/oradb`** or **`/dburi`** (the two are equivalent) to distinguish it. This prefix is case-insensitive. However, the rest of the DBUri XPath expression is *case-sensitive*, as are XPaths generally. Thus, for example, to specify database column `hr.employees` as a DBUri XPath expression, you must use `HR/EMPLOYEES`, not `hr/employees` (or a mixed-case combination), because table and column names are uppercase, by default.

See Also: `http://www.w3.org/TR/xpath` on XPath notation

DBUris are Scoped to a Database and Session

The content of the "XML views" you have of the database, and hence of the XML documents that you can construct, reflects the permissions you have to access particular database data at a given time. That is, a DBUri is scoped to a given database session, so the same DBUri can give different results in the same query, depending on the session context (which user is connected and what privileges the user has).

To complicate things a bit, there is also an XML element `PUBLIC`, under which database data is accessible without any database-schema qualification. This is a convenience feature, but it can also lead to some confusion if you forget that the XML views of the database for a given user depend on the specific access the user has to the database at a given time.

XML element `PUBLIC` corresponds to the use of a *public synonym*. For example, when queried by user `quine`, the following query tries to match table `foo` under database schema `quine`, but if no such table exists, it tries to match a public synonym named `foo`.

```
SELECT * FROM foo;
```

In the same way, XML element `PUBLIC` contains all of the database data visible to a given user, as well as all of the data visible to that user through public synonyms. So, the same DBUri URI `/PUBLIC/FOO` can resolve to `quine.foo` when user `quine` is connected, and resolve to `curry.foo` when user `curry` is connected.

DBUri Examples

A DBUri can identify a table, a row, a column in a row, or an attribute of an object column. The following sections describe how to target different object types.

Targeting a Table

You can target a complete database table, using this syntax:

```
/database_schema/table
```

Example 19–6 Using a DBUri to Target a Complete Table

In this example, a DBUri targets a complete table. An XML document is returned that corresponds to the table contents. The top-level XML element is named for the table. The values of each row are enclosed in a `ROW` element.

```
CREATE TABLE uri_tab (url URIType);
Table created.
```

```
INSERT INTO uri_tab VALUES
       (DBURIType.createURI('/HR/EMPLOYEES'));
1 row created.

SELECT e.url.getClob() FROM uri_tab e;

E.URL.GETCLOB()
---------------
<?xml version="1.0"?>
<EMPLOYEES>
 <ROW>
  <EMPLOYEE_ID>100</EMPLOYEE_ID>
  <FIRST_NAME>Steven</FIRST_NAME>
  <LAST_NAME>King</LAST_NAME>
  <EMAIL>SKING</EMAIL>
  <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
  <HIRE_DATE>17-JUN-87</HIRE_DATE>
  <JOB_ID>AD_PRES</JOB_ID>
  <SALARY>24000</SALARY>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>101</EMPLOYEE_ID>
  <FIRST_NAME>Neena</FIRST_NAME>
  <LAST_NAME>Kochhar</LAST_NAME>
  <EMAIL>NKOCHHAR</EMAIL>
  <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
  <HIRE_DATE>21-SEP-89</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 . . .

1 row selected.
```

Targeting a Row in a Table

You can target one or more specific rows of a table, using this syntax:

`/database_schema/table/ROW[predicate_expression]`

Example 19–7 Using a DBUri to Target a Particular Row in a Table

In this example, a DBUri targets a single table row. The XPath predicate expression identifies the single table row that corresponds to employee number 200. The result is an XML document with `ROW` as the top-level element.

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
       (DBURIType.createURI('/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]'));
1 row created.

SELECT e.url.getClob() FROM uri_tab e;

E.URL.GETCLOB()
--------------------------------------------------------
<?xml version="1.0"?>
```

```
 <ROW>
  <EMPLOYEE_ID>200</EMPLOYEE_ID>
  <FIRST_NAME>Jennifer</FIRST_NAME>
  <LAST_NAME>Whalen</LAST_NAME>
  <EMAIL>JWHALEN</EMAIL>
  <PHONE_NUMBER>515.123.4444</PHONE_NUMBER>
  <HIRE_DATE>17-SEP-87</HIRE_DATE>
  <JOB_ID>AD_ASST</JOB_ID>
  <SALARY>4400</SALARY>
  <MANAGER_ID>101</MANAGER_ID>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
 </ROW>

1 row selected.
```

Targeting a Column

You can target a specific column, using this syntax:

/database_schema/table/ROW[*predicate_expression*]/*column*

You can target a specific attribute of an object column, using this syntax:

/database_schema/table/ROW[*predicate_expression*]/*object_column/attribute*

You can target a specific object column whose attributes have specific values, using this syntax:

/database_schema/table/ROW[*predicate_expression_with_attributes*]/*object_column*

Example 19–8 Using a DBUri to Target a Specific Column

In this example, a DBUri targets column `last_name` for the same employee as in Example 19–7. The top-level XML element is named for the targeted column.

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
       (DBURIType.createURI('/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME'));
1 row created.

SELECT e.url.getClob() FROM uri_tab e;

E.URL.GETCLOB()
------------------------------
<?xml version="1.0"?>
 <LAST_NAME>Whalen</LAST_NAME>

1 row selected.
```

Example 19–9 Using a DBUri to Target an Object Column with Specific Attribute Values

In this example, a DBUri targets a `CUST_ADDRESS` object column containing city and postal code attributes with certain values. The top-level XML element is named for the column, and it contains child elements for each of the object attributes.

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
```

```
  (DBURIType.createURI(
     '/OE/CUSTOMERS/ROW[CUST_ADDRESS/CITY="Poughkeepsie" and
                        CUST_ADDRESS/POSTAL_CODE=12601]/CUST_ADDRESS'));
1 row created.

SELECT e.url.getClob() FROM uri_tab e;

E.URL.GETCLOB()
---------------
<?xml version="1.0"?>
 <CUST_ADDRESS>
  <STREET_ADDRESS>33 Fulton St</STREET_ADDRESS>
  <POSTAL_CODE>12601</POSTAL_CODE>
  <CITY>Poughkeepsie</CITY>
  <STATE_PROVINCE>NY</STATE_PROVINCE>
  <COUNTRY_ID>US</COUNTRY_ID>
 </CUST_ADDRESS>

1 row selected.
```

The DBUri identifies the object that has a `CITY` attribute with `Poughkeepsie` as value and a `POSTAL_CODE` attribute with `12601` as value.

Retrieving the Text Value of a Column

In many cases, it can be useful to retrieve only the text values of a column and not the enclosing tags. For example, if XSLT style sheets are stored in a `CLOB` column, you can retrieve the document text without having any enclosing column-name tags. You can use the `text()` XPath node test for this. It specifies that you want only the text value of the node. Use the following syntax:

`/oradb/`*`database_schema`*`/`*`table`*`/ROW[`*`predicate_expression`*`]/`*`column`*`/`**`text()`**

Example 19–10 Using a DBUri to Retrieve Only the Text Value of a Node

This example retrieves the text value of the employee `last_name` column for employee number 200, , without the XML tags.

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
       (DBURIType.createURI(
          '/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME/text()'));

1 row created.

SELECT e.url.getClob() FROM uri_tab e;

E.URL.GETCLOB()
---------------
Whalen

1 row selected.
```

Targeting a Collection

You can target a database collection, such as a varray or nested table. You must, however, target the entire collection – you cannot target individual members of a collection. When a collection is targeted, the XML document produced by the DBUri

contains each collection member as an XML element, with all such elements enclosed in a element named for the *type* of the collection.

Example 19–11 Using a DBUri to Target a Collection

In this example, a DBUri targets a collection of numbers. The top-level XML element is named for the collection, and its children are named for the collection *type* (`NUMBER`).

```
CREATE TYPE num_collection AS VARRAY(10) OF NUMBER;
/
Type created.

CREATE TABLE orders (item VARCHAR2(10), quantities num_collection);
Table created.

INSERT INTO orders VALUES ('boxes', num_collection(3, 7, 4, 9));
1 row created.

SELECT * FROM orders;

ITEM
----
QUANTITIES
----------
boxes
NUM_COLLECTION(3, 7, 4, 9)

1 row selected.

SELECT DBURIType('/HR/ORDERS/ROW[ITEM="boxes"]/QUANTITIES').getClob() FROM DUAL;

DBURITYPE('/HR/ORDERS/ROW[ITEM="BOXES"]/QUANTITIES').GETCLOB()
--------------------------------------------------------------
<?xml version="1.0"?>
 <QUANTITIES>
  <NUMBER>3</NUMBER>
  <NUMBER>7</NUMBER>
  <NUMBER>4</NUMBER>
  <NUMBER>9</NUMBER>
 </QUANTITIES>

1 row selected.
```

Creating New Subtypes of URIType using Package URIFACTORY

You can use PL/SQL package `URIFACTORY` to do more than create `URIType` instances. Additional methods are listed in Table 19–2.

Table 19–2 URIFACTORY Methods

Method	Description
`getURI()`	Returns the URL of the `URIType` instance.
`escapeURI()`	Escapes the URI string by replacing characters that are not permitted in URIs by their equivalent escape sequence.

Table 19–2 (Cont.) URIFACTORY Methods

Method	Description
`unescapeURI()`	Unescapes a given URI.
`registerURLHandler()`	Registers a particular type name for handling a particular URL. This is called by `getURI()` to generate an instance of the type. A Boolean argument can be used to indicate that the prefix must be stripped off before calling the appropriate type constructor.
`unregisterURLHandler()`	Unregisters a URL handler.

Of particular note is that you can use package `URIFACTORY` to define new subtypes of type `URIType`. You can then use those subtypes to provide specialized processing of URIs. In particular, you can define `URIType` subtypes that correspond to particular protocols – `URIFACTORY` will then recognize and process instances of those subtypes accordingly.

Defining new types and creating database columns specific to the new types has these advantages:

- It provides an implicit *constraint* on the columns to contain only instances of those types. This can be useful for implementing specialized indexes on a column for specific protocols. For a DBUri, for instance, you can implement specialized indexes that fetch data directly from disk blocks, rather than executing SQL queries.
- You can have different constraints on different columns, based on the type. For a HTTPUri, for instance, you can define proxy and firewall constraints on a column, so that any access through the HTTP uses the proxy server.

Registering New URIType Subtypes with Package URIFACTORY

To provide specialized processing of URIs, you define and register a new URIType subtype, as follows:

1. Create the new type using SQL statement `CREATE TYPE`. The type must implement method `createURI()`.
2. Optionally override the default methods, to perform specialized processing when retrieving data or to transform the XML data before displaying it.
3. Choose a new URI prefix, to identify URIs that use this specialized processing.
4. Register the new prefix using method `registerURLHandler()`, so that package `URIFACTORY` can create an instance of your new subtype when it receives a URI starting with the new prefix you defined.

After the new subtype is defined, a URI with the new prefix will be recognized by `URIFACTORY` methods, and you can create and use instances of the new type.

For example, suppose that you define a new protocol prefix, `ecom://`, and define a subtype of `URIType` to handle it. Perhaps the new subtype implements some special logic for method `getCLOB()`, or perhaps it makes some changes to XML tags or data in method `getXML()`. After you register prefix `ecom://` with `URIFACTORY`, a call to `getURI()` will generate an instance of the new `URIType` subtype for a URI with that prefix.

Example 19–12 URIFACTORY: Registering the ECOM Protocol

This example creates a new type, `ECOMURIType`, to handle a new protocol, `ecom://`. The example stores three different kinds of URIs in a single table: an HTTPUri, a DBUri, and an instance of the new type, `ECOMURIType`. To actually run this example, you would need to define each of the `ECOMURIType` member functions.

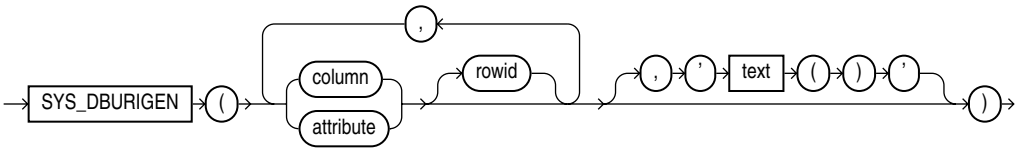
```
CREATE TABLE url_tab (urlcol varchar2(80));
Table created.

-- Insert an HTTP URL reference
INSERT INTO url_tab VALUES ('http://www.oracle.com/');
1 row created.

-- Insert a DBUri
INSERT INTO url_tab VALUES ('/oradb/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]');
1 row created.

-- Create a new type to handle a new protocol called ecom://
-- This is just an example template. For this to run, the implementations
-- of these functions needs to be specified.
CREATE OR REPLACE TYPE ECOMURIType UNDER SYS.URIType (
  OVERRIDING MEMBER FUNCTION getClob RETURN CLOB,
  OVERRIDING MEMBER FUNCTION getBlob RETURN BLOB,
  OVERRIDING MEMBER FUNCTION getExternalURL RETURN VARCHAR2,
  OVERRIDING MEMBER FUNCTION getURI RETURN VARCHAR2,
  -- Must have this for registering with the URL handler
  STATIC FUNCTION createURI(url IN VARCHAR2) RETURN ECOMURIType);
/
-- Register a new handler for the ecom:// prefixes
BEGIN
  -- The handler type name is ECOMURIType; schema is HR
  -- Ignore the prefix case, so that URIFACTORY creates the same subtype
  -- for URIs beginning with ECOM://, ecom://, eCom://, and so on.
  -- Strip the prefix before calling method createURI(),
  -- so that the string 'ecom://' is not stored inside the
  -- ECOMURIType object. It is added back automatically when
  -- you call ECOMURIType.getURI().
  URIFACTORY.registerURLHandler (prefix => 'ecom://',
                                 schemaname => 'HR',
                                 typename => 'ECOMURITYPE',
                                 ignoreprefixcase => TRUE,
                                 stripprefix => TRUE);
END;
/
PL/SQL procedure successfully completed.

-- Insert this new type of URI into the table
INSERT INTO url_tab VALUES ('ECOM://company1/company2=22/comp');
1 row created.

-- Use the factory to generate an instance of the appropriate
-- subtype for each URI in the table.

-- You would need to define the member functions for this to work:
SELECT urifactory.getURI(urlcol) FROM url_tab;

-- This would generate:
HTTPURIType('www.oracle.com'); -- an HTTPUri
DBURIType('/oradb/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]', null); -- a DBUri
ECOMURIType('company1/company2=22/comp'); -- an ECOMURIType instance
```

SYS_DBURIGEN SQL Function

You can create a DBUri by providing an XPath expression to constructor `DBURIType` or to appropriate `URIFACTORY` methods. With SQL function `sys_DburiGen`, you can alternatively create a DBUri with an *XPath that is composed* from database columns and their values.

SQL function `sys_DburiGen` takes as its argument one or more database columns or attributes, and optionally a rowid, and generates a DBUri that targets a particular column or row object. Function `sys_DburiGen` takes an additional parameter that indicates whether the text value of the node is needed. See Figure 19–2.

Figure 19–2 SYS_DBURIGEN Syntax



All columns or attributes referenced must reside in the same table. They must each reference a unique value. If you specify multiple columns, then the initial columns identify the row, and the last column identifies the column within that row. If you do not specify a database schema, then the table name is interpreted as a public synonym.

See Also: *Oracle Database SQL Reference*

Example 19–13 SYS_DBURIGEN: Generating a DBUri that Targets a Column

This example uses SQL function `sys_DburiGen` to generate a DBUri that targets column `email` of table `hr.employees` where `employee_id` is `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)

1 row selected.
```

Rules for Passing Columns or Object Attributes to SYS_DBURIGEN

A column or attribute passed to SQL function `sys_DburiGen` must obey the following rules:

- **Same table:** All columns referenced in function `sys_DburiGen` must come from the same table or view.
- **Unique mapping:** The column or object attribute must be uniquely mappable back to the table or view from which it came. The only virtual columns allowed are those produced with `VALUE` or `REF`. The column can come from a subquery with SQL function `table` or from 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.

- **`PUBLIC` element:** If the table or view targeted by the rowid or key columns does not specify a database schema, then the `PUBLIC` keyword is used. When a DBUri is accessed, the table name resolves to the same table, synonym, or database view that was visible by that name when the DBUri was created.
- **Optional `text()` argument:** By default, `DBURIType` constructs an XML document. Use `text()` as the third argument to `sys_DburiGen` to create a DBUri that targets a text node (no XML elements). For example:

  ```
  SELECT sys_DburiGen(employee_id, last_name, 'text()') FROM hr.employees,
    WHERE employee_id=200;
  ```

  This will construct a DBUri with the following URI:

  ```
  /HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME/text()
  ```

- **Single-column argument:** If there is a single-column argument, then the column is used as both the key column to identify the row and the referenced column.

Example 19–14 Passing Columns With Single Arguments to SYS_DBURIGEN

This query uses `employee_id` as both the key column and the referenced column. It generates a DBUri that targets the row with `employee_id` 7369.

```
SELECT sys_DburiGen(employee_id) FROM employees
  WHERE employee_id=200;

SYS_DBURIGEN(EMPLOYEE_ID)(URL, SPARE)
-------------------------------------
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID=''200'']/EMPLOYEE_ID', NULL)

1 row selected.
```

SYS_DBURIGEN SQL Function: Examples

Example 19–15 Inserting Database References Using SYS_DBURIGEN

```
CREATE TABLE doc_list_tab(docno NUMBER PRIMARY KEY, doc_ref SYS.DBURIType);
Table created.

-- Insert a DBUri that targets the row with employee_id=177
INSERT INTO doc_list_tab VALUES(1001, (SELECT sys_DburiGen(rowid, employee_id)
                                         FROM employees WHERE employee_id=177));
1 row created.

-- Insert a DBUri that targets the last_name column of table employees
INSERT INTO doc_list_tab VALUES(1002,
                                (SELECT sys_DburiGen(employee_id, last_name)
                                   FROM employees WHERE employee_id=177));
1 row created.

SELECT * FROM doc_list_tab;

     DOCNO
----------
DOC_REF(URL, SPARE)
--------------------------------------------------
      1001
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[ROWID=''AAAL3LAAFAAAABSABN'']/EMPLOYEE_ID', NULL)
```

```
     1002
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID=''177'']/LAST_NAME', NULL)

2 rows selected.
```

Returning Partial Results

When selecting from a large column, you might sometimes want to retrieve only a portion of the result, and create a URL to the column instead. For example, consider the case of a travel story Web site. If travel stories are stored in a table, and users search for a set of relevant stories, then you do not want to list each entire story in the search-result page. Instead, you might show just the first 20 characters of each story, to represent the gist, and then return a URL to the full story. This can be done as follows:

Example 19–16 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);
Table created.

INSERT INTO travel_story
  VALUES ('Egypt', 'This is the story of my time in Egypt....');
1 row created.
```

We create a function that returns only the first 20 characters from the story:

```
CREATE OR REPLACE FUNCTION charfunc(clobval IN CLOB) RETURN VARCHAR2 IS
  res VARCHAR2(20);
  amount NUMBER := 20;
BEGIN
  DBMS_LOB.read(clobval, amount, 1, res);
  RETURN res;
END;
/
Function created.
```

We next create a view that selects only the first twenty characters from the story, and returns a DBUri to the story column.

```
CREATE OR REPLACE VIEW travel_view AS
  SELECT story_name, charfunc(story) short_story,
         sys_DburiGen(story_name, story, 'text()') story_link
  FROM travel_story;
View created.

SELECT * FROM travel_view;

STORY_NAME
----------
SHORT_STORY
-----------
STORY_LINK(URL, SPARE)
----------------------
Egypt
This is the story of
DBURITYPE('/PUBLIC/TRAVEL_STORY/ROW[STORY_NAME=''Egypt'']/STORY/text()', NULL)

1 row selected.
```

RETURNING URLs to Inserted Objects

You can use SQL function `sys_DburiGen` in the `RETURNING` clause of DML statements to retrieve the URL of an object as it is inserted.

Example 19–17 Using SYS_DBURIGEN in the RETURNING Clause to Retrieve a URL

In this example, whenever a document is inserted into table `clob_tab`, its URL is inserted into table `uri_tab`. This is done using SQL function `sys_DburiGen` in the `RETURNING` clause of the `INSERT` statement.

```
CREATE TABLE clob_tab (docid NUMBER, doc CLOB);
Table created.
CREATE TABLE uri_tab (docs SYS.DBURIType);
Table created.
```

In PL/SQL, we specify the storage of the URL of the inserted document as part of the insertion operation, using the `RETURNING` clause and `EXECUTE IMMEDIATE`:

```
DECLARE
  ret SYS.DBURIType;
BEGIN
  -- execute the insert operation and get the URL
  EXECUTE IMMEDIATE
    'INSERT INTO clob_tab VALUES (1, ''TEMP CLOB TEST'')
       RETURNING sys_DburiGen(docid, doc, ''text()'') INTO :1'
    RETURNING INTO ret;
  -- Insert the URL into uri_tab
  INSERT INTO uri_tab VALUES (ret);
END;
/

SELECT e.docs.getURL() FROM hr.uri_tab e;
E.DOCS.GETURL()
------------------------------------------------
/ORADB/PUBLIC/CLOB_TAB/ROW[DOCID='1']/DOC/text()

1 row selected.
```

DBUriServlet

Oracle XML DB Repository resources can be retrieved using the HTTP server that is incorporated in Oracle XML DB. Oracle Database also includes a servlet, **DBUriServlet**, that makes any kind of database data available through HTTP(S) URLs. The data can be returned as plain text, HTML, or XML.

A Web client or application can access such data without using SQL or a specialized database API. You can retrieve the data by linking to it on a Web page or by requesting it through HTTP-aware APIs of Java, PL/SQL, and Perl. You can display or process the data using an application such as a Web browser or an XML-aware spreadsheet. DBUriServlet can generate content that is XML data or not, and it can transform the result using XSLT style sheets.

You make database data Web-accessible by using a URI that is composed of a servlet address (URL) plus a DBUri URI that specifies which database data to retrieve. This is the syntax, where `http://`*`server:port`* is the URL of the servlet (machine and port), and `/oradb/`*`database_schema/table`* is the DBUri URI (any DBUri URI can be used):

`http://`*`server`*`:`*`port`*`/`**`oradb`**`/`*`database_schema/table`*

When using XPath notation in a URL for the servlet, you might need to escape certain characters. You can use `URIType` method `getExternalURL()` to do this.

You can either use DBUriServlet, which is preinstalled as part of Oracle XML DB, or write your own servlet that runs on a servlet engine. The servlet reads the URI portion of the invoking URL, creates a DBUri using that URI, calls `URIType` methods to retrieve the data, and returns the values in a form such as a Web page, an XML document, or a plain-text document.

The MIME type to use is specified to the servlet through the URI:

- By default, the servlet produces MIME types `text/xml` and `text/plain`. If the DBUri path ends in `text()`, then `text/plain` is used; otherwise, an XML document is generated with MIME type `text/xml`.
- You can override the default MIME type, setting it to `binary/x-jpeg` or some other value, by using the `contenttype` argument to the servlet.

> **See Also:** Chapter 27, "Writing Oracle XML DB Applications in Java", for information on Oracle XML DB servlets

Example 19–18 Using a URL to Override the MIME Type

To retrieve the `employee_id` column of the `employee` table, you can use a URL such as one of the following, where computer *`server.oracle.com`* is running Oracle Database with a Web service listening to requests on port 8080. Step `oradb` is the virtual path that maps to the servlet.

```
-- Produces a content type of text/plain
http://server.oracle.com:8080/oradb/QUINE/A/ROW[B=200]/C/text()

-- Produces a content type of text/xml
http://server.oracle.com:8080/oradb/QUINE/A/ROW[B=200]/C
```

To override the content type, you can use a URL that passes `text/html` to the servlet as the `contenttype` parameter:

```
-- Produces a content type of text/html
http://server.oracle.com:8080/oradb/QUINE/A/ROW[B=200]/C?contenttype=text/html
```

Table 19–3 describes each of the optional URL parameters you can pass to DBUriServlet to customize its output.

Table 19–3 DBUriServlet: Optional Arguments

Argument	Description
`rowsettag`	Changes the default root tag name for the XML document. For example: `http://server:8080/oradb/HR/EMPLOYEES?rowsettag=OracleEmployees` 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 generated document. For example: `http://server:8080/oradb/HR/EMPLOYEES?contenttype=text/plain`
`transform`	Passes a URL to `URIFACTORY`, which retrieves the XSL style sheet at that location. This style sheet is then applied to the XML document being returned by the servlet. For example: `http://server:8080/oradb/HR/EMPLOYEES?transform=/oradb/QUINE/XSLS/DOC/text()&contenttype=text/html`

Customizing DBUriServlet

DBUriServlet is built into the database – to customize the servlet, you must edit the Oracle XML DB configuration file, `xdbconfig.xml`. You can edit it as the Oracle XML DB user (`XDB`), using WebDAV, FTP, Oracle Enterprise Manager, or PL/SQL. To update the file using FTP or WebDAV, download the document, edit it, and save it back into the database.

See Also:

- Chapter 27, "Writing Oracle XML DB Applications in Java"
- Chapter 28, "Administering Oracle XML DB"

DBUriServlet is installed at `/oradb/*`, which is the address specified in the `servlet-pattern` tag of `xdbconfig.xml`. The asterisk (`*`) is necessary to indicate that any path following `oradb` is to be mapped to the same servlet. `oradb` is published as the virtual path. You can change the path that will be used to access the servlet.

Example 19–19 Changing the Installation Location of DBUriServlet

In this example, the configuration file is modified to install DBUriServlet under `/dburi/*`.

```
DECLARE
  doc XMLType;
  doc2 XMLType;
BEGIN
  doc := DBMS_XDB.cfg_get();
  SELECT
    updateXML(doc,
'/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/
servlet-mappings/servlet-mapping[servlet-name="DBUriServlet"]/servlet-pattern/
text()',
              '/dburi/*')
    INTO doc2 FROM DUAL;
  DBMS_XDB.cfg_update(doc2);
  COMMIT;
END;
/
```

Security parameters, the servlet display-name, and the description can also be customized in configuration file `xdbconfig.xml` . The servlet can be removed by deleting its servlet-pattern. This can also be done using SQL function `updateXML` to update the servlet-mapping element to `NULL`.

DBUriServlet Security

Servlet security is handled by Oracle Database using roles. When users log in to the servlet, they use their database username and password. The servlet checks to ensure that the user logging has one of the roles specified in the configuration file using parameter `security-role-ref`). By default, the servlet is available to role `authenticatedUser`, and any user who logs into the servlet with a valid database password has this role.

The role parameter can be changed to restrict access to any specific database roles. To change from the default `authenticated-user` role to a role that you have created, you modify the Oracle XML DB configuration file.

Example 19–20 Restricting Servlet Access to a Database Role

This example changes the default `authenticated-user` role to role `servlet-users` (which it is assumed you have created).

```
DECLARE
  doc  XMLType;
  doc2 XMLType;
  doc3 XMLType;
BEGIN
  doc := DBMS_XDB.cfg_get();
  SELECT updateXML(doc,
'/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/
servlet-list/servlet[servlet-name="DBUriServlet"]/security-role-ref/role-name/
text()',
                   'servlet-users')
    INTO doc2 FROM DUAL;
  SELECT updateXML(doc2,
'/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/
servlet-list/servlet[servlet-name="DBUriServlet"]/security-role-ref/role-link/
text()',
                   'servlet-users')
    INTO doc3 FROM DUAL;
  DBMS_XDB.cfg_update(doc3);
  COMMIT;
END;
/
```

Configuring Package URIFACTORY to Handle DBUris

A URL such as `http://server/servlets/oradb` is handled by DBUriServlet (or by a custom servlet). When a URL such as this is stored as a `URIType` instance, it is generally desirable to use subtype `DBURIType`, since this URI targets database data.

However, if a `URIType` instance is created using methods of package `URIFACTORY` such as `getURI()`, then by default the subtype used is `HTTPURIType`, not `DBURIType`. This is because `URIFACTORY` looks only at the URI prefix, sees `http://`, and assumes that the URI targets a Web page. This results in unnecessary layers of communication and perhaps extra character conversions.

To make things more efficient, you can teach `URIFACTORY` that URIs of the given form represent database accesses and so should be realized as DBUris, not HTTPUris. You do this by registering a handler for this URI as a prefix, specifying `DBURIType` as the type of instance to generate.

Example 19–21 Registering a Handler for a DBUri Prefix

This example effectively tells `URIFACTORY` that any URI string starting with `http://server/servlets/oradb` corresponds to a database access.

```
BEGIN
  URIFACTORY.registerURLHandler('http://server/servlets/oradb',
                              'SYS', 'DBURIType', true, true);
END;
/
```

After you execute this code, all `getURI()` calls in the same session automatically create DBUris for any URI strings with prefix `http://server/servlets/oradb`.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information on `URIFACTORY` functions

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:

20

Accessing Oracle XML DB Repository Data

This chapter describes how to access data in Oracle XML DB Repository using standard protocols such as FTP and HTTP(S)/WebDAV, and other Oracle XML DB resource Application Program Interfaces (APIs). It also introduces you to using `RESOURCE_VIEW` and `PATH_VIEW` as the SQL mechanism for accessing and manipulating repository data. It includes a table for comparing repository operations through the various resource APIs.

This chapter contains these topics:

Overview of Oracle XML DB Foldering

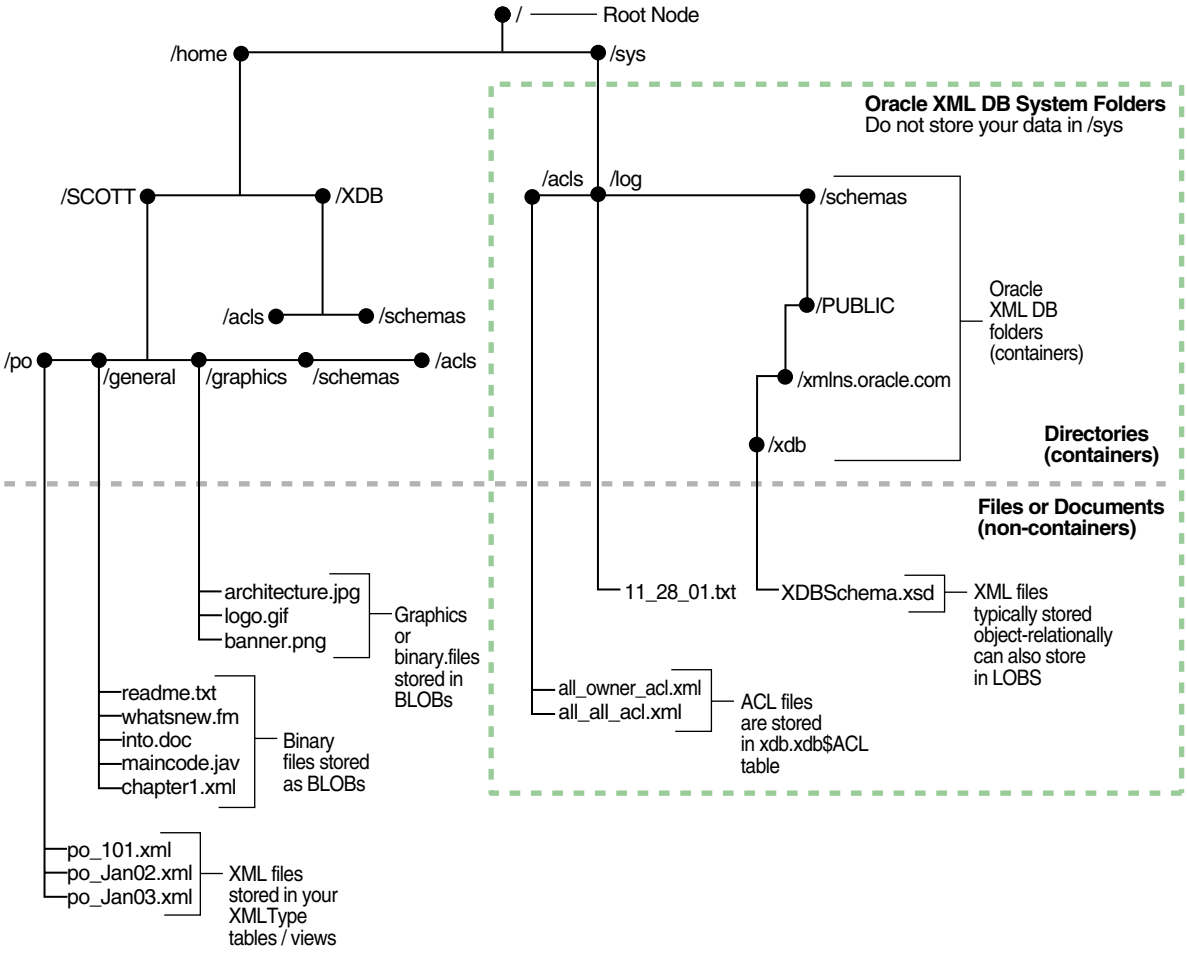
Using the foldering feature in Oracle XML DB you can store content in the database in hierarchical structures, as opposed to traditional relational database structures.

Figure 20–1 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(S), and WebDAV protocol standards as if the database content were 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. It discusses APIs that you can use to access the repository object hierarchy using Java, SQL, and PL/SQL.

Figure 20–1 A Folder Tree, Showing Hierarchical Structures in the Repository



Note: Folder `/sys` is used by Oracle XML DB to maintain *system-defined* XML schemas, Access Control Lists (ACLs), and so on. Do *not* add or modify any data in folder `/sys`.

See Also:

- Chapter 22, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"
- Chapter 23, "PL/SQL Access Using DBMS_XDB"
- Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data"

Repository Terminology and Supplied Resources

Oracle XML DB Repository is the set of database objects, across all XML and database schemas, that are mapped to path names. It is a connected, directed, acyclic graph of resources, with a single root node (`/`). Each resource in the graph has one or more associated path names: the repository supports *multiple links to a given resource*. The repository can be thought of as a file system of objects rather than files.

Repository Terminology

The following list describes terms used in Oracle XML DB Repository:

- **resource** – Any object or node in the repository hierarchy. Resources are identified by URLs.

  See Also:

  "Overview of Oracle XML DB Repository" on page 1-7

  "Oracle XML DB Resources" on page 20-4

- **folder** – A resource that can contain other resources. Sometimes called a **directory**.
- **path name** – A hierarchical name representing a path to a resource. It is composed of a slash (`/`) representing the root, possibly followed by path elements separated by slashes. A **path element** is the name of a repository resource. A path element may be composed of any character in the database character set except `\` and `/`, which have special meaning in Oracle XML DB. The slash (`/`) is the default name separator in a path name. The backslash (`\`) is used to escape special characters, giving them a literal interpretation. The Oracle XML DB configuration file, `xdbconfig.xml`, contains a list of user-defined characters that must not appear within a path name (`<invalid-pathname-chars>`).
- **resource name** or **link name** – The name of a resource within its parent folder. Resource names must be unique within a folder and are case-sensitive. Resource names are always in the UTF-8 character set (`NVARCHAR`).
- **resource content** – The body, or data, of a resource. This is what you get when you treat the resource as a file and ask for its content. This is always of type `XMLType`.
- `XDBBinary` element – An XML element that contains binary data. It is defined by the Oracle XML DB XML schema. `XDBBinary` elements are stored in the repository whenever unstructured binary data is uploaded into Oracle XML DB.
- **access control list** (ACL) – A list of database users that allowed access to one or more specific resources.

  See Also: Chapter 24, "Repository Resource Security"

Many terms used by Oracle XML DB have common synonyms used in other contexts, as shown in Table 20–1.

Table 20–1 Synonyms for Oracle XML DB Foldering Terms

Synonym	Foldering Term	Usage
collection	folder	WebDAV
directory	folder	operating systems
privilege	privilege	permission
right	privilege	various
WebDAV folder	folder	web folder
role	group	access control
revision	version	RCS, CVS
file system	repository	operating systems
hierarchy	repository	various

Table 20–1 (Cont.) Synonyms for Oracle XML DB Foldering Terms

Synonym	Foldering Term	Usage
file	resource	operating systems
binding	link	WebDAV

Supplied Files and Folders

The list of supplied Oracle XML DB Repository files and folders is as follows. In addition to using these, you can create your own folders and files wherever you want.

```
/public
/sys
/sys/acls
/sys/acls/all_all_acl.xml
/sys/acls/all_owner_acl.xml
/sys/acls/bootstrap_acl.xml
/sys/acls/ro_all_acl.xml
/sys/apps
/sys/asm
/sys/log
/sys/schemas
/sys/schemas/PUBLIC
/sys/schemas/PUBLIC/www.w3.org
/sys/schemas/PUBLIC/www.w3.org/2001
/sys/schemas/PUBLIC/www.w3.org/2001/xml.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBFolderListing.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResource.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBSchema.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBStandard.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/dav.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/ftplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/httplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/xdblog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/stats.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/xdbconfig.xsd
/xdbconfig.xml
```

Oracle XML DB Resources

Oracle XML DB Repository resources conform to the Oracle XML DB XML schema `xdbresource.xsd`. The elements in a resource include those needed to persistently store WebDAV-defined properties, such as creation date, modification date, WebDAV locks, owner, ACL, language, and character set.

Contents Element in Resource Index

A resource index has a special element called `Contents` that contains the contents of the resource.

any Element

The XML schema for a resource also defines an `any` element, with `maxoccurs` attribute unbounded. An `any` element can contain any element outside of the Oracle

XML DB XML namespace. Arbitrary instance-defined properties can be associated with the resource.

Where Is Repository Data Stored?

Oracle XML DB stores Oracle XML DB Repository data in a set of tables and indexes to which you have access. If you register an XML schema and request that the tables be generated by Oracle XML DB, then the tables are created in your database schema. You are then able to see or modify them. Other users will not be able to see your tables unless you grant them permission to do so.

Names of Generated Tables

The names of the generated tables are assigned by Oracle XML DB and can be obtained by finding the `xdb:defaultTable` attribute in your XML schema document (or in the default XML schema document). When you register an XML schema, you can alternatively provide your own table name, instead of using the default name supplied by Oracle XML DB.

> **See Also:** "Creating Default Tables During XML Schema Registration" on page 5-10

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 that conforms to a visible, registered XML schema.

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

ASM Virtual Folder

The ASM virtual folder, `/sys/asm`, is an exception to the description of the previous sections – its contents are ASM files and folders that are managed automatically by Oracle Automatic Storage Management (ASM).

> **See Also:**
>
> - "Accessing ASM Files Using Protocols and Resource APIs – For DBAs" on page 20-10
> - *Oracle Database Administrator's Guide*

Path-Name Resolution

The data relating a folder to its contents 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 the user receives an `access denied` error. Folder listings and other queries will not return a row when the `read-properties` privilege is denied on its resource.

Caution: Error handling in path-name resolution differentiates between invalid resource names and resources that are not folders, for compatibility with file systems. Because Oracle XML DB resources are accessible from outside Oracle XML DB Repository (using SQL), denying read access on a folder that contains a resource does *not prevent* read access to that resource.

Resource Deletion

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.

See Also: "Deleting Repository Resources: Examples" on page 22-13

Accessing Oracle XML DB Repository Resources

There are two ways to access Oracle XML DB Repository resources:

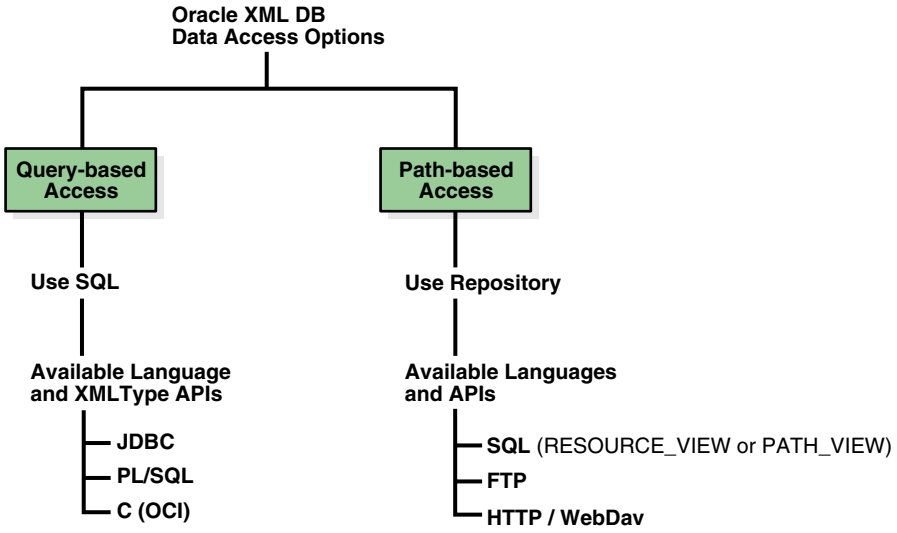
- Navigational or path-based access. This is achieved using a hierarchical index of objects or resources. Each resource has one or more unique path names that reflect its location in the hierarchy. You can use navigational access to reference any `XMLType` object in the database, without regard to its location in the relational tablespace. See *"Navigational or Path Access"* on page 20-7.
- SQL access to the repository. This is done using special views that expose resource properties and path names, and map hierarchical access operators onto the Oracle XML DB schema. See *"Query-Based Access" on page 20-11.*

Figure 20–2 illustrates these two Oracle XML DB data access options.

See Also:

- "Oracle XML DB Application Design: A. How Structured Is Your Data?" on page 2-4 for guidance on selecting an access method
- Table 20–3, " Accessing Oracle XML DB Repository: API Options" for a summary comparison of the access methods

Figure 20–2 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 20–3.
- Access Oracle XML DB Repository data using HTTP(S)/WebDAV from an Internet Explorer browser, such as when viewing Web Folders, as shown in Figure 20–4.

Figure 20–3 Oracle XML DB Folders in Windows Explorer

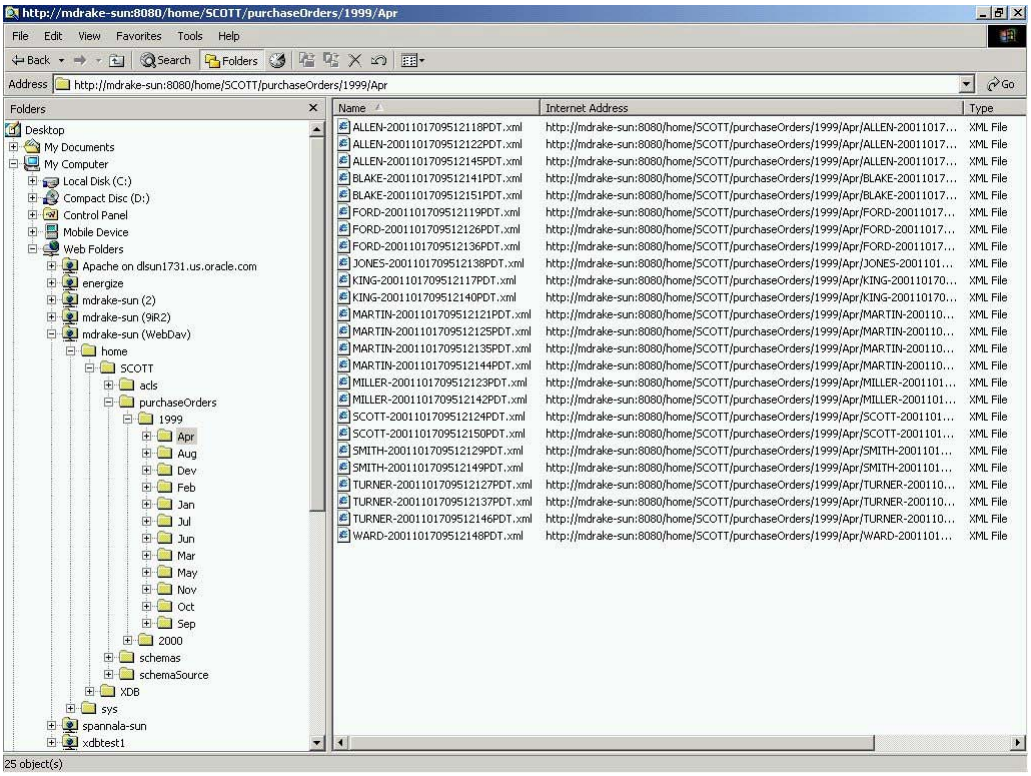
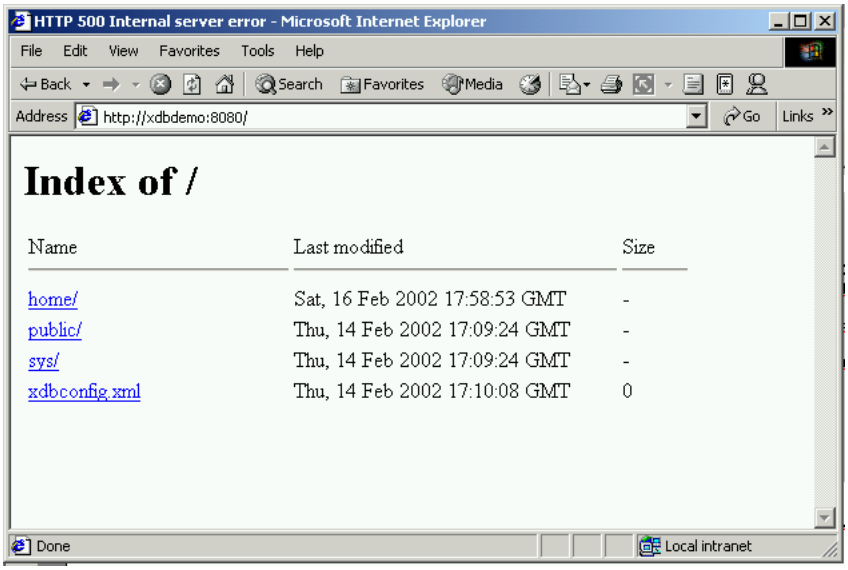


Figure 20–4 Accessing Repository Data Using HTTP(S)/WebDAV and Navigational Access From IE Browser: Viewing Web Folders



Accessing Oracle XML DB Resources Using Internet Protocols

Oracle Net Services provides one way of accessing database resources. Oracle XML DB support for Internet protocols provides another way of accessing database resources.

Where You Can Use Oracle XML DB Protocol Access

Oracle Net Services is optimized for record-oriented data. Internet protocols are designed for stream-oriented data, such as binary files or XML text documents. Oracle XML DB protocol access is a valuable alternative to Net Services in the following scenarios:

- Direct database access from file-oriented applications using the database like a file system
- Heterogeneous application server environments that require a uniform data access method (such as XML over HTTP, which is supported by most data servers, including MS SQL Server, Exchange, Notes, many XML databases, stock quote services and news feeds)
- Application server environments that require data in the form of XML text
- Web applications that use client-side XSL to format datagrams that do not need much application processing
- Web applications that use Java servlets that run inside the database
- Web access to XML-oriented stored procedures

Using Protocol Access

Follow these steps to use Oracle XML DB protocol access:

1. A connection object is established, and the protocol might read part of the request.
2. The protocol decides whether the user is already authenticated and wants to reuse an existing session or the connection must be re-authenticated (the latter is more common).
3. An existing session is pulled from the session pool, or else a new one is created.
4. If authentication has not been provided, and the request is HTTP `get` or `head`, then the session is run as the `ANONYMOUS` user. If the session has already been authenticated as the `ANONYMOUS` user, then there is no cost to reuse the existing session. If authentication has been provided, then the database re-authentication routines are used to authenticate the connection.
5. The request is parsed.
6. (HTTP only) If the requested path name maps to a servlet, then the servlet is invoked using Java Virtual Machine (VM). The servlet code writes the response to a response stream or asks `XMLType` instances to do so.

Retrieving Oracle XML DB Resources

When the protocol indicates that a resource is to be retrieved, the path name to the resource is resolved. Resources being fetched are always streamed out as XML, with the exception of resources containing the `XDBBinary` element, an element defined to be the XML binary data type, which have their contents streamed out in RAW form.

Storing Oracle XML DB Resources

When the protocol indicates that a resource must be stored, Oracle XML DB checks the document file name extension for `.xml`, `.xsl`, `.xsd`, and so on. If the document is XML, then a pre-parse step is done, whereby enough of the resource is read to determine the XML `schemaLocation` and `namespace` of the root element in the document. If a registered schema is located at the `schemaLocation` URL, and it has a

definition for the root element of the current document, then the default table specified for that root element is used to store the contents of the resource.

Using Internet Protocols and XMLType: XMLType Direct Stream Write

Oracle XML DB supports Internet protocols at the `XMLType` level by using the `writeToStream()` Java method on `XMLType`. This method is natively implemented, and writes `XMLType` data directly to the protocol request stream. This avoids Java VM execution costs and the overhead of converting database data through Java datatypes and creating Java objects, resulting in significantly higher performance. Performance is further enhanced if the Java code deals only with XML element trees that are close to the root, and does not traverse too many of the leaf elements, so that relatively few Java objects are created.

> **See Also:** Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data"

Accessing ASM Files Using Protocols and Resource APIs – For DBAs

Automatic Storage Management (ASM) organizes database files into *disk groups* for simplified management and added benefits such as database mirroring and I/O balancing.

Repository access using protocols and resource APIs (such as `DBMS_XDB`) extends to Automatic Storage Management (ASM) files. ASM files are accessed in the *virtual* repository folder `/sys/asm`. However, this access is reserved for DBAs; it is *not* intended for developers.

A typical use of such access is to copy ASM files from one database instance to another. For example, a DBA can view folder `/sys/asm` in a graphical user interface using the WebDAV protocol, and then drag-and-drop a copy of a data-pump dumpset from an ASM disk group to an operating-system file system.

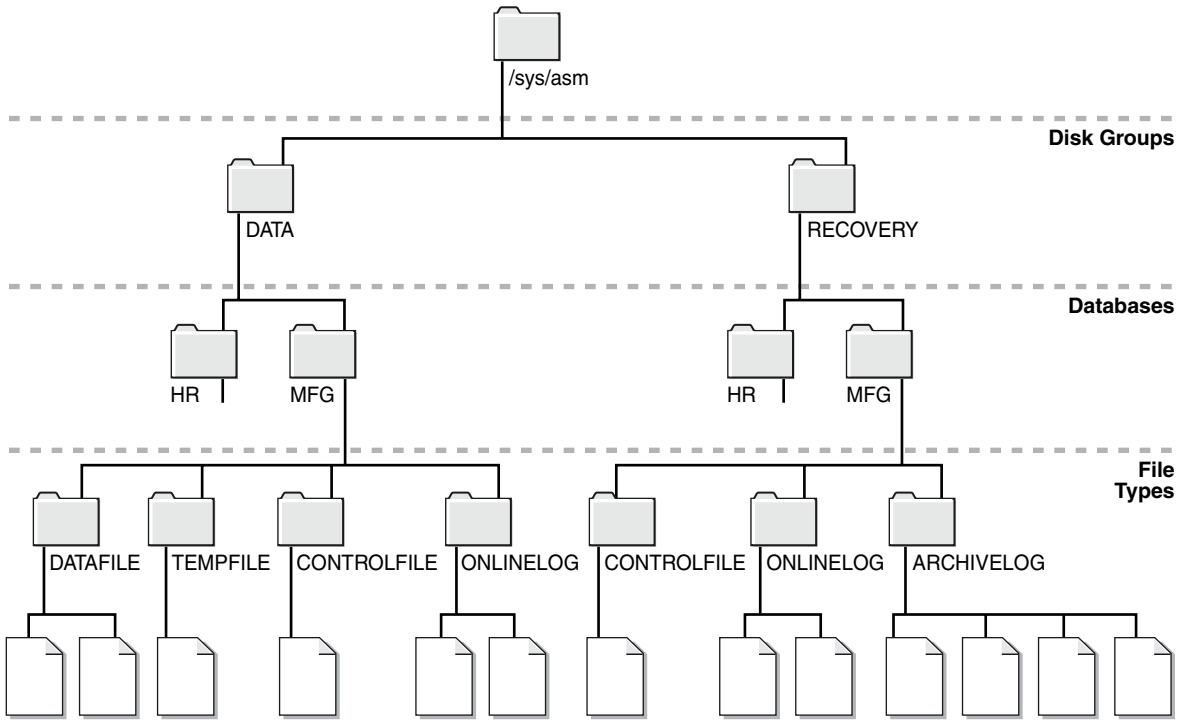
Virtual folder `/sys/asm` is created by default during Oracle XML DB installation. If the database is not configured to use ASM, the folder is empty and no operations are permitted on it.

Folder `/sys/asm` contains folders and subfolders that follow the hierarchy defined by the structure of an ASM *fully qualified filename*:

- It contains a subfolder for each mounted *disk group*.
- A disk-group folder contains a subfolder for each *database* that uses that disk group. In addition, a disk-group folder may contain files and folders corresponding to ASM *aliases* created by the administrator.
- A database folder contains file-type folders.
- A file-type folder contains ASM files, which are binary.

This hierarchy is shown in Figure 20–5, which omits directories created for aliases, for simplicity.

Figure 20–5 ASM Virtual Folder Hierarchy



The following usage restrictions apply to virtual folder `/sys/asm`. You *cannot*:

- *query* `/sys/asm` using SQL
- put regular files under `/sys/asm` (you can only only put ASM files there)
- *move* (rename) an ASM file to a different ASM disk group or to a folder outside ASM
- create *hard links* to existing ASM files or directories

In addition:

- You must have DBA privileges to view folder `/sys/asm`.
- To access `/sys/asm` using Oracle XML DB protocols, you must log in as a user other than `SYS`.

Again, ASM virtual-folder operations are intended only for *DBAs*, not developers.

See Also:

- "Using FTP with ASM Files" on page 25-10 for an example of using protocol FTP with `/sys/asm`
- *Oracle Database Administrator's Guide* for information on the syntax of a fully qualified ASM filename and details on the virtual folder structure

Query-Based Access

There are two views that enable SQL access to Oracle XML DB Repository data:

- `PATH_VIEW`
- `RESOURCE_VIEW`

Table 20–2 summarizes the differences between PATH_VIEW and RESOURCE_VIEW.

Table 20–2 Differences Between PATH_VIEW and RESOURCE_VIEW

PATH_VIEW	RESOURCE_VIEW
Contains link properties	No link properties
Has one row for each unique *path* in repository	Has one row for each *resource* in repository

Rows in these two repository views are of XMLType. In the RESOURCE_VIEW, the single path associated with a resource is arbitrarily chosen from among the possible paths that refer to the resource. Oracle XML DB provides SQL functions like under_path that enable applications to search for the resources contained (recursively) within a particular folder, obtain the resource depth, and so on.

DML can be used on the repository views to insert, rename, delete, and update resource properties and contents. Programmatic APIs must be used for other operations, such as creating links to existing resources.

See Also:

- Chapter 22, "SQL Access Using RESOURCE_VIEW and PATH_VIEW" for details on SQL access to Oracle XML DB Repository
- Chapter 24, "Repository Resource Security"

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 27, "Writing Oracle XML DB Applications in Java"

Accessing Data Stored in 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(S)/WebDAV and FTP) and Oracle XML DB protocol server

Table 20–3 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 22, "SQL Access Using RESOURCE_VIEW and PATH_VIEW"
- Chapter 23, "PL/SQL Access Using DBMS_XDB"
- Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data"
- *Oracle Database PL/SQL Packages and Types Reference*

Table 20–3 Accessing Oracle XML DB Repository: API Options

Data Access	SQL and PL/SQL	Protocols
Create resource	`DBMS_XDB.createResource()` `INSERT INTO PATH_VIEW VALUES (path, res, linkprop);`	HTTP: `PUT;` FTP: `PUT`

Update resource contents	`UPDATE RESOURCE_VIEW SET resource =`
	`updateXML(res, '/Resource/Contents', lob)`
	`WHERE equals_path(res, path) > 0`
		HTTP: `PUT;`
		FTP: `PUT`

Update resource properties	`UPDATE RESOURCE_VIEW SET resource =`
	`updateXML(res, '/Resource/propname1',`
	`newval, '/Resource/propname2' ...)`
	`WHERE equals_path(res, path) > 0`
		WebDAV: `PROPPATCH;`

Update resource ACL	`UPDATE RESOURCE_VIEW SET resource =`
	`updateXML(res, '/ Resource/ACL', XMLType)`
	`WHERE equals_path(res, path) > 0`
		—

Unlink resource (delete if last link)	`DBMS_XDB.deleteResource()`
	`DELETE FROM RESOURCE_VIEW`
	`WHERE equals_path(res, path) > 0`
		HTTP: `DELETE;`
		FTP:`delete`

Forcibly remove all links to resource	`DBMS_XDB.deleteResource()`
	`DELETE FROM PATH_VIEW`
	`WHERE extractValue(res, 'display_name')`
	`= 'My resource'`
		FTP: `quote rm_rf` *`resource`*

Move resource	`UPDATE PATH_VIEW SET path = newpath` `WHERE equals_path(res, path) > 0`	WebDAV: `MOVE;` FTP: `rename`

Copy resource	`INSERT INTO PATH_VIEW`
	`SELECT newpath, res, link FROM PATH_VIEW`
	`WHERE equals_path(res, oldpath) > 0`
		WebDAV: `COPY;`

Create link to existing resource	`CALL DBMS_XDB.link(srcpath IN VARCHAR2,`
	`linkfolder IN VARCHAR2,`
	`linkname IN VARCHAR2);`
		—

Get binary or text representation of resource contents	`SELECT XDBURIType(path).getBlob() FROM DUAL;`
	
	`SELECT p.res.extract('/Resource/Contents')`
	`FROM RESOURCE_VIEW p`
	`WHERE equals_path(res, path) > 0`
		HTTP: `GET;`
		FTP: `get`

Get `XMLType` representation of resource contents	`SELECT XDBURIType(path).getBlob().getXML FROM DUAL;`
	
	`SELECT extract(res, '/Resource/Contents/*')`
	`FROM RESOURCE_VIEW p`
	`WHERE equals_path(Res, path) > 0`
		—

Table 20–3 (Cont.) Accessing Oracle XML DB Repository: API Options

Data Access	SQL and PL/SQL	Protocols
Get resource properties	`SELECT extractValue(res, '/Resource/XXX') FROM RESOURCE_VIEW WHERE equals_path(res, path) > 0`	WebDAV:`PROPFIND (depth = 0);`
List directory	`SELECT PATH FROM PATH_VIEW WHERE under_path(res, path, 1) > 0`	WebDAV:`PROPFIND (depth = 0);`
Create folder	`Call DBMS_XDB.createFolder(VARCHAR2)`	WebDAV: `MKCOL;` FTP: `mkdir`
Unlink folder	`DBMS_XDB.deleteResource()` `DELETE FROM PATH_VIEW WHERE equals_path(res, path) > 0;`	HTTP: `DELETE;` FTP: `rmdir`
Forcibly delete folder and all links to it	`Call DBMS_XDB.deleteResource(VARCHAR2);`	—
Get resource with a row lock	`SELECT ... FROM RESOURCE_VIEW FOR UPDATE ...;`	—
Add WebDAV lock on resource	`DBMS_XDB.LockResource(path, true, true);`	WebDAV: `LOCK;` FTP: `quote lock`
Remove WebDAV lock	`BEGIN DBMS_XDB.GetLockToken(path, deltoken); DBMS_XDB.UnlockToken(path, deltoken); END;`	WebDAV: `UNLOCK;` FTP: `quote unlock`
Commit changes	`COMMIT;`	Automatic commit after each request
Rollback changes	`ROLLBACK;`	—

Managing and Controlling Access to Resources

You can set access control privileges on Oracle XML DB folders and resources.

See Also:

- Chapter 24, "Repository Resource Security" for more detail on using access control on Oracle XML DB folders
- *Oracle Database PL/SQL Packages and Types Reference*

21

Managing Resource Versions

This chapter describes how to create and manage versions of Oracle XML DB resources.

This chapter contains these topics:

Overview of Oracle XML DB Versioning

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 PL/SQL package `DBMS_XDB_VERSION` to put a resource under version-control and retrieve different versions of the resource.

Oracle XML DB Versioning Features

Versioning helps you keep track of changes to resources in Oracle XML DB Repository. The following sections discuss these features in detail. Oracle XML DB versioning features include the following:

- ***Version control on a resource***. You can turn version control on or off for an Oracle XML DB Repository resource. See "Creating a Version-Controlled Resource (VCR)".
- ***Updating process of a version-controlled resource.*** When Oracle XML DB updates a version-controlled resource (VCR), it creates a new version of the resource. This new version will not be deleted from the database when you delete the version-controlled resource. See "Updating a Version-Controlled Resource (VCR)".
- ***Loading a VCR is similar to loading a 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: Oracle XML DB 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 check-in and check-out. Everything but the resource object ID of the last version is preserved.

Oracle XML DB supports versioning of XML *schema*-based resources only if the schema tables have *no* associated triggers or constraints.

Oracle XML DB Versioning Terms Used in This Chapter

Table 21–1 lists the Oracle XML DB versioning terms used in this chapter.

Table 21–1 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	A **version-controlled resource** (**VCR**) is an Oracle XML DB resource that is under version control.
Version resource	A 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()`, `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

A resource ID is a unique system-generated ID for an Oracle XML DB resource. It helps identify resources that do not have path names. For example, a version resource is a system-generated resource and does not have a path name. You can use PL/SQL function `GetResourceByResId` to retrieve resources given the resource object ID. The first version ID is returned if a resource is versioned.

Example 21–1 Using DBMS_XDB_VERSION.GetResourceByResId To Retrieve a Resource

```
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 is created, the first version resource of the VCR is created, and the VCR is a reference to this newly-created version.

See "Version Resource ID or VCR Version" on page 21-3.

Example 21–2 Using DBMS_XDB_VERSION.MakeVersioned To Create a 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 21-3.

`MakeVersioned()` is not an auto-commit SQL operation. You have to commit the operation.

Version Resource ID 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 resource ID is returned by a few methods in package `DBMS_XDB_VERSION`, as 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 21–3 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 21–4 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 VCR to update its contents
  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)

Updating a VCR requires more steps than updating a resource that is not version-controlled. Before updating the contents and metadata properties of a VCR, check out the resource. The resource must then be checked in to make the update permanent. You must explicitly commit the SQL transaction.

To update a VCR follow these steps:

1. ***Check out the VCR,*** passing the VCR path name to Oracle XML DB.
2. ***Update the VCR.*** You can update either the contents or the metadata properties of the VCR. A new VCR version is not created until check-in – in particular, an update or a deletion operation does not permanently take effect until after check-in. (You can perform an update using SQL through `RESOURCE_VIEW` or `PATH_VIEW`, or through any protocol such as WebDAV.)
3. ***Check in the VCR or cancel its check-out***. If the resource is not checked out, then the previous version is copied onto the current version. The previous version is then deleted.

DBMS_XDB_VERSION.CheckOut() Procedure

In Oracle9*i* release 2 (9.2) and higher, the VCR check-out 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 check-out. 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 21–5 VCR Check-Out

For example:

```
BEGIN
  -- Resource '/home/SCOTT/versample.html' is checked out.
  DBMS_XDB_VERSION.checkout('/home/SCOTT/versample.html');
END;
/
```

DBMS_XDB_VERSION.CheckIn() Procedure

In Oracle9*i* release 2 (9.2) and higher, the VCR check-in operation is executed by calling `DBMS_XDB_VERSION.CheckIn()`. Procedure `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 check-out, but the check-in and check-out path names must be of the same resource.

Example 21–6 VCR Check-In

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;
/
```

DBMS_XDB_VERSION.UnCheckOut Procedure

In Oracle9*i* release 2 (9.2) and higher, a check-out is cancelled by calling `DBMS_XDB_VERSION.UnCheckOut()`. This path name does not have to be the same as the path name that was passed to check-out, but the check-in and check-out path names must be of the same resource.

Example 21–7 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. For example, you can use `RESOURCE_VIEW`, `PATH_VIEW`, or WebDAV.

> **See Also:** Chapter 22, "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 resource that is not version-controlled. When you request access to these resources, the ACL is checked.

> **See Also:** Chapter 24, "Repository Resource Security"

Version Resource

When a regular resource is converted to a VCR using `makeversion`, the first version resource is created, and the ACL of this first version resource 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 isthe same as the ACL of the checked-out resource. After a version resource is created, its ACL cannot be changed.

ACLs 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 21–2 describes the subprograms in `DBMS_XDB_VERSION`.

Table 21–2 DBMS_XDB_VERSION Functions and Procedures

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 check out 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 check-out 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. Procedure `CheckIn()` does not have to take the same path name that was passed to check-out operation. However, the check-in path name and the check-out 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 check in 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 check in 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. Procedure `UncheckOut()` does not have to take the same `path name` that was passed to check-out operation. However, the `UnCheckOut()` path name and the check-out 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 for 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 check in the resource.

Table 21–2 (Cont.) DBMS_XDB_VERSION Functions and Procedures

Function/Procedure	Description
FUNCTION GetPredecessors GetPredecessors(pathname VARCHAR2) RETURN RESID_LIST_TYPE; GetPredsByResId(resid DBMS_XDB.RESID_TYPE) RETURN RESID_LIST_TYPE;	Given a version resource or a VCR, gets the predecessors of the resource by 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. 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; GetSuccsByResId(resid DBMS_XDB.RESID_TYPE) RETURN RESID_LIST_TYPE;	Given a version resource or a VCR, gets the successors of the resource by pathname, the path name of the resource. return - list of predecessors. Getting successors by resid is more efficient than by path name. An exception is raised if the resid or pathname is not permitted. 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 change a VCR to no longer be version-controlled.
- 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.
- 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 resources. Versioning only works for non-schema-based resources, so that no uniqueness 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 check-out happens and the transaction is committed, any user can edit the resource.
- When a checked-out resource is checked in, the original previously checked-out version is added to the version history.
- Resource metadata is maintained for each version. Versions are stored in the same tables. Versioning works only for XML non-schema-based resources, hence no constraints are violated.

22

SQL Access Using RESOURCE_VIEW and PATH_VIEW

This chapter describes the predefined public views, RESOURCE_VIEW and PATH_VIEW, that provide access to Oracle XML DB repository data. It discusses SQL functions under_path and equals_path that query resources based on their path names and path and depth that return resource path names and depths, respectively.

This chapter contains these topics:

Overview of Oracle XML DB RESOURCE_VIEW and PATH_VIEW

Figure 22–1 shows how Oracle XML DB RESOURCE_VIEW and PATH_VIEW provide a mechanism for using SQL to access data stored in Oracle XML DB Repository. Data stored in the repository using protocols such as FTP and WebDAV, or using application program interfaces (APIs), can be accessed in SQL using RESOURCE_VIEW values and PATH_VIEW values.

RESOURCE_VIEW consists of a resource, itself an XMLType, that contains the name of the resource that can be queried, its ACLs, and its properties, static or extensible.

- If the content comprising the resource is XML, 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.

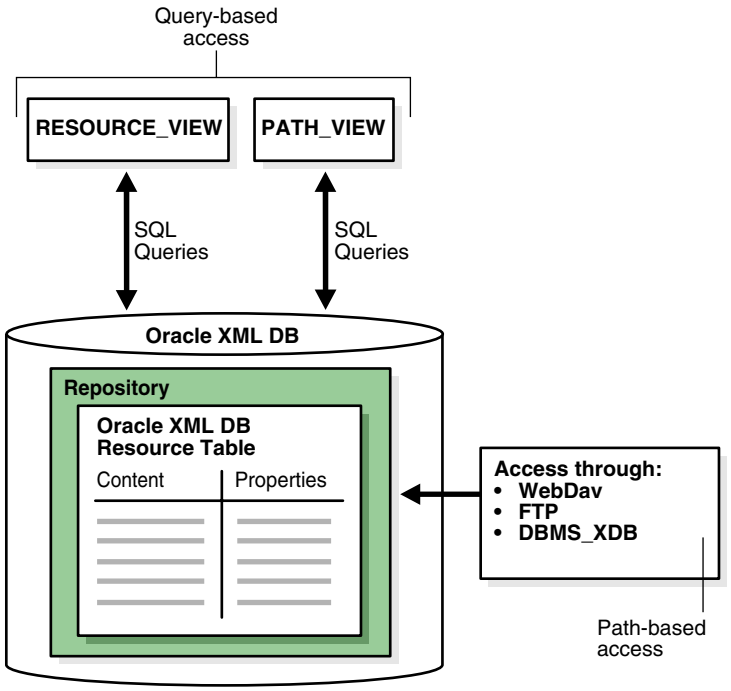
Parent-child relationships between folders (necessary to construct the hierarchy) are maintained and traversed efficiently using the hierarchical index. Text indexes are available to search the properties of a resource, and internal B*Tree indexes over Names and ACLs speed up access to these attributes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

RESOURCE_VIEW and PATH_VIEW, along with PL/SQL package DBMS_XDB, provide all query-based access to Oracle XML DB and DML functionality that is available through the API.

The base table for RESOURCE_VIEW is XDB.XDB$RESOURCE. This table should only be accessed through RESOURCE_VIEW or the DBMS_XDB API.

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

Figure 22–1 Accessing Repository Resources Using RESOURCE_VIEW and PATH_VIEW



RESOURCE_VIEW Definition and Structure

The RESOURCE_VIEW contains one row for each resource in Oracle XML DB Repository. Table 22–1 describes its structure.

Table 22–1 Structure of RESOURCE_VIEW

Column	Datatype	Description
RES	XMLType	A resource in the repository
ANY_PATH	VARCHAR2	An (absolute) path to the resource
RESID	RAW	Resource OID, which is a unique handle to the resource

PATH_VIEW Definition and Structure

The PATH_VIEW contains one row for each unique path to access a resource in Oracle XML DB Repository. Table 22–2 describes its structure.

Table 22–2 Structure of PATH_VIEW

Column	Datatype	Description
PATH	VARCHAR2	An (absolute) path to repository resource RES
RES	XMLType	The resource referred to by column PATH
LINK	XMLType	Link property
RESID	RAW	Resource OID

Figure 22–2 illustrates the structure of RESOURCE_VIEW and PATH_VIEW.

Note: Each resource may have multiple paths, called links.

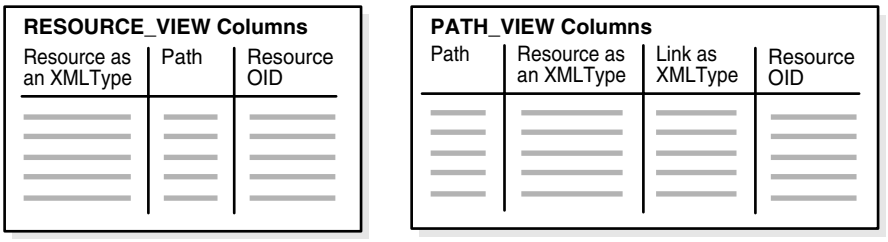
The path in the RESOURCE_VIEW is an arbitrary one and one of the accessible paths that can be used to access that resource. Oracle XML DB provides SQL function under_path, which enables applications to search for resources contained (recursively) within a particular folder, get the resource depth, and so on. Each row in the PATH_VIEW and RESOURCE_VIEW columns is of XMLType. DML on repository views can be used to insert, rename, delete, and update resource properties and contents. Programmatic APIs must be used for some operations, such as creating links to existing resources.

Paths in the ANY_PATH column of the RESOURCE_VIEW and the PATH column in the PATH_VIEW are *absolute* paths: they start at the root.

Paths returned by the path function are *relative* paths under the path name specified by function under_path. For example, if there are two resources referenced by path names /a/b/c and /a/d, respectively, then a path expression that retrieves paths under folder /a 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 function under_path 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 22–2 RESOURCE_VIEW and PATH_VIEW Structure



Understanding the Difference Between RESOURCE_VIEW and PATH_VIEW

Views RESOURCE_VIEW and PATH_VIEW differ as follows:

- PATH_VIEW displays *all* the path names to a particular resource. RESOURCE_VIEW displays *one* of the possible path names to the resource
- PATH_VIEW also displays the properties of the link

Figure 22–3 illustrates this difference between RESOURCE_VIEW and PATH_VIEW.

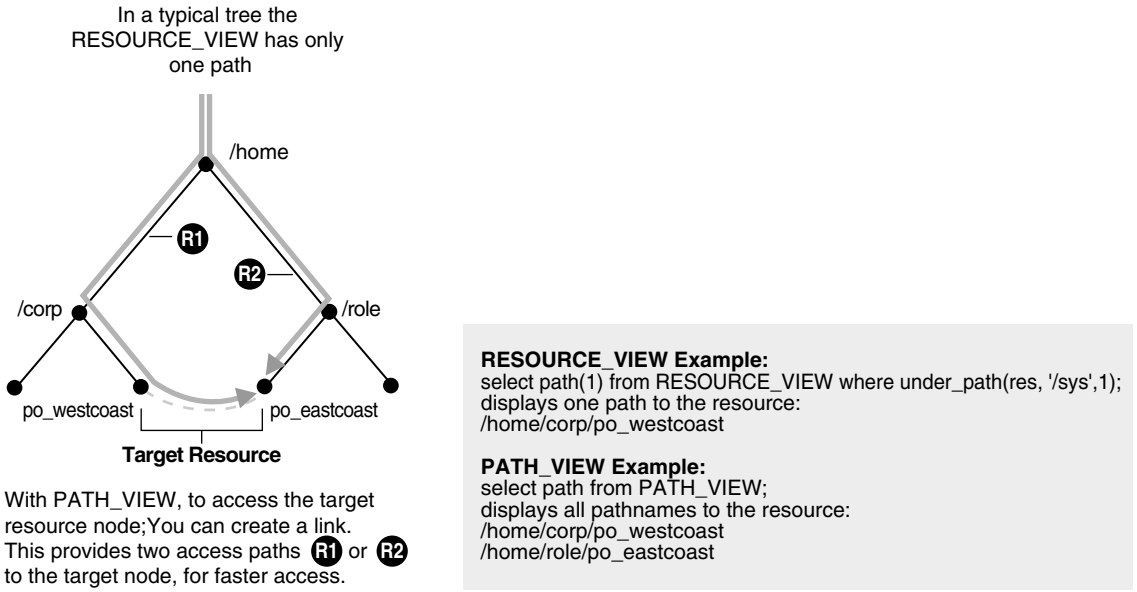
Because many Internet applications only need one URL to access a resource, RESOURCE_VIEW is widely applicable.

PATH_VIEW contains the *link* properties 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 functions `under_path` or `equals_path`, 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 22–3 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 SQL function `under_path` or other SQL functions:
  - Update resources
  - Delete resources
  - Get resources or their OID

See the "Using the RESOURCE_VIEW and PATH_VIEW APIs" and `equals_path`.

RESOURCE_VIEW and PATH_VIEW APIs

This section describes the SQL functions applicable to `RESOURCE_VIEW` and `PATH_VIEW`.

UNDER_PATH SQL Function

SQL function `under_path` 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 22–4 shows the `under_path` syntax.

Figure 22–4 UNDER_PATH Syntax

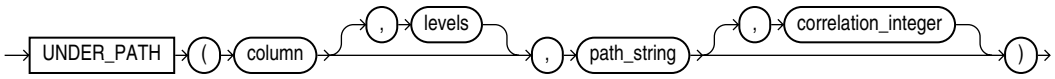


Table 22–3 details the `under_path` syntax.

Table 22–3 RESOURCE_VIEW and PATH_VIEW API Syntax: UNDER_PATH

Syntax	Description
`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.
`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 nonnegative integer. ▪ `pathname` - The path name to resolve.
`under_path(resource_column, pathname, correlation);`	Determines if a resource is under a specified path, with a correlation argument for related SQL functions. Parameters: ▪ `resource_column` - The column name or column alias of the `RESOURCE` column in the `PATH_VIEW` or `RESOURCE_VIEW`. ▪ `pathname` - The path name to resolve. ▪ `correlation` - An integer that can be used to correlate `under_path` with related SQL functions (`path` and `depth`).

Table 22–3 (Cont.) RESOURCE_VIEW and PATH_VIEW API Syntax: UNDER_PATH

Syntax	Description
`under_path(resource_column, depth, pathname, correlation);`	Determines if a resource is under a specified path with a depth argument to restrict the number of levels to search, and with a correlation argument for related SQL functions. Parameters: ▪ `resource_column` - The column name or column alias of the `RESOURCE` column in the `PATH_VIEW` or `RESOURCE_VIEW`. ▪ `depth` - The maximum depth to search. A nonnegative integer. ▪ `pathname` - The path name to resolve. ▪ `correlation` - An integer that can be used to correlate `under_path` with related SQL functions (`path` and `depth`). Note that only one of the accessible paths to the resource must be under the path argument for a resource to be returned.

EQUALS_PATH SQL Function

SQL function `equals_path` is used to find the resource with the specified path name. It is functionally equivalent to `under_path` with a depth restriction of zero.

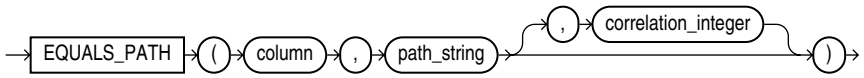
```
equals_path(resource_column, pathname);
```

where:

- `resource_column` is the column name or column alias of the `RESOURCE` column in the `PATH_VIEW` or `RESOURCE_VIEW`.
- `pathname` is the path name to resolve.

Figure 22–5 illustrates the complete `equals_path` syntax.

Figure 22–5 EQUALS_PATH Syntax



PATH SQL Function

SQL function `path` returns the relative path name of the resource under the specified `pathname` argument. Note that the path column in the `RESOURCE_VIEW` always contains the absolute path of the resource. The `path` syntax is:

```
path(correlation);
```

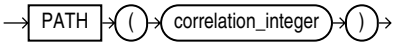
where:

- `correlation` is an integer that can be used to correlate `under_path` with related SQL functions (`path` and `depth`).

> **Note:** If a path is not under the specified `pathname` argument, a `NULL` value is returned as the output of the current path.

Figure 22–6 illustrates the `path` syntax.

Figure 22–6 PATH Syntax



DEPTH SQL Function

SQL function `depth` returns the folder depth of the resource under the specified starting path.

```
depth(correlation);
```

where:

`correlation` is an integer that can be used to correlate `under_path` with related SQL functions (`path` and `depth`).

Using the RESOURCE_VIEW and PATH_VIEW APIs

The following `RESOURCE_VIEW` and `PATH_VIEW` examples use SQL functions `under_path`, `equals_path`, `path`, and `depth`.

Accessing Repository Data Paths, Resources and Links: Examples

The following examples illustrate how you can access paths, resources, and link properties in Oracle XML DB Repository. The first few examples use resources specified by the following paths:

```
/a/b/c
/a/b/c/d
/a/e/c
/a/e/c/d
```

Example 22–1 Determining Paths Under a Path: Relative

This example uses SQL function `path` to retrieve the *relative* paths under path `/a/b`.

```
SELECT path(1) FROM RESOURCE_VIEW WHERE under_path(RES, '/a/b', 1) = 1;
```

Returns the following:

```
PATH(1)
-------
c
c/d

2 rows selected.
```

Example 22–2 Determining Paths Under a Path: Absolute

This example uses `ANY_PATH` to retrieve the *absolute* paths under path `/a/b`.

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE under_path(RES, '/a/b') = 1;
```

This returns the following:

```
ANY_PATH
--------
/a/b/c
/a/b/c/d

2 rows selected.
```

Example 22–3 Determining Paths Not Under a Path

This is the same example as Example 22–2, except that the test is *not*-equals (!=) instead of equals (=). This query finds *all paths in the repository* that are *not* under path /a/b.

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE under_path(RES, '/a/b') != 1
```

This produces a result like the following:

```
ANY_PATH
--------
/a
/a/b
/a/e
/a/e/c
/a/e/c/d
/home
/home/OE
/home/OE/PurchaseOrders
/home/OE/PurchaseOrders/2002
/home/OE/PurchaseOrders/2002/Apr
/home/OE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/OE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/OE/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
. . .
/public
/sys
/sys/acls
/sys/acls/all_all_acl.xml
/sys/acls/all_owner_acl.xml
/sys/acls/bootstrap_acl.xml
/sys/acls/ro_all_acl.xml
/sys/apps
/sys/databaseSummary.xml
/sys/log
/sys/schemas
/sys/schemas/OE
/sys/schemas/OE/localhost:8080
. . .

202 rows selected.
```

Example 22–4 Determining Paths Using Multiple Correlations

```
SELECT ANY_PATH, path(1), path(2)
  FROM RESOURCE_VIEW
  WHERE under_path(RES, '/a/b', 1) = 1 OR under_path(RES, '/a/e', 2) = 1;
```

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 selected.
```

To obtain all of the resources under a directory, you can use LIKE, as shown in Example 22–5. To obtain all of the resources up to a certain number of levels or to

obtain the relative path, use SQL function `under_path`, as shown in Example 22–7. Example 22–5 is more effiicient than Example 22–7.

Example 22–5 Using ANY_PATH with LIKE

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE ANY_PATH LIKE '/sys%';
```

This produces a result like the following:

```
ANY_PATH
--------
/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/databaseSummary.xml
/sys/log
/sys/schemas
/sys/schemas/OE
/sys/schemas/OE/localhost:8080
/sys/schemas/OE/localhost:8080/source
/sys/schemas/OE/localhost:8080/source/schemas
/sys/schemas/OE/localhost:8080/source/schemas/poSource
/sys/schemas/OE/localhost:8080/source/schemas/poSource/xsd
/sys/schemas/OE/localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd
/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
. . .

42 rows selected.
```

Example 22–6 Relative Path Names for Three Levels of Resources

```
SELECT path(1) FROM RESOURCE_VIEW WHERE under_path (RES, 3, '/sys', 1) = 1;
```

This produces a result like the following:

```
PATH(1)
-------
acls
acls/all_all_acl.xml
acls/all_owner_acl.xml
acls/bootstrap_acl.xml
acls/ro_all_acl.xml
apps
databaseSummary.xml
log
schemas
schemas/OE
schemas/OE/localhost:8080
schemas/PUBLIC
schemas/PUBLIC/www.w3.org
schemas/PUBLIC/xmlns.oracle.com

14 rows selected.
```

Example 22–7 Extracting Resource Metadata using UNDER_PATH

```
SELECT ANY_PATH, extract(RES, '/Resource') FROM RESOURCE_VIEW
   WHERE under_path(RES, '/sys') = 1;
```

This produces a result like the following:

```
ANY_PATH
--------
EXTRACT(RES,'/RESOURCE')
------------------------
/sys/acls
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd">
  <CreationDate>2005-02-07T18:31:53.093179</CreationDate>
  <ModificationDate>2005-02-07T18:31:55.852963</ModificationDate>
  <DisplayName>acls</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>ISO-8859-1</CharacterSet>
  <ContentType>application/octet-stream</ContentType>
  <RefCount>1</RefCount>
</Resource>

/sys/acls/all_all_acl.xml
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd">
  <CreationDate>2005-02-07T18:31:55.745970</CreationDate>
  <ModificationDate>2005-02-07T18:31:55.745970</ModificationDate>
  <DisplayName>all_all_acl.xml</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>ISO-8859-1</CharacterSet>
  <ContentType>text/xml</ContentType>
  <RefCount>1</RefCount>
</Resource>
. . .
41 rows selected.
```

Example 22–8 Using Functions PATH and DEPTH with PATH_VIEW

```
SELECT path(1) path, depth(1) depth FROM PATH_VIEW
  WHERE under_path(RES, 3, '/sys', 1) = 1;
```

This produces a result like the following:

```
PATH                                     DEPTH
----                                     -----
acls                                     1
acls/all_all_acl.xml                     2
acls/all_owner_acl.xml                   2
acls/bootstrap_acl.xml                   2
acls/ro_all_acl.xml                      2
apps                                     1
databaseSummary.xml                      1
log                                      1
schemas                                  1
schemas/OE                               2
schemas/OE/localhost:8080                3
schemas/PUBLIC                           2
schemas/PUBLIC/www.w3.org                3
schemas/PUBLIC/xmlns.oracle.com          3

14 rows selected.
```

Example 22–9 Extracting Link and Resource Information 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%';
```

This produces a result like the following:

```
/sys
sys
/
sys
sys

/sys/acls
acls
sys
acls
acls

/sys/acls/all_all_acl.xml
all_all_acl.xml
acls
all_all_acl.xml
all_all_acl.xml

/sys/acls/all_owner_acl.xml
all_owner_acl.xml
acls
all_owner_acl.xml
all_owner_acl.xml

/sys/acls/bootstrap_acl.xml
bootstrap_acl.xml
acls
bootstrap_acl.xml
bootstrap_acl.xml
. . .

42 rows selected.
```

Example 22–10 All Paths to a Certain Depth Under a Path

```
SELECT path(1) FROM PATH_VIEW WHERE under_path(RES, 3, '/sys', 1) > 0 ;
```

This produces a result like the following:

```
PATH(1)
-------
schemas
acls
log
schemas/PUBLIC
schemas/PUBLIC/xmlns.oracle.com
acls/bootstrap_acl.xml
acls/all_all_acl.xml
acls/all_owner_acl.xml
acls/ro_all_acl.xml
```

```
schemas/PUBLIC/www.w3.org
apps
databaseSummary.xml

12 rows selected.
```

Example 22–11 Using EQUALS_PATH to Locate a Path

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE equals_path(RES, '/sys') > 0;
```

This produces the following result:

```
ANY_PATH
--------
/sys

1 row selected.
```

Example 22–12 Retrieve RESID of a Given Resource

```
SELECT RESID FROM RESOURCE_VIEW
  WHERE extract(RES, '/Resource/Dispname') = 'example';
```

This produces a result like the following:

```
RESID
--------------------------------
F301A10152470252E030578CB00B432B

1 row selected.
```

Example 22–13 Obtaining the Path Name of a Resource from its RESID

```
DECLARE
  resid_example RAW(16);
  path          VARCHAR2(4000);
BEGIN
  SELECT RESID INTO resid_example FROM RESOURCE_VIEW
    WHERE extractValue(RES, '/Resource/DisplayName') = 'example';
  SELECT ANY_PATH INTO path FROM RESOURCE_VIEW WHERE RESID = resid_example;
  DBMS_OUTPUT.put_line('The path is: ' || path);
END;
/
The path is: /public/example

PL/SQL procedure successfully completed.
```

Example 22–14 Folders Under a Given Path

```
SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE under_path(RES, 1, '/sys') = 1
    AND existsNode(RES, '/Resource[@Container="true"]') = 1;
```

This produces a result like the following:

```
ANY_PATH
--------
/sys/acls
/sys/apps
/sys/log
/sys/schemas
```

```
4 rows selected.
```

Example 22–15 Joining RESOURCE_VIEW with an XMLType Table

```
SELECT ANY_PATH, extract(value(e), '/PurchaseOrder/LineItems').getclobval()
  FROM purchaseorder e, RESOURCE_VIEW r
  WHERE extractValue(r.RES, '/Resource/XMLRef') = ref(e) AND ROWNUM < 2;
```

This produces the following result:

```
ANY_PATH
--------------------------------------------------------------------------------
EXTRACT(VALUE(E),'/PURCHASEORDER/LINEITEMS').GETCLOBVAL()
--------------------------------------------------------------------------------
/home/OE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
<LineItems>
  <LineItem ItemNumber="1">
    <Description>Salesman</Description>
    <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>Big Deal on Madonna Street</Description>
    <Part Id="37429155424" UnitPrice="29.95" Quantity="1"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>Hearts and Minds</Description>
    <Part Id="37429166321" UnitPrice="39.95" Quantity="1"/>
  </LineItem>

. . .

  <LineItem ItemNumber="23">
    <Description>Great Expectations</Description>
    <Part Id="37429128022" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
</LineItems>

1 row selected.
```

Deleting Repository Resources: Examples

The following examples illustrate how you can delete resources and paths.

Example 22–16 Deleting Resources

If you delete only *leaf* resources, then you can use `DELETE FROM RESOURCE_VIEW`:

```
DELETE FROM RESOURCE_VIEW WHERE ANY_PATH = '/public/myfile';
```

For multiple links to the same resource, deleting from `RESOURCE_VIEW` deletes the resource together with *all* of its links; deleting from `PATH_VIEW` deletes only the link with the specified path.

Example 22–17 Deleting Links to Resources

For example, suppose `'/home/myfile1'` is a link to `'/public/myfile'`:

```
CALL DBMS_XDB.link('/public/myfile', '/home', 'myfile1');
```

This SQL DML statement deletes everything in Oracle XML DB Repository that is found at path `/home/myfile1` – both the link and the resource:

```
DELETE FROM RESOURCE_VIEW WHERE equals_path(RES, '/home/myfile1') = 1;
```

This DML statement deletes *only the link* with path `/home/file1`:

```
DELETE FROM PATH_VIEW WHERE equals_path(RES, '/home/file1') = 1;
```

Deleting Nonempty Folder Resources

The `DELETE` DML operator is not allowed on a nonempty folder. If you try to delete a nonempty folder, you must first delete its contents and then delete the resulting empty folder. This rule must of course be applied recursively to any folders contained in the target folder.

However, the order of the paths returned from a `WHERE` clause is not guaranteed, and the `DELETE` operator does not allow an `ORDER BY` clause in its table-expression subclause. This means that you *cannot* do the following:

```
DELETE FROM (SELECT 1 FROM RESOURCE_VIEW
               WHERE under_path(RES, '/public', 1) = 1
               ORDER BY depth(1) DESCENDING);
```

Example 22–18 illustrates how to delete a nonempty folder.

Example 22–18 Deleting a Nonempty Folder

In this example, folder `example` is deleted, along with its subfolder `example1`.

```
SELECT PATH FROM PATH_VIEW WHERE under_path(RES, '/home/US1') = 1;

PATH
--------------------------
/home/US1/example
/home/US1/example/example1

2 rows selected.

DECLARE
  CURSOR c1 IS
    SELECT ANY_PATH p FROM RESOURCE_VIEW
      WHERE under_path(RES, '/home/US1', 1) = 1
        AND existsNode(RES, '/Resource[Owner="US1"]') = 1
      ORDER BY depth(1) DESC;
  del_stmt VARCHAR2(500)
    := 'DELETE FROM RESOURCE_VIEW WHERE equals_path(RES, :1)=1';
BEGIN
  FOR r1 IN c1 LOOP
    EXECUTE IMMEDIATE del_stmt USING r1.p;
  END LOOP;
END;
/

PL/SQL procedure successfully completed.

SELECT PATH FROM PATH_VIEW WHERE under_path(RES, '/home/US1') = 1;

no rows selected
```

Note: As always, care should be taken to avoid deadlocks with concurrent transactions when operating on multiple rows.

Updating Repository Resources: Examples

This section illustrates how to update resources and paths.

Example 22–19 Updating a Resource

This example changes the resource at path `/test/HR/example/paper`. This is the complete resource before the update:

```
SELECT r.RES.getClobVal()
  FROM RESOURCE_VIEW r WHERE equals_path(RES, '/test/HR/example/paper') = 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>2005-04-29T16:30:01.588835</CreationDate>
  <ModificationDate>2005-04-29T16:30:01.588835</ModificationDate>
  <DisplayName>paper</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>ISO-8859-1</CharacterSet>
  <ContentType>application/octet-stream</ContentType>
  <RefCount>1</RefCount>
  <ACL>
    <acl description="Public:All privileges to PUBLIC" xmlns="http://xmlns.oracl
e.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:sch
emaLocation="http://xmlns.oracle.com/xdb/acl.xsd                          http:
//xmlns.oracle.com/xdb/acl.xsd">
      <ace>
        <principal>PUBLIC</principal>
        <grant>true</grant>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>TESTUSER1</Owner>
  <Creator>TESTUSER1</Creator>
  <LastModifier>TESTUSER1</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#binary</SchemaElement
>
  <Contents>
    <binary>4F7261636C65206F7220554E4958</binary>
  </Contents>
</Resource>

1 row selected.
```

All of the XML elements shown above are resource *metadata* elements, with the exception of `Contents`, which contains the resource *content*.

This `UPDATE` statement updates (only) the `DisplayName` metadata element.

```
UPDATE RESOURCE_VIEW r
  SET r.RES = updateXML(r.RES, '/Resource/DisplayName/text()', 'My New Paper')
```

```
    WHERE ANY_PATH='/test/HR/example/paper';

1 row updated.

SELECT r.RES.getClobVal()
  FROM RESOURCE_VIEW r WHERE equals_path(RES, '/test/HR/example/paper') = 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>2005-04-29T16:30:01.588835</CreationDate>
  <ModificationDate>2005-04-29T16:30:01.883838</ModificationDate>
  <DisplayName>My New Paper</DisplayName>
  <Language>en-US</Language>

  . . .

  <Contents>
    <binary>4F7261636C65206F7220554E4958</binary>
  </Contents>
</Resource>

1 row selected.
```

> **See Also:** Chapter 26, "User-Defined Repository Metadata" for additional examples of updating resource metadata

Note that, by default, the `DisplayName` element content, `paper`, was the same text as the last location step of the resource path, `/test/HR/example/paper`. This is only the default value, however. The `DisplayName` is independent of the resource path, so updating it does not change the path.

Element `DisplayName` is defined by the WebDAV standard, and it is recognized by WebDAV applications. Non-WebDAV applications, such as an FTP client, will not recognize the `DisplayName` of a resource. An FTP client lists the above resource as `paper` (using FTP command `ls`, for example) even after the `UPDATE` operation.

Example 22–20 Updating a Path in the PATH_VIEW

This example changes the path for the above resource from `/test/HR/example/paper` to `/test/myexample`. It is analogous to using the Unix command `mv /test/HR/example/paper /test/myexample`.

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE under_path(RES, '/test') = 1;

ANY_PATH
--------
/test/HR
/test/HR/example
/test/HR/example/paper

3 rows selected.

UPDATE PATH_VIEW
  SET PATH = '/test/myexample' WHERE PATH = '/test/HR/example/paper';

ANY_PATH
--------
```

```
/test/HR
/test/HR/example
/test/myexample

3 rows selected.
```

> **See Also:** Table 20–3, " Accessing Oracle XML DB Repository: API Options" on page 20-13 for additional examples that use the SQL functions that apply to RESOURCE_VIEW and PATH_VIEW

Working with Multiple Oracle XML DB Resources

The repository operations listed in Table 20–3 typically apply to a single resource at a time. To perform the same operation on multiple Oracle XML DB resources, or to find one or more Oracle XML DB resources that meet a certain set of criteria, use SQL with RESOURCE_VIEW and PATH_VIEW.

For example, you can perform the following operations:

- Updating based on attributes – see Example 22–21
- Finding resources inside a folder – see Example 22–22
- Copying a set of Oracle XML DB resources – see Example 22–23

Example 22–21 Updating Resources Based on Attributes

```
UPDATE RESOURCE_VIEW
  SET RES = updateXML(RES, '/Resource/DisplayName/text()', 'My New Paper')
    WHERE extractValue(resource, '/Resource/DisplayName') = 'My Paper';

SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE extractValue(RES, '/Resource/DisplayName') = 'My New Paper';

ANY_PATH
---------------
/test/myexample

1 row selected.
```

Example 22–22 Finding Resources Inside a Folder

```
SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE under_path(resource, '/sys/schemas/PUBLIC/xmlns.oracle.com/xdb') = 1;

ANY_PATH
--------------------------------------------------------------
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBFolderListing.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResource.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBSchema.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBStandard.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/dav.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/ftplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/httplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/xdblog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/stats.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/xdbconfig.xsd

12 rows selected.
```

Example 22–23 Copying Resources

This SQL DML statement copies all of the resources in folder `public` to folder `newlocation`. It is analogous to the Unix command `cp /public/* /newlocation`. Target folder `newlocation` must exist before the copy.

```
SELECT PATH FROM PATH_VIEW WHERE under_path(RES, '/test') = 1;

PATH
-----------------
/test/HR
/test/HR/example
/test/myexample

3 rows selected.

INSERT INTO PATH_VIEW
  SELECT '/newlocation/' || path(1), RES, LINK, NULL FROM PATH_VIEW
    WHERE under_path(RES, '/test', 1) = 1
    ORDER BY depth(1);

3 rows created.

SELECT PATH FROM PATH_VIEW WHERE under_path(RES, '/newlocation') = 1;

PATH
------------------------
/newlocation/HR
/newlocation/HR/example
/newlocation/myexample

3 rows selected.
```

Performance Tuning of Oracle XML DB Resource Queries

Oracle XML DB uses the `xdbconfig.xml` file for configuring the system and protocol environment. It includes an element parameter `resource-view-cache-size` that defines the in-memory size of the `RESOURCE_VIEW` cache. The default value is `1048576`.

The performance of some queries on `RESOURCE_VIEW` and `PATH_VIEW` can be improved by tuning `resource-view-cache-size`. In general, the bigger the cache size, the faster the query. The default `resource-view-cache-size` is appropriate for most cases, but you may want to enlarge your `resource-view-cache-size` element when querying a sizable `RESOURCE_VIEW`.

The extensible optimizer decides whether SQL functions `under_path` and `equals_path` are evaluated by a domain index scan or by functional implementation. To achieve the optimal query plan, the optimizer needs statistics for Oracle XML DB. Statistics can be collected by analyzing the Oracle 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;
```

The default limits for the following elements are soft limits. The system automatically adapts when these limits are exceeded.

- XDBCORE-LOADABLEUNIT-SIZE - This element indicates the maximum size to which a loadable unit (partition) can grow in Kilobytes. When a partition is read into memory or a partition is built while consuming a new document, the partition is built until it reaches the maximum size. The default value is 16 Kb.
- XDBCORE-XOBMEM-BOUND - This element indicates the maximum memory in kilobytes that a document is allowed to occupy. The default value is 1024 Kb. Once the document exceeds this number, some loadable units (partitions) are swapped out.

See Also: Chapter 28, "Administering Oracle XML DB"

Searching for Resources Using Oracle Text

Table XDB$RESOURCE in database schema XDB stores the metadata and content of repository resources. You can search for resources that contain a specific keyword by using SQL function contains with RESOURCE_VIEW or PATH_VIEW.

Example 22–24 Find All Resources Containing "Paper"

```
SELECT PATH FROM PATH_VIEW WHERE contains(RES, 'Paper') > 0;

PATH
-----------------------
/newlocation/myexample
/test/myexample

2 rows selected.
```

Example 22–25 Find All Resources Containing "Paper" that are Under a Specified Path

```
SELECT ANY_PATH FROM RESOURCE_VIEW
   WHERE contains(RES, 'Paper') > 0 AND under_path(RES, '/test') > 0;

ANY_PATH
----------------
/test/myexample

1 row selected.
```

To evaluate such queries, you must first create a Context Index on the XDB$RESOURCE table. Depending on the type of documents stored in Oracle XML DB, choose one of the following options for creating your Context Index:

- *If Oracle XML DB contains only XML documents*, that is, no binary data, a regular Context Index can be created on the XDB$RESOURCE table. This is the case for Example 22–25.

  ```
  CREATE INDEX xdb$resource_ctx_i ON XDB.XDB$RESOURCE(OBJECT_VALUE)
    INDEXTYPE IS CTXSYS.CONTEXT;
  ```

  See Also: Chapter 4, "XMLType Operations" and Chapter 10, "Full-Text Search Over XML"

- *If Oracle XML DB contains binary data* such as Microsoft Word documents, a user filter is required to filter such documents prior to indexing. Use package DBMS_XDBT (dbmsxdbt.sql) to create and configure the Context Index.

  ```
  -- Install the package - connected as SYS
  @dbmsxdbt
  ```

```
-- Create the preferences
EXEC DBMS_XDBT.createPreferences;
-- Create the index
EXEC DBMS_XDBT.createIndex;
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*, for information on installing and using `DBMS_XDBT`.
- "APIs for XML" on page 1-5

Package `DBMS_XDBT` also includes procedures to synchronize and optimize the index. You can use procedure `configureAutoSync()` to configure automatically sync the index by using job queues.

23

PL/SQL Access Using DBMS_XDB

This chapter describes the Oracle XML DB resource application program interface (API) for PL/SQL (PL/SQL package `DBMS_XDB`). It contains these topics:

Overview of PL/SQL Package DBMS_XDB

PL/SQL package `DBMS_XDB` is the Oracle XML DB resource application program interface (API) for PL/SQL. It is also known as the PL/SQL *foldering* API. This API provides functions and procedures to access and manage Oracle XML DB Repository resources using PL/SQL. It includes methods for managing resource security and Oracle XML DB configuration.

Oracle XML DB Repository is modeled on XML, and provides a database file system for any data. The repository maps path names (or URLs) onto database objects of `XMLType` and provides management facilities for these objects.

PL/SQL package `DBMS_XDB` is an API that you can use to manage all of the following:

- Oracle XML DB resources
- Oracle XML DB access control list-based Security. An ACL is a list of access control entries that determine which principals have access to which resources
- Oracle XML DB configuration

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*
- "APIs for XML" on page 1-5

DBMS_XDB: Resource Management

Table 23–1 describes the `DBMS_XDB` Oracle XML DB resource management functions and procedures.

Table 23–1 DBMS_XDB Resource Management Functions and Procedures

Function/Procedure	Description
`appendResourceMetadata`	Adds user-defined metadata to a resource.

Table 23–1 (Cont.) DBMS_XDB Resource Management Functions and Procedures

Function/Procedure	Description
`createFolder`	Creates a new folder resource.
`createOIDPath`	Creates a virtual path to a resource, based on its object identifier (OID).
`createResource`	Creates a new file resource.
`deleteResource`	Deletes a resource from the repository.
`deleteResourceMetadata`	Deletes specific user-defined metadata from a resource.
`existsResource`	Indicates whether or not a resource exists, given its absolute path.
`getLockToken`	Returns a resource lock token for the current user, given a path to the resource.
`getResOID`	Returns the object identifier (OID) of a resource, given its absolute path.
`getXDB_tablespace`	Returns the current tablespace of user `XDB`.
`link`	Creates a link to an existing resource.
`lockResource`	Obtains a WebDAV-style lock on a resource, given a path to the resource.
`moveXDB_tablespace`	Moves user XDB to the specified tablespace.
`purgeResourceMetadata`	Deletes all user-defined metadata from a resource.
`rebuildHierarchicalIndex`	Rebuilds the repository hierarchical index, after import or export operations.
`renameResource`	Renames a resource.
`unlockResource`	Unlocks a resource, given its lock token and path.
`updateResourceMetadata`	Modifies user-defined resource metadata.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

The examples in this section illustrate the use of these functions and procedures.

Example 23–1 Using DBMS_XDB to Manage Resources

This example uses package `DBMS_XDB` to manage repository resources. It creates the following:

- a folder, `mydocs`, under folder `/public`
- two file resources, `emp_selby.xml` and `emp_david.xml`
- two links to the file resources, `person_selby.xml` and `person_david.xml`

It then deletes each of the newly created resources and links. The folder contents are deleted before the folder itself.

```
DECLARE
  retb BOOLEAN;
BEGIN
  retb := DBMS_XDB.createfolder('/public/mydocs');
  retb := DBMS_XDB.createresource('/public/mydocs/emp_selby.xml',
                                  '<emp_name>selby</emp_name>');
  retb := DBMS_XDB.createresource('/public/mydocs/emp_david.xml',
                                  '<emp_name>david</emp_name>');
END;
/
PL/SQL procedure successfully completed.

CALL DBMS_XDB.link('/public/mydocs/emp_selby.xml',
```

```
                  '/public/mydocs',
                  'person_selby.xml');
Call completed.

CALL DBMS_XDB.link('/public/mydocs/emp_david.xml',
                  '/public/mydocs',
                  'person_david.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs/emp_selby.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs/person_selby.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs/emp_david.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs/person_david.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs');
Call completed.
```

> **See Also:** Chapter 26, "User-Defined Repository Metadata" for examples using `appendResourceMetadata` and `deleteResourceMetadata`

DBMS_XDB: ACL-Based Security Management

Table 23–2 lists the `DBMS_XDB` Oracle XML DB ACL- based security management functions and procedures.

Table 23–2 DBMS_XDB: Security Management Procedures and Functions

Function/Procedure	Description
`ACLCheckPrivileges`	Checks the access privileges granted to the current user by an ACL.
`changePrivileges`	Adds an ACE to a resource ACL.
`checkPrivileges`	Checks the access privileges granted to the current user for a resource.
`getACLDocument`	Retrieves the ACL document that protects a resource, given the path name of the resource.
`getPrivileges`	Returns all privileges granted to the current user for a resource.
`setACL`	Sets the ACL on a resource.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle XML Developer's Kit Programmer's Guide*

The examples in this section illustrate the use of these functions and procedures.

Example 23–2 Using Procedure DBMS_XDB.getACLDocument

In this example, database sample-schema user `hr` creates two resources: a folder, `/public/mydocs`, with a file in it, `emp_selby.xml`. Procedure `getACLDocument` is called on the file resource, showing that the `<principal>` user for the document is `PUBLIC`.

```
CONNECT HR/HR
Connected.

DECLARE
  retb BOOLEAN;
BEGIN
  retb := DBMS_XDB.createFolder('/public/mydocs');
  retb := DBMS_XDB.createResource('/public/mydocs/emp_selby.xml',
                                  '<emp_name>selby</emp_name>');
END;
/
PL/SQL procedure successfully completed.

SELECT DBMS_XDB.getACLDocument('/public/mydocs/emp_selby.xml').getClobVal()
  FROM DUAL;

DBMS_XDB.GETACLDOCUMENT('/PUBLIC/MYDOCS/EMP_SELBY.XML').GETCLOBVAL()
--------------------------------------------------------------------------------
<acl description="Public:All privileges to PUBLIC" xmlns="http://xmlns.oracle.co
m/xdb/acl.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaL
ocation="http://xmlns.oracle.com/xdb/acl.xsd                          http://xm
lns.oracle.com/xdb/acl.xsd">
  <ace>
    <principal>PUBLIC</principal>
    <grant>true</grant>
    <privilege>
      <all/>
    </privilege>
  </ace>
</acl>

1 row selected.
```

Example 23–3 Using Procedure DBMS_XDB.setACL

In this example, the system manager connects and uses procedure `setACL` to give the owner (`hr`) all privileges on the file resource created in Example 23–2. Procedure `getACLDocument` then shows that the `<principal>` user is `dav:owner`, the owner (`hr`).

```
CONNECT SYSTEM/MANAGER
Connected.

-- Give all privileges to owner, HR.
CALL DBMS_XDB.setACL('/public/mydocs/emp_selby.xml',
                     '/sys/acls/all_owner_acl.xml');
Call completed.
COMMIT;
Commit complete.

SELECT DBMS_XDB.getACLDocument('/public/mydocs/emp_selby.xml').getClobVal()
  FROM DUAL;

DBMS_XDB.GETACLDOCUMENT('/PUBLIC/MYDOCS/EMP_SELBY.XML').GETCLOBVAL()
```

```
--------------------------------------------------------------------------------
<acl description="Private:All privileges to OWNER only and not accessible to oth
ers" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:" xmlns:xsi="htt
p://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.
com/xdb/acl.xsd                          http://xmlns.oracle.com/xdb/acl.xsd">
  <ace>
    <principal>dav:owner</principal>
    <grant>true</grant>
    <privilege>
      <all/>
    </privilege>
  </ace>
</acl>

1 row selected.
```

Example 23–4 Using Function DBMS_XDB.changePrivileges

In this example, user `hr` connects and uses function `changePrivileges` to add a new access control entry (ACE) to the ACL, which gives all privileges on resource `emp_selby.xml` to user `oe`. Procedure `getACLDocument` shows that the new ACE was added to the ACL.

```
CONNECT HR/HR
Connected.

SET SERVEROUTPUT ON

-- Add an ACE giving privileges to user OE
DECLARE
  r        PLS_INTEGER;
  ace      XMLType;
  ace_data VARCHAR2(2000);
BEGIN
  ace_data := '<ace xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                        http://xmlns.oracle.com/xdb/acl.xsd
                                        DAV:http://xmlns.oracle.com/xdb/dav.xsd">
                <principal>OE</principal>
                <grant>true</grant>
                <privilege><all/></privilege>
              </ace>';
  ace := XMLType.createXML(ace_data);
  r := DBMS_XDB.changePrivileges('/public/mydocs/emp_selby.xml', ace);
END;
/

PL/SQL procedure successfully completed.

SELECT DBMS_XDB.getACLDocument('/public/mydocs/emp_selby.xml').getClobVal()
  FROM DUAL;

DBMS_XDB.GETACLDOCUMENT('/PUBLIC/MYDOCS/EMP_SELBY.XML').GETCLOBVAL()
--------------------------------------------------------------------------------
<acl description="Private:All privileges to OWNER only and not accessible to oth
ers" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:" xmlns:xsi="htt
p://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.
com/xdb/acl.xsd                          http://xmlns.oracle.com/xdb/acl.xsd" s
hared="false">
  <ace>
```

```
    <principal>dav:owner</principal>
    <grant>true</grant>
    <privilege>
      <all/>
    </privilege>
  </ace>
  <ace>
    <principal>OE</principal>
    <grant>true</grant>
    <privilege>
      <all/>
    </privilege>
  </ace>
</acl>

1 row selected.
```

Example 23–5 Using Function DBMS_XDB.changePrivileges

In this example, user `oe` connects and calls `DBMS_XDB.getPrivileges`, which shows all of the privileges granted to user `oe` on resource `emp_selby.xml`.

```
CONNECT OE/OE
Connected.

SELECT DBMS_XDB.getPrivileges('/public/mydocs/emp_selby.xml') FROM DUAL;

DBMS_XDB.GETPRIVILEGES('/PUBLIC/MYDOCS/EMP_SELBY.XML').GETCLOBVAL()
--------------------------------------------------------------------------------
<privilege xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl
.xsd http://xmlns.oracle.com/xdb/acl.xsd DAV: http://xmlns.oracle.com/xdb/dav.xs
d" xmlns:xdbacl="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:">
  <read-properties/>
  <read-contents/>
  <update/>
  <link/>
  <unlink/>
  <read-acl/>
  <write-acl-ref/>
  <update-acl/>
  <resolve/>
  <link-to/>
  <unlink-from/>
  <dav:lock/>
  <dav:unlock/>
</privilege>

1 row selected.
```

DBMS_XDB: Configuration Management

Table 23–3 lists the `DBMS_XDB` Oracle XML DB configuration management functions and procedures.

Table 23–3 DBMS_XDB: Configuration Management Functions and Procedures

Function/Procedure	Description
`cfg_get`	Returns the configuration information for the current session.
`cfg_refresh`	Refreshes the session configuration information using the current Oracle XML DB configuration file, `xdbconfig.xml`.
`cfg_update`	Updates the Oracle XML DB configuration information. This writes the configuration file, `xdbconfig.xml`.
`getFTPPort`	Returns the current FTP port number.
`getHTTPPort`	Returns the current HTTP port number.
`setFTPPort`	Sets the Oracle XML DB FTP port to the specified port number.
`setHTTPPort`	Sets the Oracle XML DB HTTP port to the specified port number.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

The examples in this section illustrate the use of these functions and procedures.

Example 23–6 Using Function DBMS_XDB.cfg_get

In this example, function `cfg_get` is used to retrieve the Oracle XML DB configuration file, `xdbconfig.xml`.

```
CONNECT SYSTEM/MANAGER
Connected.

SELECT DBMS_XDB.cfg_get() FROM DUAL;

DBMS_XDB.CFG_GET()
--------------------------------------------------------------------------------
<xdbconfig xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd" xmlns:xsi="http://w
ww.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/x
db/xdbconfig.xsd                                    http://xmlns.oracle.com/xdb
/xdbconfig.xsd">
  <sysconfig>
    <acl-max-age>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/>
    <xdbcore-log-level>0</xdbcore-log-level>
    <resource-view-cache-size>1048576</resource-view-cache-size>
    <protocolconfig>
      <common>
        . . .
      </common>
      <ftpconfig>
        . . .
      </ftpconfig>
      <httpconfig>
        <http-port>8000</http-port>
        <http-listener>local_listener</http-listener>
```

```
        <http-protocol>tcp</http-protocol>
        <max-http-headers>64</max-http-headers>
        <max-header-size>16384</max-header-size>
        <max-request-body>2000000000</max-request-body>
        <session-timeout>6000</session-timeout>
        <server-name>XDB HTTP Server</server-name>
        <logfile-path/>
        <log-level>0</log-level>
        <servlet-realm>Basic realm=&amp;quot;XDB&amp;quot;</servlet-realm>
        <webappconfig>
        . . .
        </webappconfig>
      </httpconfig>
    </protocolconfig>
    <xdbcore-xobmem-bound>1024</xdbcore-xobmem-bound>
    <xdbcore-loadableunit-size>16</xdbcore-loadableunit-size>
  </sysconfig>
</xdbconfig>

1 row selected.
```

Example 23–7 Using Procedure DBMS_XDB.cfg_update

This example illustrates the use of procedure `cfg_update`. The current configuration is retrieved as an `XMLType` instance and modified. It is then rewritten using `cfg_update`.

```
DECLARE
  configxml    SYS.XMLType;
  configxml2   SYS.XMLType;
BEGIN
  -- Get the current configuration
  configxml := DBMS_XDB.cfg_get();

  -- Modify the configuration
  SELECT updateXML(
           configxml,
           '/xdbconfig/sysconfig/protocolconfig/httpconfig/http-port/text()',
           '8000',
           'xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd"')
    INTO configxml2 FROM DUAL;

  -- Update the configuration to use the modified version
  DBMS_XDB.cfg_update(configxml2);
END;
/

PL/SQL procedure successfully completed.

SELECT DBMS_XDB.cfg_get() FROM DUAL;

DBMS_XDB.CFG_GET()
--------------------------------------------------------------------------------
<xdbconfig xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd" xmlns:xsi="http://w
ww.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/x
db/xdbconfig.xsd                                    http://xmlns.oracle.com/xdb
/xdbconfig.xsd">
  <sysconfig>
    <acl-max-age>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/>
    <xdbcore-log-level>0</xdbcore-log-level>
    <resource-view-cache-size>1048576</resource-view-cache-size>
    <protocolconfig>
      <common>
      . . .
      </common>
      <ftpconfig>
      . . .
      </ftpconfig>
      <httpconfig>
        <http-port>8000</http-port>
        . . .
      </httpconfig>
    </protocolconfig>
    <xdbcore-xobmem-bound>1024</xdbcore-xobmem-bound>
    <xdbcore-loadableunit-size>16</xdbcore-loadableunit-size>
  </sysconfig>
</xdbconfig>

1 row selected.
```

24

Repository 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:

Overview of Oracle XML DB Resource Security and ACLs

Oracle XML DB maintains object-level security for all resources in Oracle XML DB Repository.

> **Note:** XML objects that are not stored in the 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 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 28, "Administering Oracle XML DB"
- "APIs for XML" on page 1-5

How the ACL-Based Security Mechanism Works

ACLs in Oracle XML DB are themselves XML schema-based resources, stored and managed in Oracle XML DB. Each resource in Oracle XML DB Repository is protected

by an ACL. Before a user performs an operation on a resource, the user privileges on the resource are checked. The set of privileges checked depends on the operation to be performed.

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 who must register global XML schemas.

Other ACLs supplied with Oracle XML DB include the following. Each is protected by the bootstrap ACL.

- `all_all_acl.xml` Grants all privileges to all users
- `all_owner_acl.xml` Grants all privileges to the owner of the resource
- `ro_all_acl.xml` Grants read privileges to all users

All ACLs must conform to the Oracle XML DB ACL XML schema, which is located in the repository at `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd`.

All ACLs are stored in table **`XDB$ACL`**, which is owned by user `XDB`. This is an XML schema-based `XMLType` table. Each row in this table (and therefore each ACL) has a system-generated object identifier (**OID**) that can be accessed as a column named **`OBJECT_ID`**.

Each resource has a property named **`ACLOID`**. The `ACLOID` stores the OID of the ACL that protects the resource. As mentioned, an ACL is itself a resource. 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$ACL` that contains the actual content of the ACL. These two properties form the link between table `XDB$RESOURCE`, which stores Oracle XML DB resources, and table `XDB$ACL`.

> **See Also:** Appendix D, "Oracle-Supplied XML Schemas and Examples" for the ACL XML schema

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 "Integrating Oracle XML DB with LDAP" on page 24-12.

  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 "Access Privileges" on page 24-4.
- ***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 "Access Privileges" on page 24-4 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 `HR` wants to perform the `read` operation on a given resource, then the `read` privileges must be granted to `HR` 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** (database user). 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: More than one grant ACE (or deny ACE) can apply to a particular user because a user can be granted more than one role.

  An Oracle XML DB ACE element has the following properties:

  - Operation: Either `grant` or `deny`
  - Principal: A valid principal, as described previously.
  - Privileges Set: A particular set of privileges to be granted or denied for a particular principal
  - Principal Format (optional): The format of the principal. An LDAP distinguished name (DN), a short name (DB user/role or 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): 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 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 convention: `<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:*** Privileges are checked before a user is allowed to access a resource. This is done by evaluating the resource 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 part of the session, and it is used to match ACEs, which 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.

Access 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 either aggregate (composed of other privileges) or atomic.

When an ACL is stored in Oracle XML DB, the aggregate privileges retain their identity: they are not decomposed into the corresponding leaf privileges. In WebDAV terms, these are aggregate privileges that are not *abstract*.

Atomic Privileges

Table 24–1 lists the atomic privileges supported by Oracle XML DB.

Table 24–1 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`
`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`

Table 24–1 (Cont.) Oracle XML DB Supported Atomic Privileges

Privilege Name	Description	Database Counterpart
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 used as 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 the `XMLType` APIs.

Aggregate Privileges

Table 24–2 lists the aggregate privileges defined by Oracle XML DB, along with the atomic privileges of which they are composed.

Table 24–2 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 24–3 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 24–3 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

Table 24–3 (Cont.) Privileges Needed for Operations on Oracle XML DB Resources

Operation	Description	Privileges Required
GET	An FTP or HTTP(S) retrieval of resource R	read-properties, read-contents on R
SET_ACL	Set the ACL of a resource R	dav:write-acl on R
LIST	List the resources in 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 Oracle XML DB Repository are of two types:

- LOB-based (content is stored in a LOB which is part of the resource). Access is determined only by the ACL that protects the resource.
- REF-based (content is XML and is stored in a database table). Users must have the appropriate privilege in the underlying table (or view) where the XML content is stored, as well as permissions through the ACL for the resource.

Since the content 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 procedure DBMS_XDBZ.enable_hierarchy. This procedure adds two hidden columns to store the ACLOID 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. The default tables produced by XML schema registration are already hierarchy-enabled.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information on procedure DBMS_XDBZ.enable_hierarchy

In a particular table, some, but not all, objects may be mapped to Oracle XML DB resources. Only those objects mapped into Oracle XML DB Repository undergo ACL checking, but they will all have table-level security.

Note: You cannot hide data in XMLType tables from other users when using *out-of-line* storage. Out-of-line data is *not* protected by ACL security.

Working with Oracle XML DB ACLs

Oracle XML DB ACLs are (file) resources, so all of the methods that operate on resources also apply to ACLs. In addition, there are several APIs specific to ACLsin package DBMS_XDB. Those procedures and functions let you use PL/SQL to access Oracle XML DB security mechanisms, check user privileges based on a particular ACL, and list the set of privileges the current user has for a particular ACL and resource.

> **See Also:** Chapter 28, "Administering Oracle XML DB"

Creating an ACL Using DBMS_XDB.createResource

Example 24–1 Creating an ACL Using DBMS_XDB.createResource

This example creates an ACL as file resource `/TESTUSER/acl1.xml`. If applied to a resource, the ACL will grant all privilegest to the owner of a resource.

```
DECLARE
  b BOOLEAN;
BEGIN
  b := DBMS_XDB.createFolder('/TESTUSER');
  b := DBMS_XDB.createResource(
         '/TESTUSER/acl1.xml',
         '<acl description="myacl"
               xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
               xmlns:dav="DAV:"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                   http://xmlns.oracle.com/xdb/acl.xsd">
            <ace>
              <principal>dav:owner</principal>
              <grant>true</grant>
              <privilege>
                <dav:all/>
              </privilege>
            </ace>
          </acl>');
END;
```

Setting the ACL of a Resource

Example 24–2 Setting the ACL of a Resource

This example creates resource `/TESTUSER/po1.xml` and sets its ACL to `/TESTUSER/acl1.xml` using procedure `DBMS_XDB.setACL`.

```
DECLARE
  b BOOLEAN;
BEGIN
  b := DBMS_XDB.createResource('/TESTUSER/po1.xml', 'Hello');
END;
/

CALL DBMS_XDB.setACL('/TESTUSER/po1.xml', '/TESTUSER/acl1.xml');
```

Deleting an ACL

Example 24–3 illustrates how to delete an ACL using procedure `DBMS_XDB.deleteResource`.

Example 24–3 Deleting an ACL

This example deletes the ACL created in Example 24–1.

```
CALL DBMS_XDB.deleteResource('/TESTUSER/acl1.xml');
```

If a resource is being protected by an ACL that you will delete, 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, SQL function `updateXML` and related XML-updating functions 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`. The XPath location for this timout parameter is `/xdbconfig/sysconfig/acl-max-age`; the value is expressed in seconds. Sessions initiated after the ACL is modified use the new ACL without any delay.

If an ACL resource is updated with non-ACL content, the same rules apply as for deletion. Thus, if any resource is being protected by an ACL that is being updated, you must first change the ACL.

> **See Also:** "Updating XML Instances and XML Data in Tables" on page 4-15 for information on the SQL functions used here to update XML data

You can use FTP or WebDAV to update an ACL. For more details on how to use these protocols, see Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data". You can update an ACL or an access control entry (ACE) using `RESOURCE_VIEW`.

Example 24–4 Updating (Replacing) an Access Control List

This example uses SQL function `updateXML` to update the ACL `/TESTUSER/acl1.xml` by replacing it entirely. The effect is to replace the principal value `dav:owner` by `TESTUSER`, because the rest of the replacement ACL is the same as it was before.

```
UPDATE RESOURCE_VIEW r
  SET r.RES =
        updateXML(
          r.RES,
          '/r:Resource/r:Contents/a:acl',
          '<acl description="myacl"
                xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
                xmlns:dav="DAV:"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                    http://xmlns.oracle.com/xdb/acl.xsd">
             <ace>
               <principal>TESTUSER</principal>
               <grant>true</grant>
               <privilege>
                 <dav:all/>
               </privilege>
             </ace>
           </acl>',
          'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
           xmlns:a="http://xmlns.oracle.com/xdb/acl.xsd"')
  WHERE r.ANY_PATH='/TESTUSER/acl1.xml';
```

Example 24–5 Appending ACEs to an Access Control List

This example uses SQL function `appendChildXML` to append an ACE to an existing ACL. The ACE gives privileges `read-properties` and `read-contents` to user hr.

```
UPDATE RESOURCE_VIEW r
  SET r.RES =
        appendChildXML(
          r.RES,
          '/r:Resource/r:Contents/a:acl',
          XMLType('<ace xmlns="http://xmlns.oracle.com/xdb/acl.xsd">
                     <principal>HR</principal>
                     <grant>true</grant>
                     <privilege>
                       <read-properties/>
                       <read-contents/>
                     </privilege>
                   </ace>'),
                  'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                   xmlns:a="http://xmlns.oracle.com/xdb/acl.xsd"')
  WHERE r.ANY_PATH='/TESTUSER/acl1.xml';
```

Example 24–6 Deleting an ACE from an Access Control List

This examples uses SQL function `deleteXML` to delete an ACE from an ACL.

```
UPDATE RESOURCE_VIEW r
  SET r.RES =
        deleteXML(r.RES,
                  '/r:Resource/r:Contents/a:acl/a:ace',
                  'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                   xmlns:a="http://xmlns.oracle.com/xdb/acl.xsd"')
  WHERE r.ANY_PATH='/TESTUSER/acl1.xml';
```

Retrieving the ACL Document for a Given Resource

Example 24–7 illustrates how to use function `DBMS_XDB.getACLDocument` to retrieve the ACL document for a given resource.

Example 24–7 Retrieving the ACL Document for a Resource

```
SELECT DBMS_XDB.getACLDocument('/TESTUSER/po1.xml').getClobVal() FROM DUAL;

DBMS_XDB.GETACLDOCUMENT('/TESTUSER/PO1.XML').GETCLOBVAL()
--------------------------------------------------------------------------------
<acl description="myacl" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="
DAV:" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://xmlns.oracle.com/xdb/acl.xsd                                      http://x
mlns.oracle.com/xdb/acl.xsd">
  <ace>
    <principal>TESTUSER</principal>
    <grant>true</grant>
    <privilege>
      <dav:all/>
    </privilege>
  </ace>
  <ace xmlns="http://xmlns.oracle.com/xdb/acl.xsd">
    <principal>HR</principal>
    <grant>true</grant>
    <privilege>
      <read-properties/>
      <read-contents/>
    </privilege>
  </ace>
</acl>
```

```
1 row selected.
```

Retrieving Privileges Granted to the Current User for a Particular Resource

Example 24–8 illustrates how to retrieve privileges granted to the current user using function `DBMS_XDB.getPrivileges`.

Example 24–8 Retrieving Privileges Granted to the Current User for a Particular Resource

```
SELECT DBMS_XDB.getPrivileges('/TESTUSER/po1.xml').getClobVal() FROM DUAL;

DBMS_XDB.GETPRIVILEGES('/TESTUSER/PO1.XML').GETCLOBVAL()
--------------------------------------------------------------------------------
<privilege xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl
.xsd http://xmlns.oracle.com/xdb/acl.xsd DAV: http://xmlns.oracle.com/xdb/dav.xs
d" xmlns:xdbacl="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:">
  <read-properties/>
  <read-contents/>
  <update/>
  <link/>
  <unlink/>
  <read-acl/>
  <write-acl-ref/>
  <update-acl/>
  <resolve/>
  <unlink-from/>
  <dav:lock/>
  <dav:unlock/>
</privilege>

1 row selected.
```

Checking if the Current User Has Privileges on a Resource

Example 24–9 illustrates how to use function `DBMS_XDB.checkPrivileges` 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 24–9 Checking If a User Has a Certain Privileges on a Resource

This example checks to see if the access privileges `read-contents` and `read-properties` have been granted to the current user on resource `/TESTUSER/po1.xml`. The positive-integer return value shows that they have.

```
SELECT DBMS_XDB.checkPrivileges(
         '/TESTUSER/po1.xml',
         XMLType('<privilege
                     xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
                     xmlns:dav="DAV:"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                         http://xmlns.oracle.com/xdb/acl.xsd">
                   <read-contents/>
                   <read-properties/>
                 </privilege>'))
  FROM DUAL;

DBMS_XDB.CHECKPRIVILEGES('/TESTUSER/PO1.XML',
```

```
-------------------------------------------
                                         1

1 row selected.
```

Checking if the Current User Has Privileges With the ACL and Resource Owner

Function `DBMS_XDB.ACLCheckPrivileges` is typically used by applications that must perform ACL evaluation on their own, before allowing a user to perform an operation.

Example 24–10 Checking User Privileges using ACLCheckPrivileges

This example checks whether the ACL `/TESTUSER/acl1.xml` grants the privileges `read-contents` and `read-properties` to the current user, `sh`. The second argument, `TESTUSER`, is the user that is substituted for `dav:owner` in the ACL when checking. Since user `sh` does *not* match any of the users granted the specified privileges, the return value is zero.

```
CONNECT SH/SH

SELECT DBMS_XDB.ACLCheckPrivileges(
         '/TESTUSER/acl1.xml',
         'TESTUSER',
         XMLType('<privilege
                     xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
                     xmlns:dav="DAV:"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                         http://xmlns.oracle.com/xdb/acl.xsd">
                   <read-contents/>
                   <read-properties/>
                 </privilege>'))
  FROM DUAL;

DBMS_XDB.ACLCHECKPRIVILEGES('/TESTUSER/ACL1.XML','TESTUSER',
------------------------------------------------------------
                                                           0

1 row selected.
```

Retrieving the Path of the ACL that Protects a Given Resource

Example 24–11 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 24–11 Retrieving the Path of the ACL that Protects a Given Resource

This example retrieves the path to an ACL, given a resource protected by the ACL. The ACLOID of a protected resource (`r`) stores the OID of the ACL resource (`a`) that protects it. The `REF` of the ACL resource is the same as that of the object identified by the protected-resource `ACLOID`.

The `REF` of the resource `ACLOID` can be obtained using SQL function `make_ref`, which returns a `REF` to an object-table row with a given OID.

In this example, `make_ref` returns a `REF` to the row of table `XDB$ACL` whose OID is the `/Resource/ACLOID` for the resource `/TESTUSER/po1.xml` (`r`). The inner query

returns the ACLOID of the resource. The outer query returns the path to the corresponding ACL.

```
SELECT a.ANY_PATH
  FROM RESOURCE_VIEW a
  WHERE extractValue(a.RES, '/Resource/XMLRef')
        = make_ref(XDB.XDB$ACL,
                   (SELECT extractValue(r.RES, '/Resource/ACLOID')
                      FROM RESOURCE_VIEW r
                      WHERE equals_path(r.RES, '/TESTUSER/po1.xml') = 1));

ANY_PATH
------------------
/TESTUSER/acl1.xml

1 row selected.
```

Retrieving the Paths of All Resources Protected by a Given ACL

Example 24–12 retrieves the paths of all resources protected by a given ACL.

Example 24–12 Retrieving the Paths of All Resources Protected by a Given ACL

This example retrieves the paths to the resources whose ACLOID REF matches the REF of the ACL resource whose path is /TESTUSER/acl1.xml. Function make_ref returns the resource ACLOID REF.

The inner query retrieves the REF of the specified ACL. The outer query selects the paths of the resources whose ACLOID REF matches the REF of the specified ACL.

```
SELECT r.ANY_PATH
  FROM RESOURCE_VIEW r
  WHERE make_ref(XDB.XDB$ACL, extractValue(r.RES, '/Resource/ACLOID'))
        = (SELECT extractValue(a.RES, '/Resource/XMLRef')
             FROM RESOURCE_VIEW a
             WHERE equals_path(a.RES, '/TESTUSER/acl1.xml') = 1);

ANY_PATH
-----------------
/TESTUSER/po1.xml

1 row selected.
```

Integrating Oracle XML DB with LDAP

This section deals with allowing LDAP users to use the features of Oracle XML DB. The typical scenario is a single, shared database schema, to which multiple LDAP users are mapped. This mapping is maintained in the Oracle Internet Directory. Users can log in to the database using their LDAP username and password; they are then automatically mapped to the corresponding shared schema. (Users can log in using SQL or any of the supported Oracle XML DB protocols.) The implicit ACL resolution is based on the current LDAP user and the corresponding LDAP group membership information.

Before you can use LDAP users and groups as principals in Oracle XML DB ACLs, the following prerequisites must be satisfied:

- An Oracle Internet Directory must be set up, and the database must be registered with it.

- SSL authentication must be set up between the database and the Oracle Internet Directory.
- A database user must be created that corresponds to the shared database schema.
- The LDAP users must be created and mapped in the Oracle Internet Directory to the shared database schema.
- The LDAP groups must be created and their members must be specified.
- ACLs must be defined for the LDAP groups and users, and they must be used to protect the repository resources to be accessed by the LDAP users.

See Also:

- *Oracle Internet Directory Administrator's Guide* for information on setting up the Oracle Internet Directory and registering the database
- *Oracle Database Advanced Security Administrator's Guide* for information on setting up SSL authentication
- *Oracle Database Enterprise User Administrator's Guide* for information on using shared database schemas for enterprise (LDAP) users

Example 24–13 ACL Referencing an LDAP User

This is an example of an ACL for an LDAP user. Element `<principal>` contains the full *distinguished name* of the LDAP user – in this case, `cn=user1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US`.

```
<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>
```

See Also: *Oracle Internet Directory Administrator's Guide* for the format of an LDAP user distinguished name

Example 24–14 ACL Referencing an LDAP Group

This is an example of an ACL for an LDAP group. Element `<principal>` contains the full distinguished name of the 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>
```

See Also: *Oracle Internet Directory Administrator's Guide* for the format of an LDAP group distinguished name

Performance Issues for Using ACLs

Since ACLs are checked for each access to Oracle XML DB Repository, the performance of the ACL check operation is critical to the performance of the repository. In Oracle 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.

Oracle XML DB also maintains caches to improve performance when using ACLs that have LDAP principals (LDAP groups or users). The goal of these caches is to minimize network communication with the LDAP server. One is a shared cache that maps LDAP GUIDs to the corresponding LDAP nicknames and Distinguished Names (DNs). This is used when an ACL document is being displayed (or converted to `CLOB` or `VARCHAR2` values from an `XMLType` instance). To purge this cache, use procedure `DBMS_XDBZ.PurgeLDAPCache`. The other cache is session-specific and maps LDAP groups to their members (nested membership). Note that whenever Oracle XML DB encounters an LDAP group for the first time (in a session) it 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.

25

FTP, HTTP(S), and WebDAV Access to Repository Data

This chapter describes how to access Oracle XML DB Repository data using FTP, HTTP(S)/WebDAV protocols.

This chapter contains these topics:

Overview of Oracle XML DB Protocol Server

As described in Chapter 2, "Getting Started with Oracle XML DB" and Chapter 20, "Accessing Oracle XML DB Repository Data", Oracle XML DB Repository provides a hierarchical data repository in the database, designed for XML. Oracle XML DB Repository maps path names (or URLs) onto database objects of `XMLType` and provides management facilities for these objects.

Oracle XML DB also provides the Oracle XML DB *protocol server*. This supports standard Internet protocols, FTP, WebDAV, and HTTP(S), for accessing its hierarchical repository or file system. Note that HTTPS provides *secure* access to Oracle XML DB Repository.

These protocols can provide direct access to Oracle XML DB for many users without having to install additional software. The user names and passwords to be used with the protocols are the same as those for SQL*Plus. Enterprise users are also supported. DBAs can use these protocols and resource APIs such as `DBMS_XDB` to access Automatic Storage Management (ASM) files and folders in the repository virtual folder `/sys/asm`.

> **See Also:** Chapter 20, "Accessing Oracle XML DB Repository Data" for more information on accessing repository information, and restrictions on that access

Note:

- When accessing virtual folder `/sys/asm` using Oracle XML DB protocols, you must log in as a DBA user other than `SYS`.
- Oracle XML DB protocols are *not* supported on EBCDIC platforms.

Session Pooling

Oracle XML DB protocol server maintains a shared pool of sessions. Each protocol connection is associated with one session from this pool. After a connection is closed the session is put back into the shared pool and can be used to serve later connections.

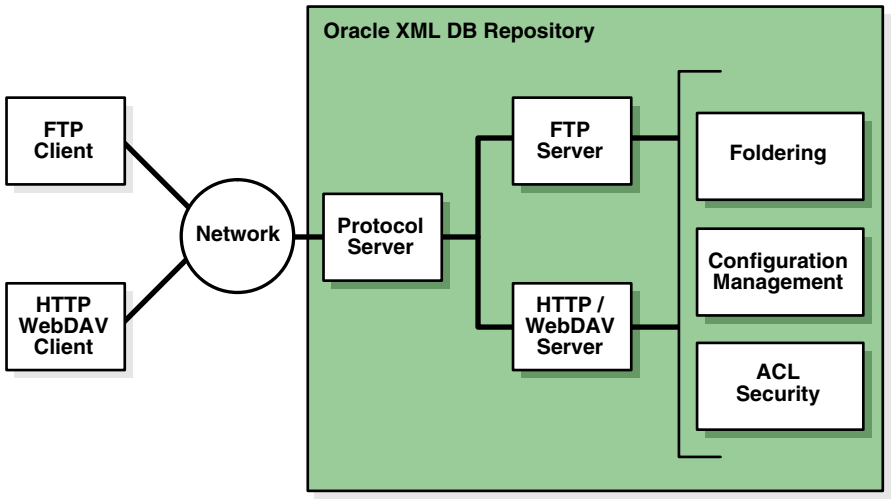
Session pooling improves performance of HTTP(S) by avoiding the cost of re-creating session states, especially when using HTTP 1.0, which creates new connections for each request. For example, a couple of small files can be retrieved by an existing HTTP/1.1 connection in the time necessary to create a database session. You can tune the number of sessions in the pool by setting session-pool-size in Oracle XML DB `xdbconfig.xml` file, or disable it by setting pool size to zero.

Session pooling can affect users writing *Java servlets*, because other users can see session state initialized by another request for a different user. Hence, servlet writers should only use session memory, such as Java static variables, to hold data for the entire application rather than for a particular user. State for each user must be stored in the database or in a look-up table, rather than assuming that a session will only exist for a single user.

See Also: Chapter 27, "Writing Oracle XML DB Applications in Java"

Figure 25–1 illustrates the Oracle XML DB protocol server components and how they are used to access files in Oracle XML DB Repository and other data. Only the relevant components of the repository are shown

Figure 25–1 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: Chapter 28, "Administering Oracle XML DB"

Configuring Protocol Server Parameters

Figure 25–1 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 25–2 shows the FTP-specific parameters. These are relative to the following XPath in the Oracle XML DB configuration schema:

  ```
  /xdbconfig/sysconfig/protocolconfig/ftpconfig
  ```

- **HTTP(S)/WebDAV specific parameters, except servlet-related parameters**. Table 25–3 shows the HTTP(S)/WebDAV-specific parameters. These parameters are relative to the following XPath in the Oracle XML DB configuration schema:

  ```
  /xdbconfig/sysconfig/protocolconfig/httpconfig
  ```

See Also:

- Chapter 28, "Administering Oracle XML DB" for more information about the configuration file `/xdbconfig.xml`
- "xdbconfig.xsd: XML Schema for Configuring Oracle XML DB" on page D-5
- "Configuring Default Namespace to Schema Location Mappings" on page 28-11 for more information about the `schemaLocation-mappings` parameter
- "Configuring XML File Extensions" on page 28-13 for more information about the `xml-extensions` parameter

For examples of the usage of these parameters, see the configuration file, `/xdbconfig.xml`.

Table 25–1 Common Protocol Configuration Parameters

Parameter	Description
`extension-mappings/mime-mappings`	Specifies the mapping of file extensions to mime types. When a resource is stored in Oracle XML DB Repository, and its mime type is not specified, this list of mappings is used to set its mime type.

Table 25–1 (Cont.) Common Protocol Configuration Parameters

Parameter	Description
`extension-mappings/lang-mappings`	Specifies the mapping of file extensions to languages. When a resource is stored in Oracle XML DB Repository, and its language is not specified, this list of mappings is used to set its language.
`extension-mappings/encoding-mappings`	Specifies the mapping of file extensions to encodings. When a resource is stored in Oracle XML DB Repository, and its encoding is not specified, this list of mappings is used to set its encoding.
`xml-extensions`	Specifies the list of filename extensions that are treated as XML content by Oracle XML DB.
`session-pool-size`	Maximum number of sessions that are kept in the protocol server session pool
`/xdbconfig/sysconfig/call-timeout`	If a connection is idle for this time (in hundredths of a second), then the shared server serving the connection is freed up to serve other connections.
`session-timeout`	Time (in hundredths of a second) after which a session (and consequently the corresponding connection) 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 period after which a WebDAV lock on a resource becomes invalid. This could be overridden by a Timeout specified by the client that locks the resource.

Table 25–2 Configuration Parameters Specific to FTP

Parameter	Description
`buffer-size`	Size of the buffer, in bytes, used to read data from the network during an FTP `put` operation. Set `buffer-size` to larger values for higher `put` performance. There is a tradeoff between `put` performance and memory usage. Value can be from 1024 to 1048496, inclusive; the default value is 8192.

Table 25–2 (Cont.) Configuration Parameters Specific to FTP

Parameter	Description
`ftp-port`	Port on which FTP server listens. By default, this is `0`, which means that FTP is *disabled*. FTP is disabled by default because the FTP specification requires that passwords be transmitted in clear text, which can present a security hazard. To enable FTP, set this parameter to the FTP port to use, such as `2100`.
`ftp-protocol`	Protocol over which the FTP server runs. By default, this is `tcp`.
`ftp-welcome-message`	A user-defined welcome message that is displayed whenever an FTP client connects to the server. If this parameter is empty or missing, then the following default welcome message is displayed: "Unauthorized use of this FTP server is prohibited and may be subject to civil and criminal prosecution."
`session-timeout`	Time (in hundredths of a second) after which an FTP connection will be terminated by the protocol server if the connection has been idle for that time.

Table 25–3 Configuration Parameters Specific to HTTP(S)/WebDAV (Except Servlet Parameters)

Parameter	Description
`http-port`	Port on which the HTTP(S)/WebDAV server listens, using protocol `http-protocol`. If this parameter is empty (`<http-port/>`), then the default value of `8080` applies. (An empty parameter is *not* recommended.) This parameter must be present, whether or not it is empty; otherwise, validation of `xdbconfig.xml` against XML schema `xdbconfig.xsd` fails. The value must be different from the value of `http2-port`; otherwise, an error is raised.
`http2-port`	Port on which the HTTP(S)/WebDAV server listens, using protocol `http2-protocol`. This parameter is *optional*, but, if present, then `http2-protocol` must also be present; otherwise, an error is raised. The value must be different from the value of `http-port`; otherwise, an error is raised. An empty parameter (`<http2-port/>`) also raises an error.
`http-protocol`	Protocol over which the HTTP(S)/WebDAV server runs on port `http-port`. Must be either `TCP` or `TCPS`. This parameter must be present; otherwise, validation of `xdbconfig.xml` against XML schema `xdbconfig.xsd` fails. An empty parameter (`<http-protocol/>`) also raises an error.

Table 25–3 (Cont.) Configuration Parameters Specific to HTTP(S)/WebDAV (Except Servlet Parameters)

Parameter	Description
`http2-protocol`	Protocol over which the HTTP(S)/WebDAV server runs on port `http2-port`. Must be either `TCP` or `TCPS`. If this parameter is empty (`<http2-protocol/>`), then the default value of `TCP` applies. (An empty parameter is *not* recommended.) This parameter is *optional*, but, if present, then `http2-port` must also be present; otherwise, an error is raised.
`session-timeout`	Time (in hundredths of a second) after which an HTTP(S) session (and consequently the corresponding connection) 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(S) header
`max-request-body`	Maximum size (in bytes) of an HTTP(S) request body
`webappconfig/welcome-file-list`	List of filenames that are considered welcome files. When an HTTP(S) `get` request for a container is received, the server first checks if there is a resource in the container with any of these names. If so, then the contents of that file are sent, instead of a list of resources in the container.
`default-url-charset`	The character set in which an HTTP(S) protocol server assumes incoming URL is encoded when it is not encoded in UTF-8 or the Content-Type field Charset parameter of the request.
`allow-repository-anonymous-access`	Indication of whether or not anonymous HTTP access to Oracle XML DB Repository data is allowed using an unlocked `ANONYMOUS` user account. The default value is `false`, meaning that unauthenticated access to repository data is *blocked*. See "Anonymous Access to Oracle XML DB Repository using HTTP" on page 25-14.

Configuring Secure HTTP (HTTPS)

To enable Oracle XML DB Repository to use *secure* HTTP connections (HTTPS), a DBA must configure the database accordingly: configure parameters `http2-port` and `http2-protocol`, enable the HTTP Listener to use SSL, and enable launching of the TCPS Dispatcher. After doing this, the DBA must stop, then restart, the database and the listener.

Enable the HTTP Listener to Use SSL

A DBA must carry out the following steps, to configure the HTTP Listener for SSL.

1. **Create a wallet for the server and import a certificate –** Use Oracle Wallet Manager to do the following:
   a. Create a wallet for the server.

b. If a valid certificate with distinguished name (DN) of the server is not available, create a certificate request and submit it to a certificate authority. Obtain a valid certificate from the authority.

c. Import a valid certificate with the distinguished name (DN) of the server into the server.

d. Save the new wallet in *obfuscated* form, so that it can be opened without a password.

See Also: *Oracle Database Advanced Security Administrator's Guide* for information on how to create a wallet

2. **Specify the wallet location to the server –** Use Oracle Net Manager to do this. Ensure that the configuration is saved to disk. This step updates files `sqlnet.ora` and `listener.ora`.
3. **Disable client authentication** at the server, since most Web clients do not have certificates. Use Oracle Net Manager to do this. This step updates file `sqlnet.ora`.
4. **Create a listening end point that uses TCP/IP with SSL –** Use Oracle Net Manager to do this. This step updates file `listener.ora`.

See Also: *Oracle Database Advanced Security Administrator's Guide* for detailed information regarding steps 1 through 4

Enable TCPS Dispatcher

A DBA must edit the database `pfile` to enable launching of a TCPS dispatcher during database startup. The following line must be added to the file, where *`SID`* is the SID of the database:

```
dispatchers=(protocol=tcps)(service=SIDxdb)
```

The database `pfile` location depends on your operating system, as follows:

- **MS Windows –** *`PARENT`*`/admin/orcl/pfile`, where *`PARENT`* is the parent folder of folder `ORACLE_HOME`
- **Unix, Linux –** `$ORACLE_HOME/admin/$ORACLE_SID/pfile`

Interaction with Oracle XML DB File-System Resources

The protocol specifications, RFC 959 (FTP), RFC 2616 (HTTP), and RFC 2518 (WebDAV) implicitly assume an abstract, hierarchical file system on the server side. This is mapped to Oracle XML DB Repository. The repository provides:

- Name resolution.
- Access control list (ACL)-based security. An ACL is a list of access control entries that determine which principals have access to a given resource or resources. See also Chapter 24, "Repository Resource Security".
- The ability to store and retrieve any content. The repository can store both binary data input through FTP and XML schema-based documents.

See Also:

- http://www.ietf.org/rfc/rfc959.txt
- http://www.ietf.org/rfc/rfc2616.txt
- http://www.ietf.org/rfc/rfc2518.txt

Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents

Oracle XML DB protocol server enhances the protocols by always checking if XML documents being inserted are based on XML schemas registered in Oracle XML DB Repository.

- If the incoming XML document specifies an XML schema, then the Oracle XML DB storage to use is determined by that XML schema. This functionality is especially useful when you must store XML documents object-relationally in the database using simple protocols like FTP or WebDAV instead of using SQL statements.
- If the incoming XML document is not XML schema-based, then it is stored as a binary document.

Event-Based Logging

In certain cases, it may be useful to log the requests received and responses sent by a protocol server. This can be achieved by setting event number `31098` to level `2`. To set this event, add the following line to your `init.ora` file and restart the database:

```
event="31098 trace name context forever, level 2"
```

Using FTP and Oracle XML DB Protocol Server

The following sections describe FTP features supported by Oracle XML DB.

Oracle XML DB Protocol Server: FTP Features

File Transfer Protocol (FTP) is one of the oldest and most popular protocols on the net. FTP is specified in RFC959 and provides access to heterogeneous file systems in a uniform manner. FTP works by providing well-defined commands (methods) for communication between the client and the server. The transfer of command messages and the return of status happens on a single connection. However, a new connection is opened between the client and the server for data transfer. With HTTP(S), commands and data are transferred using a single connection.

FTP is implemented by dedicated clients at the operating system level, file-system explorer clients, and browsers. FTP is typically session-oriented: a user session is created through an explicit logon, a number of files or directories are downloaded and browsed, and then the connection is closed.

Note: For security reasons, FTP is *disabled*, by default. This is because the IETF FTP protocol specification requires that passwords be transmitted in clear text. Disabling is done by configuring the FTP server port as zero (0). To enable FTP, set the `ftp-port` parameter to the FTP port to use, such as 2100.

See Also: RFC 959: FTP Protocol Specification – http://www.ietf.org/rfc/rfc959.txt

FTP Features That Are Not Supported

Oracle XML DB implements FTP, as defined by RFC 959, with the *exception* of the following optional features:

- Record-oriented files, for example, only the `FILE` structure of the `STRU` method is supported. This is the most widely used structure for transfer of files. It is also the default specified by the specification. Structure mount is not supported.
- Append.
- Allocate. This pre-allocates space before file transfer.
- Account. This uses the insecure Telnet protocol.
- Abort.

FTP Client Methods That Are Supported

For access to the repository, Oracle XML DB supports the following FTP client methods.

- `cdup` – change working directory to parent directory
- `cwd` – change working directory
- `dele` – delete file (not directory)
- `list`, `nlst` – list files in working directory
- `mkd` – create directory
- `noop` – do nothing (but timeout counter on connection is reset)
- `pasv`, `port` – establish a TCP data connection
- `pwd` – get working directory
- `quit` – close connection and quit FTP session
- `retr` – retrieve data using an established connection
- `rmd` – remove directory
- `rnfr`, `rnto` – rename file (two-step process: from file, to file)
- `stor` – store data using an established connection
- `syst` – get system version
- `type` – change data type: `ascii` or `image` binary types only
- `user`, `pass` – user login

See Also:

- "FTP Quote Methods" for supported FTP `quote` methods
- "Using FTP with ASM Files" on page 25-10 for an example of using FTP method `proxy`

FTP Quote Methods

Oracle Database supports several FTP `quote` methods, which provide information directly to Oracle XML DB.

- **rm_r** – Remove file or folder *<resource_name>*. If a folder, recursively remove all files and folders contained in *<resource_name>*.

  ```
  quote rm_r <resource_name>
  ```

- **rm_f** – Forcibly remove a resource.

  ```
  quote rm_f <resource_name>
  ```

- **rm_rf** – Combines `rm_r` and `rm_f`: Forcibly and recursively removes files and folders.

  ```
  quote rm_rf <resource_name>
  ```

- **set_nls_locale** – Specify the character-set encoding (*<charset_name>*) to be used for file and directory names in FTP methods (including names in method responses).

  ```
  quote set_nls_locale {<charset_name> | NULL}
  ```

  Only IANA character-set names can be specified for this parameter. If `nls_locale` is set to `NULL` or is not set, then the database character set is used.

- **set_charset** – Specify the character set of the data to be sent to the server.

  ```
  quote set_charset  {<charset_name> | NULL}
  ```

  The `set_charset` method applies to only *text* files, not binary files, as determined by the file-extension mapping to MIME types that is defined in configuration file `xdbconfig.xml`.

  If the parameter to `set_charset` is *not* `NULL`, then ***<charset_name>*** is used to determine the character set of the data.

  If the parameter to `set_charset` is `NULL`, or if no `set_charset` command is given, then the *MIME type* of the data determines the character set for the data.

  – If the MIME type is *not* `text/xml`), then the data is not assumed to be XML. The database character set is used.

  – If the MIME type is **`text/xml`**, then the data represents an XML document.

    If a ***byte order mark***[1] (BOM) is present in the XML document, then it determines the character set of the data.

    If there is *no* BOM, then:

    * If there is an ***encoding declaration*** in the XML document, then it determines the character set of the data.

    * If there is *no* encoding declaration, then the UTF-8 character set is used.

Using FTP with ASM Files

Automatic Storage Management (ASM) organizes database files into disk groups for simplified management and added benefits such as database mirroring and I/O balancing. DBAs can use protocols and resource APIs to access ASM files in the Oracle XML DB repository *virtual folder* `/sys/asm`. All files in `/sys/asm` are binary.

[1] BOM is a Unicode-standard signature that indicates the order of the stream of bytes that follows it.

Typical uses are listing, copying, moving, creating, and deleting ASM files and folders. Example 25–1 is an example of navigating the ASM virtual folder and listing the files in a subfolder.

Example 25–1 Navigating ASM Folders

The structure of the ASM virtual folder, `/sys/asm`, is described in Chapter 20, "Accessing Oracle XML DB Repository Data". In this example, the disk groups are `DATA` and `RECOVERY`; the database name is `MFG`; and the directories created for aliases are `dbs` and `tmp`. This example navigates to a subfolder, lists its files, and copies a file to the local file system.

```
ftp> open myhost 7777
ftp> user system
ftp> passwd dba
ftp> cd /sys/asm
ftp> ls
DATA
RECOVERY
ftp> cd DATA
ftp> ls
dbs
MFG
ftp> cd dbs
ftp> ls
t_dbl.f
t_axl.f
ftp> binary
ftp> get t_dbl.f, t_axl.f
ftp> put my_db2.f
```

In this example, after connecting to and logging onto database `myhost` (first three lines), FTP methods `cd` and `ls` are used to navigate and list folders, respectively. When in folder `/sys/asm/DATA/dbs`, FTP command `get` is used to copy files `t_db1.f` and `t_ax1.f` to the current folder of the local file system. Then, FTP command `put` is used to copy file `my_db2.f` from the local file system to folder `/sys/asm/DATA/dbs`.

DBAs can copy ASM files from *one database server to another*, as well as between the database and a local file system. Example 25–2 shows copying between two databases. For this, the `proxy` FTP client method can be used, if available. The `proxy` method provides a *direct* connection to two different remote FTP servers.

Example 25–2 Transferring ASM Files Between Databases with FTP proxy Method

This example copies an ASM file from one database to another. Terms with suffix `1` correspond to database `server1`; terms with suffix `2` correspond to database `server2`.

```
 1 ftp> open server1 port1
 2 ftp> user username1
 3 ftp> passwd password1
 4 ftp> cd /sys/asm/DATAFILE/MFG/DATAFILE
 5 ftp> proxy open server2 port2
 6 ftp> proxy user username2
 7 ftp> proxy passwd password2
 8 ftp> proxy cd /sys/asm/DATAFILE/MFG/DATAFILE
 9 ftp> proxy put dbs2.f tmp1.f
10 ftp> proxy get dbs1.f tmp2.f
```

In this example:

- Line 1 opens an FTP control connection to the Oracle XML DB FTP server, `server1`.
- Lines 2–3 log the DBA onto `server1`.
- Line 4 navigates to `/sys/asm/DATAFILE/MFG/DATAFILE` on `server1`.
- Line 5 opens an FTP control connection to the second database server, `server2`. At this point, the FTP command `proxy ?` could be issued to see the available FTP commands on the secondary connection. (This is not shown.)
- Lines 6–7 log the DBA onto `server2`.
- Line 8 navigates to `/sys/asm/DATAFILE/MFG/DATAFILE` on `server2`.
- Line 9 copies ASM file `dbs2.f` from `server2` to ASM file `tmp1.f` on `server1`.
- Line 10 copies ASM file `dbs1.f` from `server1` to ASM file `tmp2.f` on `server2`.

Using FTP on Standard or Nonstandard 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 nonstandard, unprotected 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

Oracle XML DB protocol server also provides session management for this protocol. After a short wait for a new command, FTP returns to the protocol layer and the shared server is freed up to serve other connections. The duration of this short wait is configurable by changing 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.

Handling Error 421. Modifying the Default Timeout Value of an FTP Session

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 25–3 Modifying the Default Timeout Value of an FTP Session

```
DECLARE
  newconfig XMLType;
BEGIN
  SELECT
    updateXML(
      DBMS_XDB.cfg_get(),
      '/xdbconfig/sysconfig/protocolconfig/ftpconfig/session-timeout/text()',
      123456789)
    INTO newconfig
    FROM DUAL;
  DBMS_XDB.cfg_update(newconfig);
END;
/
COMMIT;
```

FTP Client Failure in Passive Mode

Do not use FTP in *passive mode* to connect remotely to a server that has `HOSTNAME` configured in `Listener.ora` as `localhost` or `127.0.0.1`. If the `HOSTNAME` specified in server file `Listener.ora` is `localhost` or `127.0.0.1`, then the server is configured for *local use only*. If you try to connect remotely to the server using FTP in passive mode, the FTP client will fail. This is because the server passes IP address `127.0.0.1` (derived from `HOSTNAME`) to the client, which makes the client try to connect to itself, not to the server.

Using HTTP(S) and Oracle XML DB Protocol Server

Oracle XML DB implements HyperText Transfer Protocol (HTTP), HTTP 1.1 as defined in the RFC2616 specification.

Oracle XML DB Protocol Server: HTTP(S) Features

The Oracle XML DB HTTP(S) component in the Oracle XML DB protocol server implements the RFC2616 specification with the *exception* of the following optional features:

- `gzip` and compress transfer encodings
- byte-range headers
- The `TRACE` method (used for proxy error debugging)
- Cache-control directives (these require you to specify expiration dates for content, and are not generally used)
- TE, Trailer, Vary & Warning headers
- Weak entity tags
- Web common log format
- Multi-homed Web server

> **See Also:** RFC 2616: HTTP 1.1 Protocol Specification—`http://www.ietf.org/rfc/rfc2616.txt`

HTTP(S) Features That Are Not Supported

Digest Authentication (RFC 2617) is *not* supported. Oracle XML DB supports Basic Authentication, where a client sends the user name and password in clear text in the Authorization header.

HTTP(S) Client Methods That Are Supported

For access to the repository, Oracle XML DB supports the following HTTP(S) client methods.

- `OPTIONS` – get information about available communication options
- `GET` – get document/data (including headers)
- `HEAD` – get headers only, without document body
- `PUT` – store data in resource
- `DELETE` – delete resource

The semantics of these HTTP(S) methods are in accordance with WebDAV. Servlets and Web services may support additional HTTP(S) methods, such as `POST`.

See Also: "Supported WebDAV Client Methods" on page 25-17 for supported HTTP(S) client methods involving WebDAV

Using HTTP(S) on Nonstandard Ports

By default, HTTP listens on a nonstandard, unprotected port: 8080. To use HTTP(S) on the standard port, such as 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`.

HTTPS: Support for Secure HTTP

If properly configured, you can access Oracle XML DB Repository in a *secure* fashion, using HTTPS. See "Configuring Secure HTTP (HTTPS)" on page 25-6 for configuration information.

Note: If Oracle Database is installed on Microsoft Windows XP with *Service Pack 2* (SP2), then you must use HTTPS for WebDAV access to Oracle XML DB Repository, or else you must make appropriate modifications to the Windows XP Registry. For information on the latter, see
`http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2netwk.mspx#XSLTsection129121120120`

Anonymous Access to Oracle XML DB Repository using HTTP

Configuration parameter `allow-repository-anonymous-access` controls whether or not anonymous HTTP access to Oracle XML DB Repository data is allowed using an unlocked `ANONYMOUS` user account. The default value is `false`, meaning that unauthenticated access to repository data is *blocked*. To allow anonymous HTTP access to the repository, you must set this parameter to `true`, and unlock the `ANONYMOUS` user account.

Caution: There is an inherent ***security risk*** associated with allowing anonymous access to the repository.

Parameter `allow-repository-anonymous-access` does *not* control anonymous access to the repository using *servlets*. Each servlet has its own `security-role-ref` parameter value to control its access.

See Also:

- Table 25–3 on page 25-5 for information on parameter `allow-repository-anonymous-access`
- "Configuring Oracle XML DB Servlets" on page 27-3 for information on parameter `security-role-ref`

Using Java Servlets with HTTP(S)

Oracle XML DB supports Java servlets. To use a Java servlet, it must be registered with a unique name in the Oracle XML DB configuration file, along with parameters to customize its action. It should be compiled, and loaded into the database. Finally, the servlet name must be associated with a pattern, which can be an extension such as `*.jsp` or a path name such as `/a/b/c` or `/sys/*`, as described in Java servlet application program interface (API) version 2.2.

While processing an HTTP(S) request, the path name for the request is matched with the registered patterns. If there is a match, then the protocol server invokes the corresponding servlet with the appropriate initialization parameters. For Java servlets, the existing Java Virtual Machine (JVM) infrastructure is used. This starts the JVM if need be, which in turn 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 27, "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 Oracle XML DB in an HTTP(S) or WebDAV context:

```
http://urltest/xyz%E3%81%82%E3%82%A2
http://%E3%81%82%E3%82%A2
http://%E3%81%82%E3%82%A2/abc%E3%81%86%E3%83%8F.xml
```

Oracle XML DB processes the requested URL, any URLs within an `IF` header, any URLs within the `DESTINATION` header, and any URLs in the `REFERRED` header that contains multibyte data.

The `default-url-charset` configuration parameter can be used to accept requests from some clients that use other, nonconforming, forms of URL, with characters that are not ASCII . If a request with such characters fails, try setting this value to the native character set of the client environment. The character set used in such URL fields must be specified with an IANA charset name.

`default-url-charset` controls the encoding for nonconforming URLs. It is not required to be set unless a nonconforming client that does not send the `Content-Type` charset is used.

> **See Also:** RFC 2616: HTTP 1.1 Protocol Specification, `http://www.ietf.org/rfc/rfc2616.txt`

Characters That Are Not ASCII In URLs

Characters that are not ASCII that appear in URLs passed to an HTTP server should be converted to `UTF-8` and escaped in the `%HH` format, where `HH` is the hexadecimal notation of the byte value. For flexibility, the Oracle XML DB protocol server interprets the incoming URLs by testing whether it is encoded in one of the following character sets in the order presented here:

- UTF-8
- Charset parameter of the Content-Type field of the request if specified
- Character set if specified in the default-url-charset configuration parameter
- Character set of the database

Controlling Character Sets for HTTP(S)

The following sections describe how character sets are controlled for data transferred using HTTP(S).

Request Character Set The character set of the HTTP(S) request body is determined with the following algorithm:

- The Content-Type header is evaluated. If the Content-Type header specifies a charset value, the specified charset is used.
- The MIME type of the document is evaluated as follows:
  - If the MIME type is "`*/xml`", the character set is determined as follows:

    - If a BOM is present, then `UTF-16` is used.

    - If an encoding declaration is present, the specified encoding is used.

    - If neither a BOM nor an encoding declaration is present, `UTF-8` is used.
  - If the MIME type is text, `ISO8859-1` is used.
  - If the MIME type is neither "`*/xml`" nor `text`, the database character set is used.

There is a difference between HTTP(S) and SQL or FTP. For text documents, the default is ISO8859-1, as specified by the IETF.org *RFC 2616: HTTP 1.1 Protocol Specification*.

Response Character Set

The response generated by Oracle XML DB HTTP Server is in the character set specified in the `Accept-Charset` field of the request. `Accept-Charset` can have a list of character sets. Based on the q-value, Oracle XML DB chooses one that does not require conversion. This might not necessarily be the charset with the highest q-value. If Oracle XML DB cannot find one, then the conversion is based on the highest q-value.

Using WebDAV and Oracle XML DB

Web Distributed Authoring and Versioning (WebDAV) is an IETF standard protocol used to provide users with a file-system interface to Oracle XML Repository over the Internet. The most popular way of accessing a WebDAV server folder is through WebFolders on Microsoft Windows 2000 or Microsoft NT.

WebDAV is an extension to the HTTP 1.1 protocol that allows an HTTP server to act as a file server. It lets clients perform remote web content authoring through a coherent set of methods, headers, request body formats and response body formats. For example, a DAV-enabled editor can interact with an HTTP/WebDAV server as if it were a file system. WebDAV provides operations to store and retrieve resources, create and list contents of resource collections, lock resources for concurrent access in a coordinated manner, and to set and retrieve resource properties.

Oracle XML DB WebDAV Features

Oracle XML DB supports the following WebDAV features:

- Foldering, specified by RFC2518
- Access Control

WebDAV is a set of extensions to the HTTP(S) protocol that allow you to edit or manage your files on remote Web servers. WebDAV can also be used, for example, to:

- Share documents over the Internet
- Edit content over the Internet

> **See Also:** RFC 2518: WebDAV Protocol Specification, http://www.ietf.org/rfc/rfc2518.txt

WebDAV Features That Are Not Supported

Oracle XML DB supports the contents of RFC2518, with the following exceptions:

- Lock-NULL resources create 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.

Supported WebDAV Client Methods

For access to the repository, Oracle XML DB supports the following HTTP(S)/WebDAV client methods.

- PROPFIND (WebDAV-specific) – get properties for a resource
- PROPPATCH (WebDAV-specific) – set or remove resource properties
- LOCK (WebDAV-specific) – lock a resource (create or refresh a lock)
- UNLOCK (WebDAV-specific) – unlock a resource (remove a lock)
- COPY (WebDAV-specific) – copy a resource
- MOVE (WebDAV-specific) – move a resource
- MKCOL (WebDAV-specific) – create a folder resource (collection)

> **See Also:** "HTTP(S) Client Methods That Are Supported" on page 25-13 for additional supported HTTP(S) client methods

Using WebDAV with Microsoft Windows XP SP2

If Oracle Database is installed on Microsoft Windows XP with *Service Pack 2* (SP2), then you must use a secure connection (HTTPS) for WebDAV access to Oracle XML DB Repository, or else you must make appropriate modifications to the Windows XP Registry.

> **See Also:**
>
> - http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2netwk.mspx#XSLTsection129121120120 for information on making necessary modifications to the Windows XP registry
> - "Configuring Secure HTTP (HTTPS)" on page 25-6

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 25–2.

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 25–2 Creating a WebFolder in Windows 2000

26

User-Defined Repository Metadata

This chapter describes how to create and use XML metadata, which you associate with XML data and store in Oracle XML DB Repository.

This chapter contains these topics:

Overview of Metadata and XML

Data that you use is often associated with additional information that is not part of the content. You can use such **metadata** to group or classify data, in order to process it in different ways. For example, you might have a collection of digital photographs, and you might associate metadata with each picture, such as information about the photographic characteristics (color composition, focal length) or context (location, kind of subject: landscape, people).

An Oracle XML DB repository **resource** is an XML document that contains both metadata and data; the data is the contents of element `Contents`; all other elements in the resource contain metadata. The data of a resource can be XML, but it need not be.

You can associate resources in the Oracle XML DB repository with metadata that you define. In addition to such *user-defined metadata*, each repository resource also has associated metadata that Oracle XML DB creates automatically and uses (transparently) to manage the resource. Such *system-defined metadata* includes properties such as the owner and creation date of each resource.

Except for system-defined metadata, you decide which resource information should be treated as data and which should be treated as metadata. In the case of a photo resource, supplemental information about the photo is normally not considered to be part of the photo data, which is a binary image. For text, however, you sometimes have a choice of whether to include particular information in the resource contents (data) or keep it separate and associate it with the contents as metadata—that choice is often influenced by the applications that use or produce the data.

Kinds of Metadata – Uses of the Term

In addition to resource metadata (system-defined and user-defined), the term "metadata" is sometimes used to refer to the following:

- An XML *schema* is metadata that describes a class of XML documents.
- An XML *tag* (element or attribute name) is metadata that is used to label and organize the element content or attribute value.

You can associate metadata with an XML document that is the content of a repository resource in any of these ways:

- You can add additional XML elements containing the metadata information to the resource *contents*. For example, you could wrap digital image data in an XML document that also includes elements describing the photo. In this case, the data and its metadata are associated by being in the contents of the same resource. It is up to applications to separate the two and relate them correctly.
- You can add metadata information for a particular resource to the repository as the contents of a *separate resource*. In this case, it is up to applications to treat this resource as metadata and associate it with the data.
- You can add metadata information for a resource as repository *resource metadata*. In this case, Oracle XML DB recognizes the metadata as such. Applications can *discover* this metadata by querying the repository for it. They need not be informed separately of its existence and its association with the data.

See Also: "Oracle XML DB Resources" on page 20-4

User-Defined Resource Metadata

Of these different ways of considering metadata, this chapter is about only the last of those just listed: *user-defined resource metadata*. Such metadata is itself *represented as XML*: it is XML data that is associated with other XML data, describing it or providing supplementary, related information.

User-defined metadata for resources can be either XML schema-based or not:

- Resource metadata that is *schema-based* is stored in separate (out-of-line) tables. These are related to the resource table by the resource OID, which is stored in the hidden object column `RESID` of the metadata tables.
- Resource metadata that is *not* schema-based is stored in a `CLOB` column (`RESEXTRA`) in the resource table.

You can take advantage of schema-based metadata, in particular, to perform efficient queries and DML operations on resources. In this chapter, you will learn how to perform the following tasks involving schema-based resource metadata:

- Create and register an *XML schema* that defines the metadata for a particular kind of resource.
- *Add* metadata to a repository resource, and *update* (modify) such metadata.
- *Query* resource metadata to find associated content.
- *Delete* specific metadata associated with a resource and *purge* all metadata associated with a resource.

In addition, you will learn how to add non-schema-based metadata to a resource.

You can generally use user-defined resource metadata just as you would use resource data. In particular, *versioning* and *access control* management apply.

Typical uses of resource metadata include workflow applications, enforcing user rights management, tracking resource ownership, and controlling resource validity dates.

Scenario: Metadata for a Photo Collection

To illustrate the use of schema-based resource metadata, we will consider metadata associated with photographic image files that are stored in repository resources. You can create any number of different kinds of metadata to be associated with the same resource. In the case of image files, we will create metadata for information about both 1) the technical aspects of a photo and 2) the photo subject or the uses to which a photo might be put. We will use these two kinds of associated metadata to query photo resources.

XML Schemas to Define Resource Metadata

We first define the metadata that we want to associate with each photo resource. XML Schema is our tool for defining XML data: for each kind (technique, category) of metadata we define, we create and register an XML schema.

Example 26–1 Register an XML Schema for Technical Photo Information

The following XML schema defines metadata used to describe the technical aspects of a photo image file. We use procedure `DBMS_XMLSCHEMA.registerSchema` to register the XML schema. In order to identify this schema as defining repository resource *metadata*, we use the `ENABLE_HIERARCHY_RESMETADATA` value for the `enableHierarchy` parameter. Resource contents (data) are defined by using value `ENABLE_HIERARCHY_CONTENTS` (the default value), instead.

The properties we define here are the image height, width, color depth, title, and brief description.

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    'imagetechnique.xsd',
    '<xsd:schema targetNamespace="inamespace"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xdb="http://xmlns.oracle.com/xdb"
                 xmlns="inamespace">
       <xsd:element name="ImgTechMetadata"
                    xdb:defaultTable="IMGTECHMETADATATABLE">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="Height"      type="xsd:float"/>
             <xsd:element name="Width"       type="xsd:float"/>
             <xsd:element name="ColorDepth"  type="xsd:integer"/>
             <xsd:element name="Title"       type="xsd:string"/>
             <xsd:element name="Description" type="xsd:string"/>
           </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
      </xsd:schema>',
    enableHierarchy=>DBMS_XMLSCHEMA.ENABLE_HIERARCHY_RESMETADATA);
END;
/
```

Example 26–2 Register an XML Schema for Photo Categorization

The following XML schema defines metadata used to categorize a photo image file: to describe its content or possible uses. In this simple example, we define a single, general property for classification, named `Category`.

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    'imagecategories.xsd',
    '<xsd:schema targetNamespace="cnamespace"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xdb="http://xmlns.oracle.com/xdb"
                 xmlns="cnamespace">
       <xsd:element name="ImgCatMetadata"
                    xdb:defaultTable="IMGCATMETADATATABLE">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="Categories" type="CategoriesType"/>
           </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
        <xsd:complexType name="CategoriesType">
          <xsd:sequence>
            <xsd:element name="Category" type="xsd:string" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:schema>',
    enableHierarchy=>DBMS_XMLSCHEMA.ENABLE_HIERARCHY_RESMETADATA);
END;
/
```

Notice that there is nothing in the XML schema definitions of metadata that restrict that information to being associated with any particular kind of data. You are free to associate any type of metadata with any type of resource. And multiple types of metadata can be associated with the same resource.

Notice, too, that the XML schema does not, by itself, define its associated data as being metadata—it is the schema *registration* that makes this characterization, through `enableHierarchy` value `ENABLE_HIERARCHY_RESMETADATA`. If the same schema were registered instead with `enableHierarchy` value `ENABLE_HIERARCHY_CONTENTS` (the default value), then it would define not metadata for resources, but resource *contents* with the same information. Of course, the same XML schema cannot be registered more than once under the same name.

Adding, Updating, and Deleting Resource Metadata

You can add, update, and delete user-defined resource metadata in the following ways:

- use PL/SQL procedures in package `DBMS_XDB`:
  - `appendResourceMetadata` – add metadata to a resource
  - `updateResourceMetadata` – modify resource metadata
  - `deleteResourceMetadata` – delete specific metadata from a resource
  - `purgeResourceMetadata` – delete *all* metadata from a resource
- use SQL DML statements `INSERT`, `UPDATE`, and `DELETE` to update the resource directly

- use WebDAV protocol method `PROPPATCH`

Using the latter two methods, updating and deleting metadata are done in the same way as adding metadata. If you supply a complete `Resource` element for one of these operations, then keep in mind that each resource metadata property must be a child (not just a descendent) of element `Resource`—if you want multiple metadata elements of the same kind, you must collect them as children of a single parent metadata element. The order among such top-level user-defined resource metadata properties is unimportant and is not necessarily maintained by Oracle XML DB.

The separate PL/SQL procedures in package `DBMS_XDB` are similar in their use. Each can be used with either XML schema-based or non-schema-based metadata. Some forms (signatures) of some of the procedures apply only to schema-based metadata. Procedures `appendResourceMetadata` and `deleteResourceMetadata` are illustrated here with examples.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information on the procedures in PL/SQL package `DBMS_XDB`

Using APPENDRESOURCEMETADATA to Add Metadata

You can use procedure `DBMS_XDB.appendResourceMetadata` to add user-defined metadata to resources.

Example 26–3 Add Metadata to a Resource – Technical Photo Information

This example creates a photo resource and adds XML schema-based metadata of type `ImgTechMetadata` to it, recording the technical information about the photo.

```
DECLARE
  returnbool BOOLEAN;
BEGIN
  returnbool := DBMS_XDB.createResource(
                  '/public/horse_with_pig.jpg',
                  bfilename('MYDIR', 'horse_with_pig.jpg'));
  DBMS_XDB.appendResourceMetadata(
    '/public/horse_with_pig.jpg',
    XMLType('<i:ImgTechMetadata
                 xmlns:i="inamespace"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="inamespace imagetechnique.xsd">
               <Height>1024</Height>
               <Width>768</Width>
               <ColorDepth>24</ColorDepth>
               <Title>Pig Riding Horse</Title>
               <Description>Picture of a pig riding a horse on the beach,
taken outside hotel window.</Description>
             </i:ImgTechMetadata>'));
END;
/
```

Example 26–4 Add Metadata to a Resource – Photo Content Categories

This example adds metadata of type `ImgTechMetadata` to the same resource as Example 26–3, placing the photo in several user-defined content categories.

```
BEGIN
  DBMS_XDB.appendResourceMetadata(
    '/public/horse_with_pig.jpg',
    XMLType('<c:ImgCatMetadata
                 xmlns:c="cnamespace"
```

```
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="cnamespace imagecategories.xsd">
                <Categories>
                  <Category>Vacation</Category>
                  <Category>Animals</Category>
                  <Category>Humor</Category>
                  <Category>2005</Category>
                </Categories>
              </c:ImgCatMetadata>'));
END;
/

PL/SQL procedure successfully completed.

SELECT * FROM imgcatmetadatatable;

SYS_NC_ROWINFO$
--------------------------------------------------------------------------------
<c:ImgCatMetadata xmlns:c="cnamespace" xmlns:xsi="http://www.w3.org/2001/XMLSche
ma-instance" xsi:schemaLocation="cnamespace imagecategories.xsd">
  <Categories>
    <Category>Vacation</Category>
    <Category>Animals</Category>
    <Category>Humor</Category>
    <Category>2005</Category>
  </Categories>
</c:ImgCatMetadata>

1 row selected.
```

Using DELETERESOURCEMETADATA to Delete Metadata

You can use procedure `DBMS_XDB.deleteResourceMetadata` to delete specific metadata associated with a resource. To delete *all* of the metadata associated with a resource, you can use procedure `DBMS_XDB.purgeResourceMetadata`.

Example 26–5 Delete Specific Metadata from a Resource

This example deletes the category metadata that was added to the photo resource in Example 26–4. By default, both the resource link (`REF`) to the metadata and the metadata table identified by that link are deleted. An optional parameter can be used to specify that only the link is to be deleted; the metadata table is then left as is but becomes unrelated to the resource. In this example, the default behavior is used.

```
BEGIN
  DBMS_XDB.deleteResourceMetadata('/public/horse_with_pig.jpg',
                                  'cnamespace',
                                  'ImgCatMetadata');
END;
/

PL/SQL procedure successfully completed.

SELECT * FROM imgcatmetadatatable;

no rows selected
```

Using SQL DML to Add Metadata

An alternative to using procedure `DBMS_XDB.appendResourceMetadata` to add, update, or delete resource metadata is to update the `RESOURCE_VIEW` directly using DML statements `INSERT` and `UPDATE`. Adding resource metadata in this way is illustrated by Example 26–6.

Example 26–6 Add Metadata to a Resource Using DML with RESOURCE_VIEW

This example shows how to accomplish the same thing as Example 26–3 by inserting the metadata directly into `RESOURCE_VIEW` using SQL statement `UPDATE`. Other SQL DML statements may be used similarly.

```
UPDATE RESOURCE_VIEW
  SET RES =
        insertChildXML(
          RES,
          '/r:Resource',
          'c:ImgCatMetadata',
          XMLType('<c:ImgCatMetadata
                      xmlns:c="cnamespace"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      xsi:schemaLocation="cnamespace imagecategories.xsd">
                    <Categories>
                      <Category>Vacation</Category>
                      <Category>Animals</Category>
                      <Category>Humor</Category>
                      <Category>2005</Category>
                    </Categories>
                  </c:ImgCatMetadata>'),
          'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
           xmlns:c="cnamespace"')
  WHERE equals_path(RES, '/public/horse_with_pig.jpg') = 1;
/

SELECT * FROM imgcatmetadatatable;

SYS_NC_ROWINFO$
--------------------------------------------------------------------------------
<c:ImgCatMetadata xmlns:c="cnamespace" xmlns:xsi="http://www.w3.org/2001/XMLSche
ma-instance" xsi:schemaLocation="cnamespace imagecategories.xsd">
  <Categories>
    <Category>Vacation</Category>
    <Category>Animals</Category>
    <Category>Humor</Category>
    <Category>2005</Category>
  </Categories>
</c:ImgCatMetadata>

1 row selected.
```

The following query extracts the inserted metadata using `RESOURCE_VIEW`, rather than directly using metadata table `imgcatmetadatatable` (the actual result is *not* pretty-printed):

```
SELECT extract(RES,
               '/r:Resource/c:ImgCatMetadata',
               'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                xmlns:c="cnamespace"')
  FROM RESOURCE_VIEW
  WHERE equals_path(RES, '/public/horse_with_pig.jpg') = 1;
```

```
EXTRACT(RES,'/R:RESOURCE/C:IMGCATMETADATA','XMLNS:R="HTTP://XMLNS.ORACLE.COM/XDB
--------------------------------------------------------------------------------
<c:ImgCatMetadata xmlns:c="cnamespace"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="cnamespace imagecategories.xsd">
  <Categories>
    <Category>Vacation</Category>
    <Category>Animals</Category>
    <Category>Humor</Category>
    <Category>2005</Category>
  </Categories>
</c:ImgCatMetadata>

1 row selected.
```

Using WebDAV PROPPATCH to Add Metadata

Another alternative to using procedure `DBMS_XDB.appendResourceMetadata` to add resource metadata is to use the `PROPPATCH` method of the WebDAV protocol. This is illustrated by Example 26–7. Metadata updates and deletions can be made similarly.

Example 26–7 Add Metadata with WebDAV PROPPATCH

This example shows how to accomplish the same thing as Example 26–4 by inserting the metadata using the WebDAV protocol `PROPPATCH` method. Using appropriate tools, your application creates such a `PROPPATCH` WebDAV request and sends it to the WebDAV server for processing.

To update user-defined metadata, you proceed in the same way. To *delete* user-defined metadata, the WebDAV request is similar, but it has `D:remove` in place of `D:set`.

```
PROPPATCH /public/horse_with_pig.jpg HTTP/1.1
Host: www.myhost.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 609
Authorization: Basic dGRhZHhkYl9tZXRhOnRkYWR4ZGJfbWV0YQ==
Connection: close

<?xml version="1.0" encoding="utf-8" ?>
<D:propertyupdate xmlns:D="DAV:" xmlns:Z="http://www.w3.com/standards/z39.50/">
  <D:set>
    <D:prop>
      <c:ImgCatMetadata
          xmlns:c="cnamespace"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="cnamespace imagecategories.xsd">
        <Categories>
          <Category>Vacation</Category>
          <Category>Animals</Category>
          <Category>Humor</Category>
          <Category>2005</Category>
        </Categories>
      </c:ImgCatMetadata>
    </D:prop>
  </D:set>
</D:propertyupdate>
```

Querying Schema-Based Resource Metadata

When you register an XML schema using the `enableHierarchy` value `ENABLE_HIERARCHY_RESMETADATA`, an additional column, `RESID`, is added automatically to the `XMLType` tables used to store the metadata. This column stores the object identifier (OID) of the resource associated with the metadata. You can use column `RESID` when querying metadata, to join the metadata with the associated data.

You can query metadata in these ways:

- Query `RESOURCE_VIEW` for the metadata. For example:

```
SELECT COUNT(*) FROM RESOURCE_VIEW
  WHERE
    existsNode(RES,
               '/r:Resource/c:ImgCatMetadata/Categories/Category
                [text()="Vacation"]',
               'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                xmlns:c="cnamespace"') = 1;

  COUNT(*)
----------
         1

1 row selected.
```

- Query the XML schema-based table for the user-defined metadata directly, and join this metadata back to the resource table, identifying which resource to select. To do this, we use the `RESID` column of the metadata table. For example:

```
SELECT COUNT(*) FROM RESOURCE_VIEW rs, imgcatmetadatatable ct
  WHERE existsNode(RES,
                   '/r:Resource/c:ImgCatMetadata/Categories/Category
                    [text()="Vacation"]',
                   'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                    xmlns:c="cnamespace"') = 1
    AND rs.RESID = ct.RESID;

  COUNT(*)
----------
         1

1 row selected.
```

The latter method is recommended, for performance reasons. Direct queries of the `RESOURCE_VIEW` alone *cannot be optimized* using XPath rewrite, since there is no way to determine whether or not target elements like `Category` are stored in the `CLOB` value or in an out-of-line table.

To improve performance further, create an index on each metadata column you intend to query.

Example 26–8 Query XML Schema-Based Resource Metadata

This example queries both kinds of photo resource metadata, retrieving the paths to the resources that are categorized as vacation photos and have title "Pig Riding Horse".

```
SELECT ANY_PATH
  FROM RESOURCE_VIEW rs, imgcatmetadatatable ct, imgtechmetadatatable tt
  WHERE existsNode(RES,
                   '/r:Resource/c:ImgCatMetadata/Categories/Category
                    [text()="Vacation"]',
```

```
                  'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                   xmlns:c="cnamespace"') = 1
    AND existsNode(RES,
                  '/r:Resource/i:ImgTechMetadata/Title
                   [text()="Pig Riding Horse"]',
                  'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                   xmlns:i="inamespace"') = 1
    AND rs.RESID = ct.RESID
    AND rs.RESID = tt.RESID;

ANY_PATH
-------------------------
/public/horse_with_pig.jpg

1 row selected.
```

XML Image Metadata from Binary Image Metadata

In previous sections of this chapter we have used a simple user-defined XML schema that defines technical image metadata, `imagetechnique.xsd`, to illustrate ways of adding and changing repository resource metadata. That simple XML schema is not intended to be realistic with respect to technical photographic image information.

However, nearly all digital cameras now include image metadata as part of the binary image files they produce, and Oracle *inter*Media, which is part of Oracle Database, provides tools for extracting and converting this binary metadata to XML. Oracle *inter*Media XML schemas are automatically registered with Oracle XML DB Repository to convert binary image metadata of the followings kinds to XML data:

- EXIF – Exchangeable Image File Format
- IPTC-NAA IIM – International Press Telecommunications Council-Newspaper Association of America Information Interchange Model
- XMP – Extensible Metadata Platform

EXIF is the metadata standard for digital still cameras; EXIF metadata is stored in TIFF and JPEG image files. IPTC and XMP metadata is commonly embedded in image files by desktop image-processing software.

See Also:

- *Oracle interMedia User's Guide* for information on working with digital image metadata, including examples of extracting binary image metadata and converting it to XML
- *Oracle interMedia Reference* for information on the XML schemas supported by Oracle *inter*Media for use with image metadata

Adding Non-Schema-Based Resource Metadata

You store user-defined resource metadata that is *not* schema-based as a `CLOB` instance in the `ResExtra` element of the associated resource. The default XML schema for a resource has a top-level element **`any`** (declared with `maxOccurs= "unbounded"`) that allows any valid XML data as part of the resource document; this metadata is stored in `CLOB` column `RESEXTRA` of the resource table.

The following skeleton shows the structure and position of non-schema-based resource metadata:

```
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
```

```
      <Owner>DESELBY</Owner>
      ... <!-- other system-defined metadata -->
      <!-- contents of the resource>
      <Contents>
        ...
      </Contents>
      <!-- User-defined metadata (appearing within different namespace) -->
      <ResExtra>
        <MyOwnMetadata xmlns="http://www.example.com/custommetadata">
          <MyElement1>value1</MyElement1>
          <MyElement2>value2</MyElement2>
        </MyOwnMetadata>
      <ResExtra>
    </Resource>
```

You can set and access non-schema-based resource metadata belonging to namespaces other than `XDBResource.xsd` by using any of the methods described previously for schema-based resource metadata. Example 26–9 illustrates this for the case of SQL DML operations, adding user-defined metadata directly to the `<RESOURCE>` *document*.

Example 26–9 Add Non-Schema-Based Metadata to a Resource

This example shows how to add non-schema-based metadata to a resource using SQL DML.

```
DECLARE
  res BOOLEAN;
BEGIN
  res := DBMS_XDB.createResource('/public/NurseryRhyme.txt',
                                 bfilename('MYDIR',
                                           'tdadxdb-xdb_repos_meta-011.txt'),
                                 nls_charset_id('AL32UTF8'));
UPDATE RESOURCE_VIEW
  SET RES = insertChildXML(RES,
                           '/r:Resource/r:ResExtra',
                           'n:NurseryMetadata',
                           XMLType('<n:NurseryMetadata xmlns:n="nurserynamespace">
                                      <Author>Mother Goose</Author>
                                    <n:NurseryMetadata>'),
                           'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                            xmlns:n="nurserynamespace"')
  WHERE equals_path(RES, '/public/NurseryRhyme.txt') = 1;
END;
/

PL/SQL procedure successfully completed.


SELECT rs.RES.getClobVal() FROM RESOURCE_VIEW rs
  WHERE equals_path(RES, '/public/NurseryRhyme.txt') = 1;

RS.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>2005-05-24T13:51:48.043234</CreationDate>
  <ModificationDate>2005-05-24T13:51:48.290144</ModificationDate>
  <DisplayName>NurseryRhyme.txt</DisplayName>
  <Language>en-US</Language>
```

```
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>text/plain</ContentType>
  <RefCount>1</RefCount>
  <ACL>
    <acl description="Public:All privileges to PUBLIC" xmlns="http://xmlns.oracl
e.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:sch
emaLocation="http://xmlns.oracle.com/xdb/acl.xsd                          http:
//xmlns.oracle.com/xdb/acl.xsd">
      <ace>
        <principal>PUBLIC</principal>
        <grant>true</grant>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>TDADXDB_META</Owner>
  <Creator>TDADXDB_META</Creator>
  <LastModifier>TDADXDB_META</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#text</SchemaElement>
  <Contents>
    <text>Mary had a little lamb
Its fleece was white as snow
and everywhere that Mary went
that lamb was sure to go
</text>
  </Contents>
</Resource>

1 row selected.
```

PL/SQL Procedures Affecting Resource Metadata

The following PL/SQL procedures perform resource metadata operations:

- `DBMS_XMLSCHEMA.registerSchema` – Register an XML schema. Parameter `enableHierarchy` affects resource metadata.
- `DBMS_XDBZ.enable_hierarchy` – Enable repository support for an `XMLType` table or view. Use parameter `hierarchy_type` with a value of `DBMS_XDBZ.ENABLE_HIERARCHY_RESMETADATA` to enable resource metadata. This adds column `RESID` to track the resource associated with the metadata.
- `DBMS_XDBZ.disable_hierarchy` – Disable all repository support for an `XMLType` table or view.
- `DBMS_XDBZ.is_hierarchy_enabled` – Tests, using parameter `hierarchy_type`, whether the specified type of hierarchy is currently enabled for the specified `XMLType` table or view. Value `DBMS_XDBZ.IS_ENABLED_RESMETADATA` for hierarchy_type tests enablement of resource metadata.
- `DBMS_XDB.appendResourceMetadata` – Add metadata to a resource.
- `DBMS_XDB.deleteResourceMetadata` – Delete specified metadata from a resource.
- `DBMS_XDB.purgeResourceMetadata` – Delete all user-defined metadata from a resource. For schema-based resources, optional parameter `delete_option` can be used to specify whether or not to delete the metadata information, as well as unlink it.

- `DBMS_XDB.updateResourceMetadata` – Update the metadata for a resource.

  See Also: *Oracle Database PL/SQL Packages and Types Reference* for detailed information on these PL/SQL procedures

27

Writing Oracle XML DB Applications in Java

This chapter describes how to write Oracle XML DB applications in Java. It includes design guidelines for writing Java applications including servlets, and how to configure the Oracle XML DB servlets.

This chapter contains these topics:

Overview of Oracle XML DB Java Applications

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(S) as the protocol to access Oracle Database.

Which Oracle XML DB APIs Are Available Inside and Outside the Database?

All Oracle XML DB application program interfaces (APIs) are available to applications running both in the server and outside the database, including:

- JDBC support for `XMLType`
- `XMLType` class
- Java DOM implementation

Design Guidelines: Java Inside or Outside the Database?

When choosing an architecture for writing Java Oracle XML DB applications, consider the following guidelines:

HTTP(S): Accessing Java Servlets or Directly Accessing XMLType Resources

If the downstream client wants to deal with XML in its textual representation, then using HTTP(S) to either access the Java servlets or directly access `XMLType` resources, 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(S) 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(S). They are not intended as a platform for developing an entire Internet application. In that case, the application servlet should be deployed in Oracle Application Server application server and access data in the database either using JDBC, or by using the `java.net.*` or similar APIs to get XML data through HTTP(S).

They are best used for applications that want to get into the database, manipulate the data, and write it out quickly as XML, not to format HTML pages for end-users.

Writing Oracle XML DB HTTP Servlets in Java

Oracle XML DB provides a protocol server that supports FTP, HTTP 1.1, WebDAV, and Java Servlets. 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 Oracle XML DB Repository. Many of the XML elements in this file are the same as those defined by the Java Servlet 2.2 specification portion of Java 2 Enterprise Edition (J2EE), and have the same semantics. Table 27–1 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 27–1 XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
`auth-method`	Java	no	Specifies an HTTP authentication method required for access	--
`charset`	Oracle	yes	Specifies an IANA character set name	For example: ISO8859, UTF-8
`charset-mapping`	Oracle	yes	Specifies a mapping between a filename extension and a charset	--
`context-param`	Java	no	Specifies a parameter for a web application	Not yet supported
`description`	Java	yes	A string for describing a servlet or Web application	Supported for servlets
`display-name`	Java	yes	A string to display with a servlet or web 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(S) error code	Defined by RFC 2616
`error-page`	Java	yes	Defines a URL to redirect to if an error is encountered.	Can be specified through an HTTP(S) error, an uncaught Java exception, or through an uncaught Oracle error message
`exception-type`	Java	yes	Classname of a Java exception mapped to an error page	--

Table 27–1 (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(S) URL
`login-config`	Java	No	Specifies a method for authentication	Not yet supported
`mime-mapping`	Java	Yes	Specifies a mapping between filename extension and the MIME type of the content	--
`mime-type`	Java	Yes	MIME type name for resource content	For example: text/xml or application/octet-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(S) 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 27–1 (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 enclosed in double-quotes.
`session-config`	Java	No	Configuration information for an `HTTPSession`	`HTTPSession` is not supported
`session-timeout`	Java	No	Timeout for an HTTP(S) session	`HTTPSession` is not supported
`small-icon`	Java	Yes	Small icon to associate with a servlet	--
`taglib`	Java	No	JSP tag library	JSPs currently not supported
`taglib-uri`	Java	No	URI for JSP tag library description file relative to 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 27–1 (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` request	Example: `index.html`

Note:

- The following parameters defined for the `web.xml` file by Java are usable only by J2EE-compliant Enterprise Java Bean containers, and are not required for Java Servlet Containers that do not support a full J2EE environment: *env-entry, env-entry-name, env-entry-value, env-entry-type, ejb-ref, ejb-ref-type, home, remote, ejb-link, resource-ref, res-ref-name, res-type, res-auth*
- The following elements are used to define access control for resources: *security-constraint, web-resource-collection, web-resource-name, http-method, user-data-constraint, transport-guarantee, auth-constrain*. Oracle XML DB provides this functionality through access control lists (ACLs). An ACL is a list of access control entries that determines which principals have access to a given resource or resources. A future release will support using a `web.xml` file to generate ACLs.

See Also: Chapter 28, "Administering Oracle XML DB" for more information about configuring the `/xdbconfig.xml` file

HTTP Request Processing for Oracle XML DB Servlets

Oracle XML DB handles an HTTP request using the following steps:

1. If a connection has not yet been established, then Oracle Listener hands the connection to a shared server dispatcher.
2. When a new HTTP request arrives, the dispatcher wakes up a shared server.
3. The HTTP headers are parsed into appropriate structures.
4. The shared server attempts to allocate a database session from the Oracle XML DB session pool, if available, but otherwise will create a new session.
5. A new database call is started, as well as a new database transaction.
6. If HTTP(S) has included authentication headers, then the session will be authenticated as that database user (just as if the user logged into SQL*Plus). If no authentication information is included, and the request is `GET` or `HEAD`, then Oracle XML DB attempts to authenticate the session as the `ANONYMOUS` user. If that database user account is locked, then no unauthenticated access is allowed.
7. The URL in the HTTP request is matched against the servlets in the `xdbconfig.xml` file, as specified by the Java Servlet 2.2 specification.

8. The Oracle XML DB Servlet Container is invoked in the Java VM inside Oracle. If the specified servlet has not been initialized yet, then the servlet is initialized.
9. The Servlet reads input from the `ServletInputStream`, and writes output to the `ServletOutputStream`, and returns from the `service()` method.
10. If no uncaught Oracle error occurred, then the session is put back into the session pool.

See Also: Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data"

Session Pool and Oracle 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 27–2 lists Java Servlet 2.2 methods that are not implemented. They result in runtime exceptions.

Table 27–2 Java 2.2 Methods That Are Not Implemented

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 27–1 Writing an Oracle XML DB Servlet

The servlet code looks like this:

```
/* 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>Oracle 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 SQL function `updateXML`.

> **Note:** You cannot delete file `/xdbconfig.xml`, 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 provides information on Oracle tools that you can use with Oracle XML DB. It describes tools for managing Oracle XML DB, loading XML data, and exchanging XML data.

Part VI contains the following chapters:

28

Administering Oracle XML DB

This chapter describes how to administer Oracle XML DB. It includes information on installing, upgrading, and configuring Oracle XML DB.

This chapter contains these topics:

Installing and Reinstalling Oracle XML DB

You are required to install Oracle XML DB manually under the following conditions:

- "Installing or Reinstalling Oracle XML DB From Scratch" on page 28-1
- "Upgrading an Existing Oracle XML DB Installation" on page 28-4

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 28-3.

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(S), 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(S), and WebDAV

You can store data in tables outside this tablespace and access the data through Oracle XML DB 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* Oracle XML DB Repository data *inaccessible*.

See Also: "Anonymous Access to Oracle XML DB Repository using HTTP" on page 25-14 for information on allowing unauthenticated access to the repository

Dynamic Protocol Registration of FTP and HTTP(S) Services with Local Listener

Oracle XML DB installation, includes a dynamic protocol registration that registers FTP and HTTP(S) services with the local Listener. You can perform start, stop, and query with `lsnrctl`. For example:

- start: `lsnrctl start`
- stop: `lsnrctl stop`
- query: `lsnrctl status`

Changing FTP or HTTP(S) Port Numbers To change FTP and HTTP(S) port numbers, update the tags `<ftp-port>`, `<http-port>`, and `<http2-port>` in file `/xdbconfig.xml` in the Oracle XML DB Repository.

After updating the port numbers, dynamic protocol registration automatically stops FTP/HTTP(S) service on old port numbers and starts them on new port numbers if the local Listener is up. If local Listener is not up, restart the Listener after updating the port numbers.

See Also: Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data" for information on configuring protocols

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.

Note: If the Listener is running on a port that is not standard (for example, not 1521), then, in order for the protocols to register with the correct listener, the `init.ora` file must contain a `local_listener` entry. This references a `TNSNAME` entry that points to the correct Listener. After editing the `init.ora` parameter you must regenerate the `SPFILE` entry using `CREATE SPFILE`.

Installing Oracle XML DB Manually Without DBCA

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 Oracle 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.

> **See Also:** "Anonymous Access to Oracle XML DB Repository using HTTP" on page 25-14 for information on allowing unauthenticated access to the repository

Reinstalling Oracle XML DB

> **Caution:** *All* user data stored in Oracle XML DB Repository is *lost* if you drop user XDB.

To reinstall Oracle XML DB follow these steps:

1. Remove the dispatcher by removing the Oracle 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. Shut down, then restart the database instance.
6. Execute `catqm.sql`.
7. Execute `catxdbj.sql`.
8. Install Oracle XML DB manually as described in "Installing Oracle XML DB Manually Without DBCA" on page 28-2.

Upgrading an Existing Oracle XML DB Installation

The following considerations apply to *all* upgrades to Oracle Database 10g:

- Run script `catproc.sql`, as always.
- As a post upgrade step, if you want Oracle XML DB functionality, then you must install Oracle XML DB manually as described in "Installing Oracle XML DB Manually Without DBCA" on page 28-2.
- Any upgrade to Oracle Database 10*g* Release 2 (10.2) uses the default value (`false`) for configuration parameter `allow-repository-anonymous-access`, meaning that unauthenticated access to Oracle XML DB Repository is *blocked*.

See Also:

- "Anonymous Access to Oracle XML DB Repository using HTTP" on page 25-14 for information on enabling unauthenticated access to Oracle XML DB Repository
- "Formatting of XML Dates and Timestamps" on page 16-5 for information on the different treatment of XML dates and timestamps in Oracle XML DB starting with release 10*g*R2

Upgrading Oracle XML DB from Release 9.2 to Release 10*g*

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*.

Note: In Oracle Database 10*g*, configuration of Oracle XML DB is validated against the configuration XML schema, `http://xmlns.oracle.com/xdb/xdbconfig.xsd`. Existing, invalid configuration documents *must be validated* before upgrading to release 10*g*.

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.

Upgrading an Existing LCR XML Schema

This section applies only to the upgrade of an existing Oracle9*i* release 2 (9.2) database with Oracle XML DB where the *LCR XML schema* has been loaded. It applies if you upgrade directly from 9.2 to Oracle Database 10*g* (either 10.1 or 10.2). It does *not* apply

if you have previously upgraded from 9.2 to 10.1 and are now upgrading from 10.1 to 10.2.

Use this query to check if your 9.2 database has the LCR XML schema loaded:

```
SELECT count(*) FROM XDB.XDB$SCHEMA s
  WHERE s.xmldata.schema_url =
        'http://xmlns.oracle.com/streams/schemas/lcr/streamslcr.xsd';
```

If this query returns `0`, the existing LCR XML schema is not registered, and you can *skip the rest of this section*.

As part of the upgrade process, an existing LCR XML schema is dropped (through a call to PL/SQL procedure `DBMS_XMLSCHEMA.deleteSchema`), and the new version of the LCR XML schema is registered. The call to `deleteSchema` fails if the LCR XML schema has any dependent objects. These objects could be `XMLType` XML schema-based tables, columns, or other XML schemas that reference the LCR XML schema.

If the `deleteSchema` call succeeds and the new XML schema is registered, no further action is required to upgrade the XML LCR schema. You can determine whether or not the new schema was successfully registered by running this query:

```
SELECT count(*)
  FROM XDB.XDB$ELEMENT e, XDB.XDB$SCHEMA s
  WHERE s.xmldata.schema_url =
        'http://xmlns.oracle.com/streams/schemas/lcr/streamslcr.xsd'
    AND ref(s) = e.xmldata.property.parent_schema
    AND e.xmldata.property.name = 'extra_attribute_values';
```

If this query returns `1`, then no further action is required: the database has the most current version of the LCR XML schema. You can *skip the rest of this section*.

If the query returns `0`, then the LCR XML schema upgrade failed because of objects that depend on the schema. You need to upgrade the LCR XML schema and its dependent objects using the `copyEvolve` procedure available in the Oracle Database 10*g* version of Oracle XML DB, as follows:

1. Set event `22830` to level `8` your Oracle Database 10*g* database session.
2. Call PL/SQL procedure `DBMS_XMLSCHEMA.copyEvolve`. See Chapter 8, "XML Schema Evolution" for details.

See Also: *Oracle Streams Concepts and Administration*

Using Oracle Enterprise Manager to Administer Oracle XML DB

Oracle Enterprise Manager is a graphical tool supplied with Oracle Database that lets you perform database administration tasks easily. You can use it to perform the following tasks related to Oracle XML DB:

- Configure Oracle XML DB – view or edit parameters for the Oracle XML DB configuration file, `/xdbconfig.xml`.

  For information on configuring Oracle XML DB without using Oracle Enterprise Manager, see "Configuring Oracle XML DB Using xdbconfig.xml" on page 28-6.

- Search, create, edit, and delete Oracle XML DB Repository *resources* and their associated *access-control lists* (ACLs).

For information on creating and managing resources without using Oracle Enterprise Manager, see Part V, "Oracle XML DB Repository: Foldering, Security, and Protocols".

- Search, create, edit, and delete `XMLType` *tables* and *views*.
- Search, create, register, and delete *XML schemas*.

  For information on manipulating XML schemas without using Oracle Enterprise Manager, see Chapter 5, "XML Schema Storage and Query: Basic".
- Create *function-based indexes* based on XPath expressions.

  For information on creating function-based indexes without using Oracle Enterprise Manager, see "Creating Function-Based Indexes on XMLType Tables and Columns" on page 4-34.

> **See Also:** The online help available with Oracle Enterprise Manager, for information on using Enterprise Manager to perform these tasks

Configuring Oracle XML DB Using xdbconfig.xml

Oracle XML DB is managed internally through a configuration file, `/xdbconfig.xml`, which is stored as a resource in Oracle XML DB Repository. As an alternative to using Oracle Enterprise Manager to configure Oracle XML DB, you can configure it directly using the Oracle XML DB configuration file.

The configuration file can be modified at runtime. Simply updating the configuration file creates a new version of this repository resource. At the start of each session, the current version of the configuration file is bound to that session. The session uses this configuration-file version for its duration, unless you make an explicit call to refresh the session to the latest version.

Oracle XML DB Configuration File, xdbconfig.xml

The configuration of Oracle XML DB is defined and stored in an Oracle XML DB Repository resource, `/xdbconfig.xml`, which conforms to the Oracle XML DB configuration XML schema: `http://xmlns.oracle.com/xdb/xdbconfig.xsd`. To configure or reconfigure Oracle XML DB, update file `/xdbconfig.xml`. Its structure is described in the following sections.

> **See Also:** "xdbconfig.xsd: XML Schema for Configuring Oracle XML DB" on page D-5 for a complete listing of the Oracle XML DB configuration XML schema

<xdbconfig> (Top-Level Element)

Element `<xdbconfig>` is the top-level element. Its structure is as follows:

```
<xdbconfig>
    <sysconfig> ... </sysconfig>
    <userconfig> ... </userconfig>
</xdbconfig>
```

Element `<sysconfig>` defines system-specific, built-in parameters. Element `<userconfig>` allows users to store new custom parameters.

<sysconfig> (Child of <xdbconfig>)

Element `<sysconfig>` is a child of `<xdbconfig>`. Its structure is as follows:

```
<sysconfig>
    general parameters
    <protocolconfig> ... </protocolconfig>
</sysconfig>
```

Element <sysconfig> includes as content several general parameters that apply to all of Oracle XML DB, such as the maximum age of an access control list (ACL) and whether or not Oracle XML DB is case sensitive. Child <protocolconfig> contains protocol-specific parameters.

<userconfig> (Child of <xdbconfig>)

Element <userconfig> is a child of <xdbconfig>. It contains any parameters that you may want to add.

<protocolconfig> (Child of <sysconfig>)

Element <protocolconfig> is a child of <sysconfig>. Its structure is as follows:

```
<protocolconfig>
  <common> ... </common>
  <ftpconfig> ... </ftpconfig>
  <httpconfig> ... </httpconfig>
</protocolconfig>
```

Under <common>, Oracle Database stores parameters that apply to all protocols, such as MIME-type information. Parameters that are specific to protocols FTP and HTTP(S) are in elements <ftpconfig> and <httpconfig>, respectively.

> **See Also:** Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data", Table 25–1, Table 25–2, and Table 25–3, for a list of protocol configuration parameters

<httpconfig> (Child of <protocolconfig>)

Element <httpconfig> is a child of <protocolconfig>. Its structure is as follows:

```
<httpconfig>
  ...
  <webappconfig>
    ...
    <servletconfig>
      ...
      <servlet-list>
        <servlet> ... </servlet>
        ...
      </servlet-list>
    </servletconfig>
  </webappconfig>
  ...
  <plsql> ... </plsql>
</httpconfig>
```

Element <httpconfig> has the following child elements, in addition to others:

- <webappconfig> – used to configure Web-based applications. This includes Web application-specific parameters, such as icon name, display name for the application, and a list of servlets.

Element <servletconfig> is a child of <webappconfig> that is used to define servlets. It has child <servlet-list>, which has child <servlet> (see "<servlet> (Descendent of <httpconfig>)" on page 28-8).

- <plsql> – used in connection with Oracle HTML DB. This element is managed by PL/SQL package DBMS_EPG and must *not* be changed by hand.

See Also:

- Chapter 25, "FTP, HTTP(S), and WebDAV Access to Repository Data", Table 25–1, Table 25–2, and Table 25–3, for a list of protocol configuration parameters
- *Oracle HTML DB User's Guide*, for information on Oracle HTML DB
- *Oracle Database PL/SQL Packages and Types Reference*, for information on package DBMS_EPG

<servlet> (Descendent of <httpconfig>)

Element <servlet> is a descendent of <httpconfig> – see "<httpconfig> (Child of <protocolconfig>)" on page 28-7. It is used to configure servlets, including Java servlets.

An optional <plsql> element, child of <servlet>, is used in connection with Oracle HTML DB. It is managed by PL/SQL package DBMS_EPG and must *not* be changed by hand.

See Also:

- Chapter 27, "Writing Oracle XML DB Applications in Java" for information on configuring Java servlets
- *Oracle HTML DB User's Guide*, for information on Oracle HTML DB
- *Oracle Database PL/SQL Packages and Types Reference*, for information on package DBMS_EPG

Oracle XML DB Configuration File Example

The following is a sample Oracle XML DB configuration file:

Example 28–1 Oracle XML DB Configuration File

```
<xdbconfig xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://xmlns.oracle.com/xdb/xdbconfig.xsd
                                     http://xmlns.oracle.com/xdb/xdbconfig.xsd">
   <sysconfig>
      <acl-max-age>900</acl-max-age>
      <acl-cache-size>32</acl-cache-size>
      <invalid-pathname-chars>,</invalid-pathname-chars>
      <case-sensitive>true</case-sensitive>
      <call-timeout>300</call-timeout>
      <max-link-queue>65536</max-link-queue>
      <max-session-use>100</max-session-use>
      <persistent-sessions>false</persistent-sessions>
      <default-lock-timeout>3600</default-lock-timeout>
      <xdbcore-logfile-path>/sys/log/xdblog.xml</xdbcore-logfile-path>
```

```
<xdbcore-log-level>0</xdbcore-log-level>
<resource-view-cache-size>1048576</resource-view-cache-size>

<protocolconfig>
    <common>
       <extension-mappings>
          <mime-mappings>
            <mime-mapping>
              <extension>au</extension>
              <mime-type>audio/basic</mime-type>
            </mime-mapping>
            <mime-mapping>
              <extension>avi</extension>
              <mime-type>video/x-msvideo</mime-type>
            </mime-mapping>
            <mime-mapping>
              <extension>bin</extension>
              <mime-type>application/octet-stream</mime-type>
            </mime-mapping>
          </mime-mappings>

          <lang-mappings>
            <lang-mapping>
              <extension>en</extension>
              <lang>english</lang>
            </lang-mapping>
          </lang-mappings>

          <charset-mappings>
          </charset-mappings>

          <encoding-mappings>
            <encoding-mapping>
              <extension>gzip</extension>
              <encoding>zip file</encoding>
            </encoding-mapping>
            <encoding-mapping>
              <extension>tar</extension>
              <encoding>tar file</encoding>
            </encoding-mapping>
          </encoding-mappings>
       </extension-mappings>

       <session-pool-size>50</session-pool-size>
       <session-timeout>6000</session-timeout>
    </common>

    <ftpconfig>
      <ftp-port>2100</ftp-port>
      <ftp-listener>local_listener</ftp-listener>
      <ftp-protocol>tcp</ftp-protocol>
      <logfile-path>/sys/log/ftplog.xml</logfile-path>
      <log-level>0</log-level>
      <session-timeout>6000</session-timeout>
      <buffer-size>8192</buffer-size>
    </ftpconfig>

    <httpconfig>
      <http-port>8080</http-port>
      <http-listener>local_listener</http-listener>
```

```
            <http-protocol>tcp</http-protocol>
            <max-http-headers>64</max-http-headers>
            <session-timeout>6000</session-timeout>
            <server-name>XDB HTTP Server</server-name>
            <max-header-size>16384</max-header-size>
            <max-request-body>2000000000</max-request-body>
            <logfile-path>/sys/log/httplog.xml</logfile-path>
            <log-level>0</log-level>
            <servlet-realm>Basic realm="XDB"</servlet-realm>
            <webappconfig>
              <welcome-file-list>
                <welcome-file>index.html</welcome-file>
                <welcome-file>index.htm</welcome-file>
              </welcome-file-list>
              <error-pages>
              </error-pages>
              <servletconfig>
                <servlet-mappings>
                  <servlet-mapping>
                    <servlet-pattern>/oradb/*</servlet-pattern>
                    <servlet-name>DBURIServlet</servlet-name>
                  </servlet-mapping>
                </servlet-mappings>

                <servlet-list>
                  <servlet>
                    <servlet-name>DBURIServlet</servlet-name>
                    <display-name>DBURI</display-name>
                    <servlet-language>C</servlet-language>
                    <description>Servlet for accessing DBURIs</description>
                    <security-role-ref>
                      <role-name>authenticatedUser</role-name>
                      <role-link>authenticatedUser</role-link>
                    </security-role-ref>
                  </servlet>
                </servlet-list>
              </servletconfig>
            </webappconfig>
          </httpconfig>
        </protocolconfig>
      <xdbcore-xobmem-bound>1024</xdbcore-xobmem-bound>
      <xdbcore-loadableunit-size>16</xdbcore-loadableunit-size>
    </sysconfig>

</xdbconfig>
```

Oracle XML DB Configuration API

You can access the Oracle XML DB configuration file, `xdbconfig.xml`, the same way you access any other XML schema-based resource in the hierarchy. It can be accessed using FTP, HTTP(S), WebDAV, Oracle Enterprise Manager, or any of the resource and Document Object Model (DOM) APIs for Java, PL/SQL, or C (OCI).

For convenience, there is a PL/SQL API provided as part of the `DBMS_XDB` package for configuration access. It exposes the following functions:

- `cfg_get` – Returns the configuration information for the current session.
- `cfg_refresh` – Refreshes the session configuration information using the current configuration file. Typical uses of `cfg_refresh` include the following:

  - You have modified the configuration and now want the session to pick up the latest version of the configuration information.
  - It has been a long running session, the configuration has been modified by a concurrent session, and you want the current session to pick up the latest version of the configuration information.

- `cfg_update` – Updates the configuration information, writing the configuration file. A `COMMIT` is performed.

Example 28–2 Updating the Configuration File Using cfg_update() and cfg_get()

This example updates parameters `ftp-port` and `http-port` in the configuration file.

```
DECLARE
  v_cfg XMLType;
BEGIN
  SELECT updateXML(DBMS_XDB.cfg_get(),
                   '/xdbconfig/descendant::ftp-port/text()',
                   '2121',
                   '/xdbconfig/descendant::http-port/text()',
                   '19090')
    INTO v_cfg FROM DUAL;
  DBMS_XDB.cfg_update(v_cfg);
  COMMIT;
END;
/
```

If you have many parameters to update, then it can be easier to use FTP, HTTP(S), or Oracle Enterprise Manager to update the configuration, rather than `cfg_update`.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Configuring Default Namespace to Schema Location Mappings

Oracle XML DB identifies schema-based `XMLType` instances by pre-parsing the input XML document. If the appropriate `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attribute is found, 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.

Oracle XML DB provides a mechanism to configure default schema location mappings. If the appropriate `xsi:` attribute is not specified in the XML document, the default schema location mappings will be used. Element `schemaLocation-mappings` of the Oracle XML DB configuration XML schema, `xdbconfig.xsd`, can be used to specify the mapping between *(namespace, element)* pairs and the default schema location. If the *element* value is empty, the mapping applies to all global elements in the specified namespace. If the `namespace` value is empty, it corresponds to the null namespace.

The definition of the `schemaLocation-mappings` element is as follows:

```
<element name="schemaLocation-mappings"
         type="xdbc:schemaLocation-mapping-type" minOccurs="0"/>

<complexType name="schemaLocation-mapping-type"><sequence>
     <element name="schemaLocation-mapping"
              minOccurs="0" maxOccurs="unbounded">
     <complexType><sequence>
       <element name="namespace" type="string"/>
```

```
        <element name="element" type="string"/>
        <element name="schemaURL" type="string"/>
      </sequence></complexType>
      </element></sequence>
 </complexType>
```

The schema location used depends on mappings in the Oracle XML DB configuration file for the namespace used and the root document element. For example, assume that the document does not have the appropriate `xsi:` attribute to indicate the schema location. Consider a document root element *R* in namespace *N*. The algorithm for identifying the default schema location is as follows:

1. If the Oracle XML DB configuration file has a mapping for *N* and *R*, the corresponding schema location is used.
2. If the configuration file has a mapping for *N*, but not *R*, the schema location for *N* is used.
3. If the document root *R* does not have any namespace, the schema location for *R* is used.

For example, suppose that your Oracle XML DB configuration file includes the following mapping:

```
<schemaLocation-mappings>
   <schemaLocation-mapping>
      <namespace>http://www.oracle.com/example</namespace>
      <element>root</element>
      <schemaURL>http://www.oracle.com/example/sch.xsd</schemaURL>
   </schemaLocation-mapping>
   <schemaLocation-mapping>
      <namespace>http://www.oracle.com/example2</namespace>
      <element></element>
      <schemaURL>http://www.oracle.com/example2/sch.xsd</schemaURL>
   </schemaLocation-mapping>
   <schemaLocation-mapping>
     <namespace></namespace>
     <element>specialRoot</element>
     <schemaURL>http://www.oracle.com/example3/sch.xsd</schemaURL>
   </schemaLocation-mapping>
</schemaLocation-mappings>
```

The following schema locations are used:

- Namespace = `http://www.oracle.com/example`
  Root Element = `root`
  Schema URL = `http://www.oracle.com/example/sch.xsd`

  This mapping is used when the instance document specifies:

  ```
  <root xmlns="http://www.oracle.com/example">
  ```

- Namespace = `http://www.oracle.com/example2`
  Root Element = `null` (any global element in the namespace)
  Schema URL = `http://www.oracle.com/example2/sch.xsd`

  This mapping is used when the instance document specifies:

  ```
  <root xmlns="http://www.oracle.example2">
  ```

- Namespace = `null` (i.e null namespace)
  Root Element = `specialRoot`
  Schema URL = `http://www.oracle.com/example3/sch.xsd`

This mapping is used when the instance document specifies:

```
<specialRoot>
```

> **Note:** This functionality is available only on the server side, that is, when XML is parsed on the server. If XML is parsed on the client side, the appropriate `xsi:` attribute is still required.

Configuring XML File Extensions

Oracle XML DB Repository treats certain files as XML documents, based on their file extensions. When such files are inserted into the repository, Oracle XML DB pre-parses them to identify the schema location (or uses the default mapping if present) and inserts the document into the appropriate default table.

By default, the following extensions are considered as XML file extensions: **`xml`**, **`xsd`**, **`xsl`**, **`xlt`**. In addition, Oracle XML DB provides a mechanism for applications to specify other file extensions as XML file extensions. The **`xml-extensions`** element is defined in the configuration schema, `http://xmlns.oracle.com/xdb/xdbconfig.xsd`, as follows:

```
<element name="xml-extensions"
         type="xdbc:xml-extension-type" minOccurs="0"/>


<complexType name="xml-extension-type"><sequence>
     <element name="extension" type="xdbc:exttype"
              minOccurs="0" maxOccurs="unbounded">
     </element></sequence>
 </complexType>
```

For example, the following fragment from the Oracle XML DB configuration file, `xdbconfig.xml`, specifies that files with extensions `vsd`, `vml`, and `svgl` should be treated as XML files:

```
<xml-extensions>
  <extension>vsd</extension>
  <extension>vml</extension>
  <extension>svgl</extension>
</xml-extensions>
```

29

Loading XML Data Using SQL*Loader

This chapter describes how to load XML data into Oracle XML DB with a focus on SQL*Loader.

This chapter contains these topics:

- Overview of Loading XMLType Data into Oracle Database
- Using SQL*Loader to Load XMLType Data
- Loading Very Large XML Documents into Oracle Database

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

Overview of 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. 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: SQL*Loader does not support direct-path loading if the data involves inheritance.

See Also: *Chapter 30, "Importing and Exporting XMLType Tables"* and *Oracle Database Utilities*

Oracle XML DB Repository information is *not* exported when user data is exported. This means that neither the resources nor any information are 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 a LOBFILE value 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 LOB files 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 XML Columns Containing LOB Data from LOBFILEs

LOB data can be lengthy enough so that it makes sense to load it from a LOBFILE instead of from a primary datafile. In LOBFILEs, LOB data instances are still considered to be in fields (predetermined size, delimited, length-value), but these fields are not organized into records (the concept of a record does not exist within LOBFILEs). Therefore, the processing overhead of dealing with records is avoided. This type of organization of data is ideal for LOB loading.

There is no requirement that a LOB from a LOBFILE fit in memory. SQL*Loader reads LOBFILEs in 64 KB chunks.

In LOBFILEs the data can be in any of the following types of fields, any of which can be used to load XML columns:

- A single LOB field into which the entire contents of a file can be read
- Predetermined size fields (fixed-length fields)
- Delimited fields (that is, `TERMINATED BY` or `ENCLOSED BY`)

  The clause `PRESERVE BLANKS` is not applicable to fields read from a LOBFILE.
- Length-value pair fields (variable-length fields).

  To load data from this type of field, use the `VARRAY`, `VARCHAR`, or `VARCHAR2` SQL*Loader datatypes.

Specifying LOBFILEs

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 Using SQL*Loader" on page 3-9

Example 29–1 illustrates how to load `XMLType` data into Oracle Database.

Example 29–1 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`. The code registers the XML schema, `person.xsd`, in Oracle XML DB, and then creates table `foo`. You can alternatively 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";
```

Here is the content of the control file, `load_data.ctl`, 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 `load_data.ctl`, the data is present in the control file itself, and a record spanned only one line (it is split over several lines here, for printing purposes).

In the following example, the data is present in a separate file, `person.dat`, from the control file, `lod2.ctl`. File `person.dat` contains more than one row, and each row spans more than one line. Here is the control file, `lod2.ctl`:

```
LOAD DATA
INFILE *
INTO TABLE foo TRUNCATE
XMLType(xmldata)
FIELDS(fill filler CHAR(1),
       xmldata LOBFILE (CONSTANT person.dat)
       TERMINATED BY '<!-- end of record -->')
BEGINDATA
0
0
0
```

The three zeroes (`0`) after `BEGINDATA` indicate that three records are present in the data file, `person.dat`. Each record is terminated by `<!-- end of record -->`. 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>
<!-- end of record -->
<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>
<!-- end of record -->
<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>
<!-- end of record -->
```

Here is the SQL*Loader command for loading the XML data into Oracle Database:

```
sqlldr [username]/[password] lod2.ctl (optional: direct=y)
```

30

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 import and export of `XMLType` tables and columns that store XML data and are based on a registered XML schema. You can import and export tables, whether or not they are based on XML schemas.

Non-Schema-Based XMLType Tables and Columns

Data from `XMLType` tables and columns that are not associated with an XML schema can be imported and exported in a manner similar to LOB columns. The export dump file stores the XML text.

XML Schema-Based XMLType Tables

Oracle Database supports the import and export 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 a call to PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema`.
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 row-level security (RLS) policies and path-index triggers are not exported for hierarchy-enabled tables: when these tables are imported, they are *not* hierarchy-enabled.
- Hidden columns `ACLOID` and `OWNERID` are *not* exported for these tables. In an imported database the values of these columns could be different, so they should be re-initialized.

Using Transportable Tablespaces with Oracle XML DB

You can move XML schema-based or non-schema-based data from one database to another using the *transportable tablespace* feature. This is generally very fast, because it involves only copying the tablespace and recreating the tablespace metadata.

When *exporting*:

- Hierachy information is lost (see "Guidelines for Exporting Hierarchy-Enabled Tables").
- Ensure that the set of tablespaces to be exported is *read-only* and *self-contained*. You can check that it is self-contained by using `DBMS_TTS.transport_set_check`.
- After exporting, run the RMAN tool to take care of any endian issues.
- Use `ignore=y` when importing, if you do *not* want an error to be raised when an XML schema referenced by the imported data already exists and is different.

When *importing*, any XML schemas referenced by the data to be imported are also imported. If such a schema already exists in the target database, then an *error* is signaled if any of the following is true:

- `ignore=n` is specified in the import utility
- `ignore=y` is specified in the import utility and the imported schema is not identical to the existing schema in the target database.

Resources and Foldering Do Not Fully Support IMPORT/EXPORT

Oracle XML DB supports a foldering mechanism that uses path names and URIs to refer to data (repository resources), rather than table names, column names, and so on. This foldering mechanism is not entirely supported using IMPORT/EXPORT.

However, for resources based on a registered XML schema, the `XMLType` tables storing the data can be imported and exported. During export, only the XML data is exported; the relationship in the Oracle XML DB foldering hierarchy is lost.

Repository Metadata is Not Exported During a Full Database Export

Oracle XML DB stores the metadata (and data unrelated to XML Schema) for Oracle XML DB Repository in the `XDB` database-user schema. Because Oracle Database does not support the export of the repository structure, these metadata tables and structures are not exported during a full database export.

The entire `XDB` user schema is skipped during a full database export and any database objects owned by `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.

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 30–1 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 30–2 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 30–3 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 30–4 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 30–5 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 global 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*.

31

Exchanging XML Data with Oracle Streams AQ

Oracle Streams Advanced Queuing (AQ) provides database integrated message queuing functionality:

- It enables and manages asynchronous communication of two or more applications using messages
- It supports point-to-point and publish/subscribe communication models

Integration of message queuing with Oracle Database brings the integrity, reliability, recoverability, scalability, performance, and security features of Oracle Database to message queuing. It also facilitates the extraction of intelligence from message flows.

This chapter describes how XML data can be exchanged using AQ. It contains these topics:

How Do AQ and XML Complement Each Other?

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(S) 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 31–1 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.

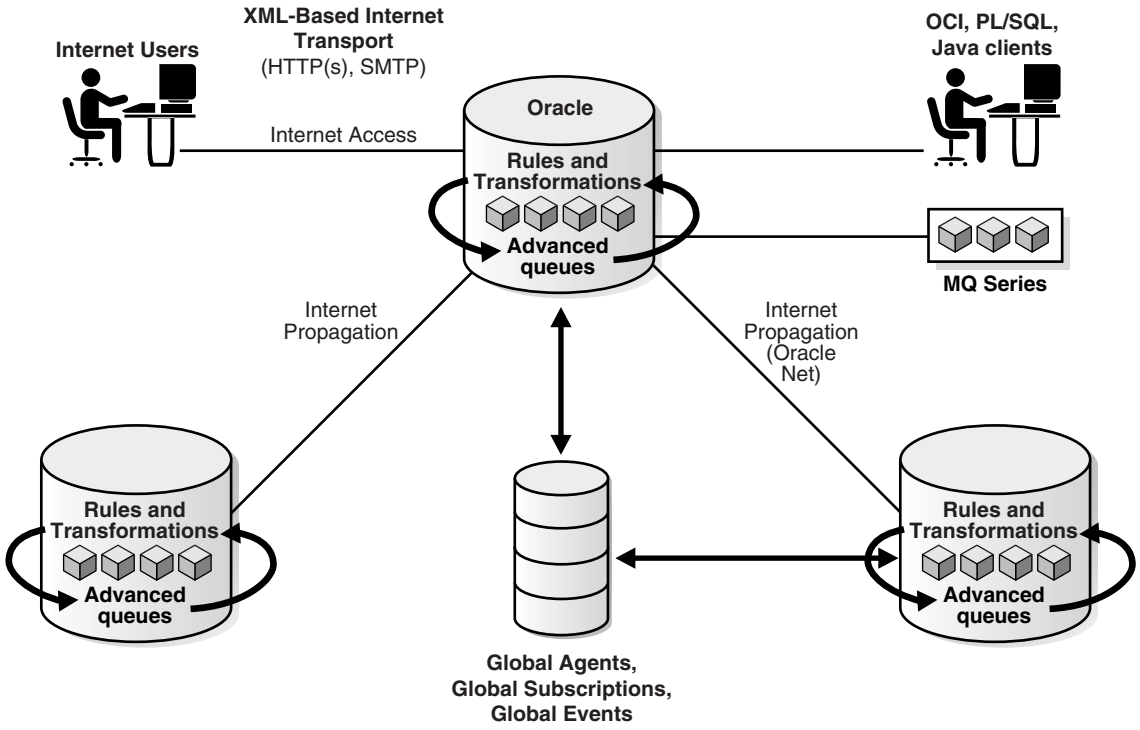
  Oracle uses this in its enterprise application integration products. XML messages are sent from applications to an Oracle AQ hub. This serves as a message server for any application that wants the message. Through this hub-and-spoke architecture, XML messages can be communicated asynchronously to multiple loosely coupled receiving applications.

Figure 31–1 shows XML payload messages transported using AQ in the following ways:

- Web-based application that uses an AQ operation over an HTTP(S) 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(S) or SMTP

The figure also shows that AQ clients can access data using OCI, Java, or PL/SQL.

Figure 31–1 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 A, "XML Schema for LCRs".

XMLType Attributes in Object Types

You can create queues that use Oracle object types containing `XMLType` attributes. These queues can be used to transmit and store messages that are XML documents. Using `XMLType`, you can do the following:

- Store any type of message in a queue
- Store documents internally as `CLOB` values
- Store more than one type of payload in a queue
- Query `XMLType` columns using functions like `existsNode`
- 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(S) and SMTP.

iDAP uses the `text/xml` content type to specify the body of the SOAP request. XML provides the presentation for iDAP request and response messages as follows:

- All request and response tags are scoped in the SOAP namespace.
- AQ operations are scoped in the iDAP namespace.
- The sender includes namespaces in iDAP elements and attributes in the SOAP body.
- The receiver processes iDAP messages that have correct namespaces; for the requests with incorrect namespaces, the receiver returns an invalid request error.
- The SOAP namespace has this value:
  `http://schemas.xmlsoap.org/soap/envelope/`
- The iDAP namespace has this value:
  `http://ns.oracle.com/AQ/schemas/access`

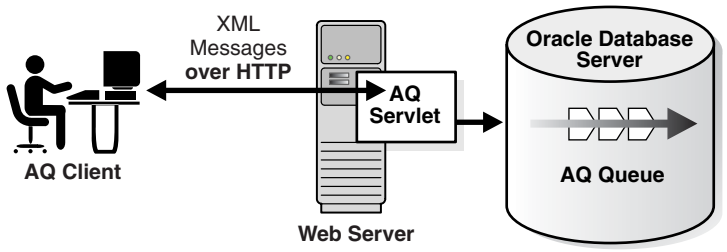
> **See Also:** *Oracle Streams Advanced Queuing User's Guide and Reference*

iDAP Architecture

Figure 31–2 shows the following components needed to send HTTP(S) messages:

- A client program that sends XML messages, conforming to iDAP format, to the AQ Servlet. This can be any HTTP client, such as Web browsers.
- The Web server or `ServletRunner` which hosts the AQ servlet that can interpret the incoming XML messages, for example, Apache/Jserv or Tomcat.
- Oracle Server/Database. Oracle Streams AQ servlet connects to Oracle Database to perform operations on your queues.

Figure 31–2 iDAP Architecture for Performing AQ Operations Using HTTP(S)



XMLType Queue Payloads

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

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

Example 31–1 XMLType and AQ: Creating a Table and Queue, and Transforming Messages

In the BooksOnline application, assume that the Overseas Shipping site represents an order using `SYS.XMLType`. The Order Entry site represents an order as an Oracle object, `ORDER_TYP`.

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 orders outside the US. */
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 31–2 XMLType and AQ: Dequeuing Messages

Assume that an application processes orders for customers in Canada. This application can dequeue messages using the following procedure:

```
/* Create procedure to enqueue into single-consumer queues: */
CREATE OR REPLACE PROCEDURE get_canada_orders() AS
  deq_msgid             RAW(16);
  dopt                  DBMS_AQ.dequeue_options_t;
  mprop                 DBMS_AQ.message_properties_t;
  deq_order_data        SYS.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 an Oracle Text `contains` expression to have the best of both of these options.

See Also: Chapter 10, "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.

See Also: *Oracle Streams Advanced Queuing User's Guide and Reference*

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.

Part VII

Appendixes

Part VII of this manual provides background material as a set of appendixes:

- Appendix A, "XML Schema Primer"
- Appendix B, "XPath and Namespace Primer"
- Appendix C, "XSLT Primer"
- Appendix D, "Oracle-Supplied XML Schemas and Examples"
- Appendix E, "Oracle XML DB Restrictions"

A

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, "Introduction to Oracle XML DB", "XML Schema Support" on page 1-13

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 uses 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`, then all elements and types defined by the 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.

If an XML schema 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 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 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.

Overview of XML Schema

Parts of this overview 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://www.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 XML 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 the Purchase-Order XML Schema

The purchase-order XML 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 XML Schema, po.xsd

The purchase-order XML schema is contained in file `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 XML 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 XML 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 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 value. 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 A–1 summarizes the attribute values used in element and attribute declarations to constrain their occurrences.

Table A–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.
(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 "Overview of the W3C Namespaces in XML Recommendation" on page B-12.

Simple Types

The purchase-order XML 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 A–2 lists all the simple types built into XML Schema, along with examples of the different types.

Table A–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
`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.

Table A–2 (Cont.) Simple Types Built into XML Schema

Simple Type	Examples (delimited by commas)	Notes
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	--
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

Table A–2 (Cont.) Simple Types Built into XML Schema

Simple Type	Examples (delimited by commas)	Notes
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 that define a canonical lexical format have been established for this type – 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 XML 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 A–3. Atomic types, list types, and the union types described in the next section are collectively called simple types. An value of atomic type is indivisible. 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 XML 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 XML 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 XML 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 XML 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 XML 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 XML 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 XML 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 `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 A–3 lists XML Schema features. Note that XML Schema features *include* DTD features.

Table A–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` Others are derived datatypes that are defined in terms of primitive types.	DTDs do not support datatypes other than character strings.
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.

Table A–3 (Cont.) XML Schema Features Compared to DTD Features

XML Schema Feature	DTD Features

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.	--
`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 A-27, artist? implied that artist is optional. ■ * = zero or more. ■ + = one or more in the "DTD Example" on page A-27, author+ implies that more than one author is possible. ■ (none) = exactly one.

Table A–3 (Cont.) XML Schema Features Compared to DTD Features

XML Schema Feature	DTD Features
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. XML schema `purchaseOrder.xsd` 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>
```

B

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:

Overview of 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 string, number, and Boolean values.

XPath uses a compact syntax that is not XML syntax to facilitate the use of XPath expressions 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 XPath values that return Boolean, number, or string values. 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 B-8. A summary of XML Namespaces is provided in "Overview of the W3C Namespaces in XML Recommendation" on page B-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
- *XML In a Nutshell*, by Elliotte Rusty Harold and W. Scott Means, O'Reilly, January 2001, http://www.oreilly.com/catalog/xmlnut/chapter/ch09.html
- http://www.w3.org/TR/2002/NOTE-unicode-xml-20020218/ for information about using Unicode in XML

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 less-than (<), then this must not appear in the XML source as a less-than character, 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 B-4 describes the semantics of location paths using the unabbreviated syntax
- "Location Path Examples Using Abbreviated Syntax" on page B-5 describes the abbreviated syntax

Location Path Examples Using Unabbreviated Syntax

Table B–1 lists examples of location paths using the unabbreviated syntax.

Table B–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 children of the context node, whatever their node type
`attribute::name`	Selects the `name` attribute of the context node
`attribute::*`	Selects all 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; 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 `para` elements in the same document as the context node
`/descendant::olist/child::item`	Selects all `item` elements that have an `olist` parent and 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

Table B–1 (Cont.) XPath: Location Path Examples Using Unabbreviated Syntax

Unabbreviated Location Path	Description
`child::para[position()=last()-1]`	Selects the penultimate `para` child of the context node
`child::para[position()>1]`	Selects all `para` children of the context node other than its first `para` child
`following-sibling::chapter[position()=1]`	Selects the next `chapter` sibling of the context node
`preceding-sibling::chapter[position()=1]`	Selects the previous `chapter` sibling of the context node
`/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 `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 B–2 lists examples of location paths using abbreviated syntax.

Table B–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 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 document element `doc`
`chapter//para`	Selects the `para` element descendants of the `chapter` element children of the context node

Table B–2 (Cont.) XPath: Location Path Examples Using Abbreviated Syntax

Abbreviated Location Path	Description
`//para`	Selects all `para` descendants of the document root, and thus selects all `para` elements in the same document as the context node
`//olist/item`	Selects all `item` elements in the same document as the context node that have an `olist` parent
.	Selects the context node
`.//para`	Selects the `para` element descendants of the context node
..	Selects the parent of the context node
`../@lang`	Selects the `lang` attribute of the parent of the context node
para[@type="warning"]	Selects all `para` children of the context node that have a `type` attribute with value `warning`
`para[@type="warning"][5]`	Selects the fifth `para` child of the context node that has a `type` attribute with value `warning`
`para[5][@type="warning"]`	Selects the fifth `para` child of the context node, if that child has a `type` attribute with value `warning`
`chapter[title="Introduction"]`	Selects the `chapter` children of the context node that have one or more `title` children with string-value `Introduction`
`chapter[title]`	Selects the `chapter` children of the context node that have one or more `title` children
`employee[@secretary and @assistant]`	Selects all `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 an at-sign (`@`).

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 //

Two slashes (`//`) 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 a period (`.`) 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 two periods (`..`) 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: Overview of the W3C Namespaces in XML Recommendation on page B-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 someXML processors that do not validate.

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 nonempty

> **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 both have a null namespace URI or both have 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.

Overview of 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 B-16 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: Prefixes Beginning X-M-L

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'>
  <!-- namespace of 'price' element 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'>
   <!-- namespace of 'taxClass' attribute 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 an XML processor that does not validate.

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

Overview of 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 B-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 and the W3C XML Information Set

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 processor that does not validate 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).

C

XSLT Primer

This appendix describes introductory information about the W3C XSL and XSLT Recommendation.

This appendix contains these topics:

Overview of 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 9, "Transforming and Validating XMLType Data"

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://www.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 9, "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 B, "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) 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 website maintenance.

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 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>
```

D

Oracle-Supplied XML Schemas and Examples

This appendix includes the definition and structure of `RESOURCE_VIEW` and `PATH_VIEW` and the Oracle XML DB-supplied XML schemas. It also includes a full listing of the purchase-order XML schemas used in various examples, and the C example for loading XML content into Oracle XML DB.

This appendix contains these topics:

XDBResource.xsd: XML Schema for Oracle XML DB Resources

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 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 XDB properties - these should NOT be changed -->
        <element name="sysconfig">
          <complexType>
            <sequence>

              <!-- generic XDB 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"/>
                  <element name="xml-extensions"
                           type="xdbc:xml-extension-type"
                           minOccurs="0"/>
                </sequence>
              </complexType>
            </element>
            <element name="session-pool-size" type="unsignedInt"
                     default="50"/>
            <element name="session-timeout" type="unsignedInt"
                     default="6000"/>
            <element name="allow-anonymous-write" type="boolean"
                     minOccurs="0" default="false"/>
          </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"/>
        <element name="http2-port" type="unsignedShort"
                minOccurs="0"/>
        <element name="http2-protocol" type="string"
                default="tcp" minOccurs="0"/>
        <element name="plsql" minOccurs="0">
          <complexType>
```

```
                              <sequence>
                                <element name="log-level"
                                         type="unsignedInt" minOccurs="0"/>
                                <element name="max-parameters"
                                         type="unsignedInt"
                                         minOccurs="0"/>
                              </sequence>
                            </complexType>
                          </element>
                        </sequence>
                      </complexType>
                    </element>
                  </sequence>
                </complexType>
              </element>
              <element name="schemaLocation-mappings"
                       type="xdbc:schemaLocation-mapping-type"
                       minOccurs="0"/>
              <element name="xdbcore-xobmem-bound" type="unsignedInt"
                       default="1024" minOccurs="0"/>
              <element name="xdbcore-loadableunit-size" type="unsignedInt"
                       default="16" minOccurs="0"/>
            </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="plsql-servlet-config">
    <sequence>
      <element name="database-username" type="string" minOccurs="0"/>
      <element name="authentication-mode" minOccurs="0">
        <simpleType>
          <restriction base="string">
            <enumeration value="Basic"/>
            <enumeration value="SingleSingOn"/>
            <enumeration value="GlobalOwa"/>
            <enumeration value="CustomOwa"/>
            <enumeration value="PerPackageOwa"/>
          </restriction>
        </simpleType>
      </element>
      <element name="session-cookie-name" type="string" minOccurs="0"/>
      <element name="session-state-management" minOccurs="0">
        <simpleType>
          <restriction base="string">
            <enumeration value="StatelessWithResetPackageState"/>
            <enumeration value="StatelessWithFastResetPackageState"/>
            <enumeration value="StatelessWithPreservePackageState"/>
          </restriction>
        </simpleType>
      </element>
      <element name="max-requests-per-session" type="unsignedInt" minOccurs="0"/>
      <element name="default-page" type="string" minOccurs="0"/>
      <element name="document-table-name" type="string" minOccurs="0"/>
```

```
      <element name="document-path" type="string" minOccurs="0"/>
      <element name="document-procedure" type="string" minOccurs="0"/>
      <element name="upload-as-long-raw" type="string" minOccurs="0"
               maxOccurs="unbounded"/>
      <element name="path-alias" type="string" minOccurs="0"/>
      <element name="path-alias-procedure" type="string" minOccurs="0"/>
      <element name="exclusion-list" type="string" minOccurs="0"
               maxOccurs="unbounded"/>
      <element name="cgi-environment-list" type="string" minOccurs="0"
               maxOccurs="unbounded"/>
      <element name="compatibility-mode" type="unsignedInt" minOccurs="0"/>
      <element name="nls-language" type="string" minOccurs="0"/>
      <element name="fetch-buffer-size" type="unsignedInt" minOccurs="0"/>
      <element name="error-style" minOccurs="0">
        <simpleType>
          <restriction base="string">
            <enumeration value="ApacheStyle"/>
            <enumeration value="ModplsqlStyle"/>
            <enumeration value="DebugStyle"/>
          </restriction>
        </simpleType>
      </element>
      <element name="transfer-mode" minOccurs="0">
        <simpleType>
          <restriction base="string">
            <enumeration value="Char"/>
            <enumeration value="Raw"/>
          </restriction>
        </simpleType>
      </element>
      <element name="before-procedure" type="string" minOccurs="0"/>
      <element name="after-procedure" type="string" minOccurs="0"/>
      <element name="bind-bucket-lengths" type="unsignedInt" minOccurs="0"
               maxOccurs="unbounded"/>
      <element name="bind-bucket-widths" type="unsignedInt" minOccurs="0"
               maxOccurs="unbounded"/>
      <element name="always-describe-procedure" minOccurs="0">
        <simpleType>
          <restriction base="string">
            <enumeration value="On"/>
            <enumeration value="Off"/>
          </restriction>
        </simpleType>
      </element>
      <element name="info-logging" minOccurs="0">
        <simpleType>
          <restriction base="string">
            <enumeration value="InfoDebug"/>
          </restriction>
        </simpleType>
      </element>
    </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>
                <element name="plsql" type="xdbc:plsql-servlet-config"
                         minOccurs="0"/>
              </sequence>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>

<complexType name="lang-mapping-type">
  <sequence>
```

```
    <element name="lang-mapping" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="extension" type="xdbc:exttype"/>
          <element name="lang" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>

<complexType name="charset-mapping-type">
  <sequence>
    <element name="charset-mapping" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="extension" type="xdbc:exttype"/>
          <element name="charset" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>

<complexType name="encoding-mapping-type">
  <sequence>
    <element name="encoding-mapping" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="extension" type="xdbc:exttype"/>
          <element name="encoding" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>

<complexType name="mime-mapping-type">
  <sequence>
    <element name="mime-mapping" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="extension" type="xdbc:exttype"/>
          <element name="mime-type" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>

<complexType name="xml-extension-type">
  <sequence>
    <element name="extension" type="xdbc:exttype"
             minOccurs="0" maxOccurs="unbounded">
    </element>
  </sequence>
</complexType>

<complexType name="schemaLocation-mapping-type">
  <sequence>
```

```
            <element name="schemaLocation-mapping"
                     minOccurs="0" maxOccurs="unbounded">
              <complexType>
                <sequence>
                  <element name="namespace" type="string"/>
                  <element name="element" type="string"/>
                  <element name="schemaURL" type="string"/>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>

        <simpleType name="exttype">
            <restriction base="string">
              <pattern value="[^\*\./]*"/>
            </restriction>
        </simpleType>
      </schema>
```

Purchase-Order XML Schemas

This section contains the complete listings of the annotated purchase-order XML schemas used in various examples, particularly in Chapter 3. Example D–2 represents a modified version of Example D–1; the modification is used in Chapter 8 to illustrate XML schema evolution.

Example D–1 Annotated Purchase-Order XML Schema, purchaseOrder.xsd

This is the complete listing of the annotated XML schema presented in Example 3–8 on page 3-20.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0"
           xdb:storeVarrayAsTable="true">
  <xs:element name="PurchaseOrder" type="PurchaseOrderType" xdb:defaultTable="PURCHASEORDER"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Reference" type="ReferenceType" minOccurs="1" xdb:SQLName="REFERENCE"/>
      <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
      <xs:element name="Reject" type="RejectionType" minOccurs="0" xdb:SQLName="REJECTION"/>
      <xs:element name="Requestor" type="RequestorType" xdb:SQLName="REQUESTOR"/>
      <xs:element name="User" type="UserType" minOccurs="1" xdb:SQLName="USERID"/>
      <xs:element name="CostCenter" type="CostCenterType" xdb:SQLName="COST_CENTER"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                  xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                  xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
      <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
    <xs:sequence>
      <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"
                  xdb:SQLName="LINEITEM" xdb:SQLCollType="LINEITEM_V"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
    <xs:sequence>
      <xs:element name="Description" type="DescriptionType"
                  xdb:SQLName="DESCRIPTION"/>
      <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
```

```
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
               xdb:SQLType="NUMBER"/>
</xs:complexType>
<xs:complexType name="PartType" xdb:SQLType="PART_T">
  <xs:attribute name="Id" xdb:SQLName="PART_NUMBER" xdb:SQLType="VARCHAR2">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:minLength value="10"/>
        <xs:maxLength value="14"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="Quantity" type="moneyType" xdb:SQLName="QUANTITY"/>
  <xs:attribute name="UnitPrice" type="quantityType" xdb:SQLName="UNITPRICE"/>
</xs:complexType>
<xs:simpleType name="ReferenceType">
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION" xdb:SQLCollType="ACTION_V">
      <xs:complexType xdb:SQLType="actioN_t">
        <xs:sequence>
          <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
          <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_ACTIONED"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
  <xs:sequence>
    <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
    <xs:element name="address" type="AddressType" minOccurs="0" xdb:SQLName="SHIP_TO_ADDRESS"/>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
```

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

Example D–2 Revised Purchase-Order XML Schema

This is the complete listing of the revised annotated XML schema presented in Example 8–1 on page 8-4. Text that is in **bold face** is additional or significantly different from that in the schema of Example D–1.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0">
  <xs:element
    name="PurchaseOrder" type="PurchaseOrderType"
    xdb:defaultTable="PURCHASEORDER"
    xdb:columnProps=
      "CONSTRAINT purchaseorder_pkey PRIMARY KEY (XMLDATA.reference),
       CONSTRAINT valid_email_address FOREIGN KEY (XMLDATA.userid)
         REFERENCES hr.employees (EMAIL)"
    xdb:tableProps=
      "VARRAY XMLDATA.ACTIONS.ACTION STORE AS TABLE ACTION_TABLE
        ((CONSTRAINT action_pkey PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       VARRAY XMLDATA.LINEITEMS.LINEITEM STORE AS TABLE LINEITEM_TABLE
        ((constraint LINEITEM_PKEY primary key (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       lob (XMLDATA.NOTES) STORE AS (ENABLE STORAGE IN ROW STORAGE(INITIAL 4K NEXT 32K))"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
      <xs:element name="Reject" type="RejectionType" minOccurs="0" xdb:SQLName="REJECTION"/>
      <xs:element name="Requestor" type="RequestorType" xdb:SQLName="REQUESTOR"/>
      <xs:element name="User" type="UserType" xdb:SQLName="USERID"/>
      <xs:element name="CostCenter" type="CostCenterType" xdb:SQLName="COST_CENTER"/>
      <xs:element name="BillingAddress" type="AddressType" minOccurs="0"
                  xdb:SQLName="BILLING_ADDRESS"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                  xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                  xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
      <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
      <xs:element name="Notes" type="NotesType" minOccurs="0" xdb:SQLType="CLOB"
                  xdb:SQLName="NOTES"/>
    </xs:sequence>
    <xs:attribute name="Reference" type="ReferenceType" use="required" xdb:SQLName="REFERENCE"/>
    <xs:attribute name="DateCreated" type="xs:dateTime" use="required"
                  xdb:SQLType="TIMESTAMP WITH TIME ZONE"/>
  </xs:complexType>
  <xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
    <xs:sequence>
      <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded" xdb:SQLName="LINEITEM"
                  xdb:SQLCollType="LINEITEM_V"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
    <xs:sequence>
      <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
      <xs:element name="Quantity" type="quantityType"/>
    </xs:sequence>
    <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                  xdb:SQLType="NUMBER"/>
  </xs:complexType>
  <xs:complexType name="PartType" xdb:SQLType="PART_T">
    <xs:simpleContent>
      <xs:extension base="UPCCodeType">
        <xs:attribute name="Description" type="DescriptionType" use="required"
                      xdb:SQLName="DESCRIPTION"/>
        <xs:attribute name="UnitCost" type="moneyType" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:simpleType name="ReferenceType">
```

```
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION" xdb:SQLCollType="ACTION_V">
      <xs:complexType xdb:SQLType="ACTION_T">
        <xs:sequence>
          <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
          <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_ACTIONED"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
  <xs:sequence>
    <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
    <xs:choice>
      <xs:element name="address" type="AddressType" minOccurs="0"/>
      <xs:element name="fullAddress" type="FullAddressType" minOccurs="0"
                  xdb:SQLName="SHIP_TO_ADDRESS"/>
    </xs:choice>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostCenterType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="4"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="VendorType">
```

```
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PurchaseOrderNumberType">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
<xs:simpleType name="SpecialInstructionsType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="2048"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NameType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="FullAddressType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="256"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="TelephoneType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="24"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DateType">
        <xs:restriction base="xs:date"/>
</xs:simpleType>
<xs:simpleType name="CommentsType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="2048"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DescriptionType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="256"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="AddressType" xdb:SQLType="ADDRESS_T">
  <xs:sequence>
    <xs:element name="StreetLine1" type="StreetType"/>
    <xs:element name="StreetLine2" type="StreetType" minOccurs="0"/>
    <xs:element name="City" type="CityType"/>
    <xs:choice>
      <xs:sequence>
        <xs:element name="State" type="StateType"/>
        <xs:element name="ZipCode" type="ZipCodeType"/>
      </xs:sequence>
      <xs:sequence>
        <xs:element name="Province" type="ProvinceType"/>
        <xs:element name="PostCode" type="PostCodeType"/>
      </xs:sequence>
      <xs:sequence>
        <xs:element name="County" type="CountyType"/>
        <xs:element name="Postcode" type="PostCodeType"/>
      </xs:sequence>
    </xs:choice>
```

```
    <xs:element name="Country" type="CountryType"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="StreetType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CityType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="64"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="StateType">
  <xs:restriction base="xs:string">
    <xs:minLength value="2"/>
    <xs:maxLength value="2"/>
    <xs:enumeration value="AK"/>
    <xs:enumeration value="AL"/>
    <xs:enumeration value="AR"/>
    <xs:enumeration value="AS"/>
    <xs:enumeration value="AZ"/>
    <xs:enumeration value="CA"/>
    <xs:enumeration value="CO"/>
    <xs:enumeration value="CT"/>
    <xs:enumeration value="DC"/>
    <xs:enumeration value="DE"/>
    <xs:enumeration value="FL"/>
    <xs:enumeration value="FM"/>
    <xs:enumeration value="GA"/>
    <xs:enumeration value="GU"/>
    <xs:enumeration value="HI"/>
    <xs:enumeration value="IA"/>
    <xs:enumeration value="ID"/>
    <xs:enumeration value="IL"/>
    <xs:enumeration value="IN"/>
    <xs:enumeration value="KS"/>
    <xs:enumeration value="KY"/>
    <xs:enumeration value="LA"/>
    <xs:enumeration value="MA"/>
    <xs:enumeration value="MD"/>
    <xs:enumeration value="ME"/>
    <xs:enumeration value="MH"/>
    <xs:enumeration value="MI"/>
    <xs:enumeration value="MN"/>
    <xs:enumeration value="MO"/>
    <xs:enumeration value="MP"/>
    <xs:enumeration value="MQ"/>
    <xs:enumeration value="MS"/>
    <xs:enumeration value="MT"/>
    <xs:enumeration value="NC"/>
    <xs:enumeration value="ND"/>
    <xs:enumeration value="NE"/>
    <xs:enumeration value="NH"/>
    <xs:enumeration value="NJ"/>
    <xs:enumeration value="NM"/>
    <xs:enumeration value="NV"/>
    <xs:enumeration value="NY"/>
    <xs:enumeration value="OH"/>
    <xs:enumeration value="OK"/>
    <xs:enumeration value="OR"/>
    <xs:enumeration value="PA"/>
    <xs:enumeration value="PR"/>
    <xs:enumeration value="PW"/>
```

```
      <xs:enumeration value="RI"/>
      <xs:enumeration value="SC"/>
      <xs:enumeration value="SD"/>
      <xs:enumeration value="TN"/>
      <xs:enumeration value="TX"/>
      <xs:enumeration value="UM"/>
      <xs:enumeration value="UT"/>
      <xs:enumeration value="VA"/>
      <xs:enumeration value="VI"/>
      <xs:enumeration value="VT"/>
      <xs:enumeration value="WA"/>
      <xs:enumeration value="WI"/>
      <xs:enumeration value="WV"/>
      <xs:enumeration value="WY"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ZipCodeType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{5}"/>
      <xs:pattern value="\d{5}-\d{4}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CountryType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="64"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CountyType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="32"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="PostCodeType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="12"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ProvinceType">
    <xs:restriction base="xs:string">
      <xs:minLength value="2"/>
      <xs:maxLength value="2"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NotesType">
    <xs:restriction base="xs:string">
      <xs:maxLength value="32767"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="UPCCodeType">
    <xs:restriction base="xs:string">
      <xs:minLength value="11"/>
      <xs:maxLength value="14"/>
      <xs:pattern value="\d{11}"/>
      <xs:pattern value="\d{12}"/>
      <xs:pattern value="\d{13}"/>
      <xs:pattern value="\d{14}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Loading XML Using C (OCI)

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

This example is partially listed in Chapter 3, "Using Oracle XML DB", "Loading XML Content Using C" on page 3-6.

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

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

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

```
  {
    printf("FAILED: OCIHandleAlloc() on errhp\n");
    return OCI_ERROR;
  }
/* Allocate server handle */
if (status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp,
                            (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on srvhp\n");
    return OCI_ERROR;
  }
/* Allocate service context handle */
if (status = OCIHandleAlloc((dvoid *) envhp,
                            (dvoid **) &(svchp), (ub4) OCI_HTYPE_SVCCTX,
                            (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on svchp\n");
    return OCI_ERROR;
  }
/* Allocate session handle */
if (status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &sesshp ,
                            (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on sesshp\n");
    return OCI_ERROR;
  }
/* Allocate statement handle */
if (OCIHandleAlloc((dvoid *)envhp, (dvoid **) &stmthp,
                   (ub4)OCI_HTYPE_STMT, (CONST size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on stmthp\n");
    return status;
  }
if (status = OCIServerAttach((OCIServer *) srvhp, (OCIError *) errhp,
                             (CONST oratext *)"", 0, (ub4) OCI_DEFAULT))
  {
    printf("FAILED: OCIServerAttach() on srvhp\n");
    return OCI_ERROR;
  }
/* Set server attribute to service context */
if (status = OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                        (dvoid *) srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER,
                        (OCIError *) errhp))
  {
    printf("FAILED: OCIAttrSet() on svchp\n");
    return OCI_ERROR;
  }
/* Set user attribute to session */
if (status = OCIAttrSet((dvoid *)sesshp, (ub4) OCI_HTYPE_SESSION,
                        (dvoid *)username,
                        (ub4) strlen((const char *)username),
                        (ub4) OCI_ATTR_USERNAME, (OCIError *) errhp))
  {
    printf("FAILED: OCIAttrSet() on authp for user\n");
    return OCI_ERROR;
  }
/* Set password attribute to session */
if (status = OCIAttrSet((dvoid *) sesshp, (ub4) OCI_HTYPE_SESSION,
                        (dvoid *)password,
                        (ub4) strlen((const char *)password),
                        (ub4) OCI_ATTR_PASSWORD, (OCIError *) errhp))
  {
    printf("FAILED: OCIAttrSet() on authp for password\n");
    return OCI_ERROR;
  }
/* Begin a session  */
```

```
  if (status = OCISessionBegin((OCISvcCtx *) svchp,
                               (OCIError *) errhp,
                               (OCISession *) sesshp, (ub4) OCI_CRED_RDBMS,
                               (ub4) OCI_STMT_CACHE))
    {
      printf("FAILED: OCISessionBegin(). Make sure database is up and the username/password is valid. \n");
      return OCI_ERROR;
    }
  /* Set session attribute to service context */
  if (status = OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                          (dvoid *)sesshp, (ub4) 0, (ub4) OCI_ATTR_SESSION,
                          (OCIError *) errhp))
    {
      printf("FAILED: OCIAttrSet() on svchp\n");
      return OCI_ERROR;
    }
}

/* Free OCI handles, and disconnect                        */
void free_oci()
{
  sword  status = 0;

  /* End the session */
  if (status = OCISessionEnd((OCISvcCtx *)svchp, (OCIError *)errhp,
                             (OCISession *)sesshp, (ub4) OCI_DEFAULT))
    {
      if (envhp)
        OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
      return;
    }
  /* Detach from the server */
  if (status = OCIServerDetach((OCIServer *)srvhp, (OCIError *)errhp,
                               (ub4)OCI_DEFAULT))
    {
      if (envhp)
        OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
      return;
    }
  /* Free the handles */
  if (stmthp) OCIHandleFree((dvoid *)stmthp, (ub4) OCI_HTYPE_STMT);
  if (sesshp) OCIHandleFree((dvoid *)sesshp, (ub4) OCI_HTYPE_SESSION);
  if (svchp) OCIHandleFree((dvoid *)svchp, (ub4) OCI_HTYPE_SVCCTX);
  if (srvhp) OCIHandleFree((dvoid *)srvhp, (ub4) OCI_HTYPE_SERVER);
  if (errhp) OCIHandleFree((dvoid *)errhp, (ub4) OCI_HTYPE_ERROR);
  if (envhp) OCIHandleFree((dvoid *)envhp, (ub4) OCI_HTYPE_ENV);
  return;
}

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

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

  /* Get an XML context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
```

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

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

  /* Free XML instances */
  if (doc) XmlFreeDocument((xmlctx *)xctx, (xmldocnode *)doc);

  /* Free XML CTX */
  OCIXmlDbFreeXmlCtx(xctx);
  free_oci();
}
```

Initializing and Terminating an XML Context (OCI)

Example D–4 shows how to use OCI functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()` to initialize and terminate the XML context. It constructs an XML document using the C DOM API and saves it to the database.

Example D–4 is partially listed in Chapter 14, "Using the C API for XML", "Initializing and Terminating an XML Context" on page 14-3. It assumes that the following SQL code has first been executed to create table `my_table` in database schema `CAPIUSER`:

```
CONNECT CAPIUSER/CAPIUSER
CREATE TABLE my_table OF XMLType;
```

Example D–4 Using OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx()

```
#ifndef S_ORACLE
#include <s.h>
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
#ifndef XML_ORACLE
#include <xml.h>
#endif
```

```
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>

typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
        oratext *password;
} test_ctx;

/* Helper function 1: execute a sql statement which binds xml data */
STATICF sword exec_bind_xml(OCISvcCtx *svchp,
                    OCIError *errhp,
                    OCIStmt *stmthp,
                    void *xml,
                    OCIType *xmltdo,
                    OraText *sqlstmt);

/* Helper function 2: Initialize OCI handles and connect */
STATICF sword init_oci_handles(test_ctx *ctx);

/* Helper function 3: Free OCI handles and disconnect */
STATICF sword free_oci_handles(test_ctx *ctx);

void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;

  oratext ins_stmt[] = "insert into my_table values (:1)";
  oratext tlpxml_test_sch[] = "<TOP/>";
  ctx->username = (oratext *)"CAPIUSER";
  ctx->password = (oratext *)"CAPIUSER";

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

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

  /* Start processing - first, check that this DOM supports XML 1.0 */
```

```
  printf("\n\nSupports XML 1.0? : %s\n",
         XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
         "YES" : "NO");

  /* Parse a document */
  if (!(doc = XmlLoadDom(xctx, &err, "buffer", tlpxml_test_sch,
                         "buffer_length", sizeof(tlpxml_test_sch)-1,
                         "validate", TRUE, NULL)))
  {
    printf("Parse failed, code %d\n", err);
  }
  else
  {
    /* Get the document element */
    top = (xmlnode *)XmlDomGetDocElem(xctx, doc);

    /* Print out the top element */
    printf("\n\nOriginal top element is :\n");
    XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

    /* Print out the document-note that the changes are reflected here */
    printf("\n\nOriginal document is :\n");
    XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

    /* Create some elements and add them to the document */
    quux = (xmlnode *) XmlDomCreateElem(xctx ,doc, (oratext *) "QUUX");
    foo = (xmlnode *) XmlDomCreateElem(xctx, doc, (oratext *) "FOO");
    foo_data = (xmlnode *) XmlDomCreateText(xctx, doc, (oratext *) "data");
    foo_data = XmlDomAppendChild(xctx, (xmlnode *) foo, (xmlnode *) foo_data);
    foo = XmlDomAppendChild(xctx, quux, foo);
    quux = XmlDomAppendChild(xctx, top, quux);

    /* Print out the top element */
    printf("\n\nNow the top element is :\n");
    XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

    /* Print out the document. Note that the changes are reflected here */
    printf("\n\nNow the document is :\n");
    XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

    /* Insert the document into my_table */
    status = OCITypeByName(ctx->envhp, ctx->errhp, ctx->svchp,
                           (const text *) "SYS", (ub4) strlen((char *)"SYS"),
                           (const text *) "XMLTYPE",
                           (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
                           (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                           (OCIType **) &xmltdo);
    if (status == OCI_SUCCESS)
    {
      exec_bind_xml(ctx->svchp, ctx->errhp, ctx->stmthp, (void *)doc, xmltdo,
                    ins_stmt);
    }
  }
  /* Free xml ctx */
  OCIXmlDbFreeXmlCtx(xctx);

  /* Free envhp, svchp, errhp, stmthp */
  free_oci_handles(ctx);
}
```

```
/* Helper function 1: execute a SQL statement that binds xml data */
STATICF sword exec_bind_xml(OCISvcCtx *svchp,
                    OCIError *errhp,
                    OCIStmt *stmthp,
                    void *xml,
                    OCIType *xmltdo,
                    OraText *sqlstmt)
{
  OCIBind *bndhp1 = (OCIBind *) 0;
  sword status = 0;
  OCIInd ind = OCI_IND_NOTNULL;
  OCIInd *indp = &ind;
  if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                             (ub4)strlen((char *)sqlstmt),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)) {
    printf("Failed OCIStmtPrepare\n");
    return OCI_ERROR;
  }
  if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *) 0,
                           (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                           (ub2 *)0, (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)) {
    printf("Failed OCIBindByPos\n");
    return OCI_ERROR;
  }
  if(status = OCIBindObject(bndhp1, errhp, (CONST OCIType *) xmltdo, (dvoid **)
                            &xml,
                            (ub4 *) 0, (dvoid **) &indp, (ub4 *) 0)) {
    printf("Failed OCIBindObject\n");
    return OCI_ERROR;
  }
  if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                             (ub4) OCI_DEFAULT)) {
    printf("Failed OCIStmtExecute\n");
    return OCI_ERROR;
  }
  return OCI_SUCCESS;
}

/* Helper function 2: Initialize OCI handles and connect */
STATICF sword init_oci_handles(test_ctx *ctx)
{
  sword status;
  ctx->dur = OCI_DURATION_SESSION;
  if (OCIEnvCreate((OCIEnv **) &(ctx->envhp), (ub4) OCI_OBJECT,
                   (dvoid *) 0, (dvoid * (*)(dvoid *,size_t)) 0,
                   (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
                   (void (*)(dvoid *, dvoid *)) 0, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIEnvCreate()\n");
    return OCI_ERROR;
  }
  /* Allocate error handle */
  if (OCIHandleAlloc((dvoid *) ctx->envhp, (dvoid **) &(ctx->errhp),
                     (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on errhp\n");
    return OCI_ERROR;
  }
  /* Allocate server handle */
```

```
if (status = OCIHandleAlloc((dvoid *) ctx->envhp, (dvoid **) &ctx->srvhp,
                        (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on srvhp\n");
  return OCI_ERROR;
}
/* Allocate service context handle */
if (status = OCIHandleAlloc((dvoid *) ctx->envhp,
                           (dvoid **) &(ctx->svchp), (ub4) OCI_HTYPE_SVCCTX,
                           (size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on svchp\n");
  return OCI_ERROR;
}
/* Allocate session handle */
if (status = OCIHandleAlloc((dvoid *) ctx->envhp, (dvoid **) &ctx->sesshp ,
                        (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on sesshp\n");
  return OCI_ERROR;
}
/* Allocate statement handle */
if (OCIHandleAlloc((dvoid *)ctx->envhp, (dvoid **) &ctx->stmthp,
                   (ub4)OCI_HTYPE_STMT, (CONST size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on stmthp\n");
  return status;
}
if (status = OCIServerAttach((OCIServer *) ctx->srvhp, (OCIError *) ctx->errhp,
                            (CONST oratext *)"", 0, (ub4) OCI_DEFAULT))
{
  printf("FAILED: OCIServerAttach() on srvhp\n");
  return OCI_ERROR;
}
/* Set server attribute to service context */
if (status = OCIAttrSet((dvoid *) ctx->svchp, (ub4) OCI_HTYPE_SVCCTX,
                        (dvoid *) ctx->srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER,
                        (OCIError *) ctx->errhp))
{
  printf("FAILED: OCIAttrSet() on svchp\n");
  return OCI_ERROR;
}
/* Set user attribute to session */
if (status = OCIAttrSet((dvoid *)ctx->sesshp, (ub4) OCI_HTYPE_SESSION,
                        (dvoid *)ctx->username,
                        (ub4) strlen((char *)ctx->username),
                        (ub4) OCI_ATTR_USERNAME, (OCIError *) ctx->errhp))
{
  printf("FAILED: OCIAttrSet() on authp for user\n");
  return OCI_ERROR;
}
/* Set password attribute to session */
if (status = OCIAttrSet((dvoid *) ctx->sesshp, (ub4) OCI_HTYPE_SESSION,
                        (dvoid *)ctx->password,
                        (ub4) strlen((char *)ctx->password),
                        (ub4) OCI_ATTR_PASSWORD, (OCIError *) ctx->errhp))
{
  printf("FAILED: OCIAttrSet() on authp for password\n");
  return OCI_ERROR;
}
```

```
  /* Begin a session  */
  if (status = OCISessionBegin((OCISvcCtx *) ctx->svchp,
                               (OCIError *) ctx->errhp,
                               (OCISession *) ctx->sesshp, (ub4) OCI_CRED_RDBMS,
                               (ub4) OCI_STMT_CACHE))
  {
    printf("FAILED: OCISessionBegin(). Make sure database is up and the \
            username/password is valid. \n");
    return OCI_ERROR;
  }
  /* Set session attribute to service context */
  if (status = OCIAttrSet((dvoid *) ctx->svchp, (ub4) OCI_HTYPE_SVCCTX,
                          (dvoid *)ctx->sesshp, (ub4) 0, (ub4) OCI_ATTR_SESSION,
                          (OCIError *) ctx->errhp))
  {
    printf("FAILED: OCIAttrSet() on svchp\n");
    return OCI_ERROR;
  }
  return status;
}

/* Helper function 3: Free OCI handles and disconnect */
STATICF sword free_oci_handles(test_ctx *ctx)
{
  sword  status = 0;
  /* End the session */
  if (status = OCISessionEnd((OCISvcCtx *)ctx->svchp, (OCIError *)ctx->errhp,
                             (OCISession *)ctx->sesshp, (ub4) OCI_DEFAULT))
  {
    if (ctx->envhp)
      OCIHandleFree((dvoid *)ctx->envhp, OCI_HTYPE_ENV);
    return status;
  }
  /* Detach from the server */
  if (status = OCIServerDetach((OCIServer *)ctx->srvhp, (OCIError *)ctx->errhp,
      (ub4)OCI_DEFAULT))
  {
    if (ctx->envhp)
      OCIHandleFree((dvoid *)ctx->envhp, OCI_HTYPE_ENV);
    return status;
  }
  /* Free the handles */
  if (ctx->stmthp) OCIHandleFree((dvoid *)ctx->stmthp, (ub4) OCI_HTYPE_STMT);
  if (ctx->sesshp) OCIHandleFree((dvoid *)ctx->sesshp, (ub4) OCI_HTYPE_SESSION);
  if (ctx->svchp)  OCIHandleFree((dvoid *)ctx->svchp,  (ub4) OCI_HTYPE_SVCCTX);
  if (ctx->srvhp)  OCIHandleFree((dvoid *)ctx->srvhp,  (ub4) OCI_HTYPE_SERVER);
  if (ctx->errhp)  OCIHandleFree((dvoid *)ctx->errhp,  (ub4) OCI_HTYPE_ERROR);
  if (ctx->envhp)  OCIHandleFree((dvoid *)ctx->envhp,  (ub4) OCI_HTYPE_ENV);
  return status;
}
```

E

Oracle XML DB Restrictions

This appendix describes the restrictions associated with Oracle XML DB.

- **Replication –** Oracle XML DB does not support replication of `XMLType` tables.
- **Extending Resource Metadata Properties –** 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 support references within a scalar, `XMLType`, or LOB data column.
- **Thin JDBC Driver Not Supported by Some XMLType Functions –** Methods `extract()`, `transform()`, and `existsNode()` work only with the *thick* JDBC driver. Not all `oracle.xdb.XMLType` functions are supported by the thin JDBC driver. If you do not use `oracle.xdb.XMLType` classes and the OCI driver, you could lose performance benefits.
- **NVARCHAR and NCHAR SQLTypes 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 file (.xsd) 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.
- **Identifier Length Limited to 4000 Characters –** Oracle XML DB supports only XML identifiers that are 4000 characters long, or shorter.
- **64K Limit on Text Nodes and Attribute Values** – Each text node or attribute value processed by Oracle XML DB is limited in size to 64K bytes.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X