DB2

IBM

**DB2 Version 9**
for Linux, UNIX, and Windows

# XML Extender Administration and Programming

SC18-9750-00

DB2®

IBM

**DB2 Version 9**
for Linux, UNIX, and Windows

# XML Extender Administration and Programming

SC18-9750-00

Before using this information and the product it supports, be sure to read the general information under *Notices*.

**Edition Notice**

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

**© Copyright International Business Machines Corporation 1999 - 2006. All rights reserved.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents


**About this guide . . . . . vii**
Who should use this guide . . . . . vii
How to get a current version of this guide . . . . vii
How to use this guide. . . . . vii
Highlighting conventions . . . . . viii

**Part 1. Introduction . . . . . 1**

**Chapter 1. Introduction . . . . . 3**
Introduction to XML Extender . . . . . 3
XML Documents . . . . . 3
How XML data is handled in DB2 . . . . . 4
Features of XML Extender . . . . . 5
XML Extender tutorial lessons . . . . . 6
Lesson: Storing an XML document in an XML column . . . . . 7
Lesson: Composing an XML document . . . . . 17

**Part 2. Administration . . . . . 33**

**Chapter 2. Administration . . . . . 35**
Administration tools for XML Extender . . . . . 35
Preparing to administer XML Extender . . . . . 35
Migrating XML Extender from previous versions or releases. . . . . 35
XML Extender administration overview . . . . . 36
XML Extender administration wizard. . . . . 36
Access and storage methods. . . . . 36
When to use the XML column method . . . . . 38
When to use the XML collection method . . . . 38
Planning for XML columns . . . . . 38
XML data types for the XML columns . . . . 38
Elements and attributes to index for XML columns . . . . . 39
The DAD file for XML columns . . . . . 39
Planning for XML collections . . . . . 40
Validation . . . . . 40
The DAD file for XML collections . . . . . 40
Mapping schemes for XML collections . . . . 42
Types of mapping schemes . . . . . 43
Mapping scheme requirements . . . . . 45
Decomposition table size requirements for RDB node mapping . . . . . 49
Validating XML documents automatically . . . . 50
Enabling databases for XML. . . . . 50
Creating an XML table. . . . . 51
Storing a DTD in the repository table. . . . . 52
Enabling XML columns . . . . . 53
Planning side tables . . . . . 56
Indexing side tables . . . . . 57
Composing XML documents by using SQL mapping 58
Composing XML collections by using RDB_node mapping . . . . . 61
Decomposing an XML collection by using RDB_node mapping . . . . . 65

**Part 3. Programming . . . . . 69**

**Chapter 3. XML columns . . . . . 71**
Managing data in XML columns . . . . . 71
XML columns as a storage and access method . . . 71
Defining and enabling an XML column . . . . . 72
Using indexes for XML column data . . . . . 73
Storing XML data . . . . . 74
Default casting functions for storing XML data 75
Storage UDFs for storing XML data . . . . . 75
Method for retrieving an XML document . . . . 76
Retrieving an entire XML document . . . . . 77
Default casting functions for retrieving XML data . . . . . 77
Using the Content() UDF for retrieving XML data . . . . . 77
Retrieving element contents and attribute values from XML documents . . . . . 79
Updating XML data . . . . . 80
Updating an entire XML document . . . . . 81
Updating with a default casting function . . 81
Updating XML documents with a storage UDF. . . . . 81
Updating specific elements and attributes of an XML document . . . . . 81
Methods for searching XML documents . . . . . 82
Searching the XML document by structure . . . 82
Example: searching with direct query on side tables . . . . . 83
Example: searching from a joined view . . . 83
Example: searching with extracting UDFs . . 83
Example: searching on elements or attributes with multiple occurrence . . . . . 84
Deleting XML documents. . . . . 84
Limitations when invoking functions from Java Database (JDBC). . . . . 84

**Chapter 4. Managing data in XML collections . . . . . 87**
XML collections as a storage and access method . . 87
Managing data in XML collections. . . . . 88
Preparing to compose XML documents from DB2 data . . . . . 88
Composing documents using dxxGenXML . . 88
Composing documents using dxxRetrieveXML 90
Decomposing XML documents into DB2 database data . . . . . 92
Enabling an XML collection for decomposition . 94
Decomposition table size limits. . . . . 95
Updating and deleting data in XML collections . . 95
Updating data in an XML collection . . . . . 96


© Copyright IBM Corp. 1999 - 2006

Updating element and attribute values . . . 96
Deleting element and attribute instances. . . 96
Deleting an XML document from an XML collection . . . . . 97
Searching XML collections . . . . . 97
Composing XML documents using search criteria 97
Searching for decomposed XML data . . . . . 98
Mapping schemes for XML collections . . . . . 99
Requirements for using SQL mapping . . . . . 101
Requirements for RDB_Node mapping . . . . . 103
Stylesheets for an XML collection. . . . . 106
Location paths . . . . . 106
Location path syntax . . . . . 107
Enabling XML collections . . . . . 109
Disabling XML collections . . . . . 110

**Chapter 5. XML schemas . . . . . 113**
Advantages of using XML schemas instead of DTDs . . . . . 113
XML schema complexType element . . . . . 113
Data types, elements and attributes in schemas . . 114
Simple data types in XML schemas . . . . . 114
Elements in XML schemas . . . . . 115
Attributes in XML schemas. . . . . 115
Examples of an XML schema . . . . . 115
XML document instance using the schema. . . 116
XML document instance using a DTD . . . . 116

**Part 4. Reference . . . . . 119**

**Chapter 6. The dxxadm administration command . . . . . 121**
dxxadm command overview . . . . . 121
Syntax of the dxxadm administration command 121
Options for the administration command . . . . 121
enable_db option of the dxxadm command . . 122
disable_db option of the dxxadm command . . 123
enable_column option of the dxxadm command 124
disable_column option of the dxxadm command 125
enable_collection option of the dxxadm command . . . . . 126
disable_collection option of the dxxadm command . . . . . 127

**Chapter 7. XML Extender user-defined types . . . . . 129**

**Chapter 8. XML Extender user-defined functions . . . . . 131**
Types of XML Extender user-defined functions . . 131
UDT and UDF names for XML Extender . . . . 132
Storage functions . . . . . 132
Storage functions in XML Extender overview 132
XMLCLOBFromFile() function. . . . . 132
XMLFileFromCLOB() function. . . . . 133
XMLFileFromVarchar() function . . . . . 134
XMLVarcharFromFile() function . . . . . 135
Retrieval functions . . . . . 135
Retrieval functions in XML Extender . . . . 135
Content(): retrieve from XMLFILE to a CLOB 136
Content(): retrieve from XMLVARCHAR to an external server file. . . . . 137
Content(): retrieval from XMLCLOB to an external server file. . . . . 138
Extraction functions . . . . . 140
Extracting functions in XML Extender . . . . 140
extractInteger() and extractIntegers(). . . . . 140
extractSmallint() and extractSmallints() . . . . 141
extractDouble() and extractDoubles() . . . . 142
extractReal() and extractReals() . . . . . 143
extractChar() and extractChars() . . . . . 145
extractVarchar() and extractVarchars() . . . . 146
extractCLOB() and extractCLOBs() . . . . . 147
extractDate() and extractDates() . . . . . 148
extractTime() and extractTimes() . . . . . 149
extractTimestamp() and extractTimestamps() . . 150
Update functions in XML Extender . . . . . 152
How the Update() function processes the XML document . . . . . 155
Multiple occurrence . . . . . 156
Validation functions . . . . . 156
SVALIDATE() function . . . . . 157
Syntax. . . . . 157
Parameters . . . . . 157
Examples. . . . . 157
DVALIDATE() function . . . . . 158
Syntax. . . . . 158
Parameters . . . . . 158
Examples. . . . . 159

**Chapter 9. Document access definition (DAD) files . . . . . 161**
Creating a DAD file for XML columns . . . . . 161
DAD files for XML collections. . . . . 163
SQL composition . . . . . 165
RDB node composition . . . . . 165
Composition from rows that have null values 166
DTD for the DAD file . . . . . 166
Dynamically overriding values in the DAD file . . 171

**Chapter 10. XML Extender stored procedures . . . . . 179**
XML Extender stored procedures - Overview. . . 179
Calling XML Extender stored procedures . . . . 179
Stored Procedures that return CLOBs . . . . . 180
XML Extender administration stored procedures 181
XML Extender administration stored procedures - Overview . . . . . 181
dxxEnableDB() stored procedure . . . . . 182
dxxDisableDB() stored procedure. . . . . 182
dxxEnableColumn() stored procedure . . . . 183
dxxDisableColumn() stored procedure . . . . 184
dxxEnableCollection() stored procedure . . . 185
dxxDisableCollection() stored procedure . . . 186
XML Extender composition stored procedures . . 186
XML Extender composition stored procedures - Overview. . . . . 186
dxxGenXML() stored procedure . . . . . 187
dxxRetrieveXML() stored procedure . . . . . 190

dxxGenXMLClob stored procedure . . . . . 193
dxxRetrieveXMLClob stored procedure . . . . 195
XML Extender decomposition stored procedures 197
XML Extender decomposition stored procedures - Overview . . . . . 197
dxxShredXML() stored procedure. . . . . 197
dxxInsertXML() stored procedure. . . . . 199

**Chapter 11. XML Extender stored procedures and functions for MQSeries . . . . . 201**
XML Extender stored procedures and functions for MQSeries - Overview. . . . . 201
XML Extender MQSeries functions . . . . . 202
XML Extender MQSeries functions - Overview 202
MQPublishXML function . . . . . 202
MQReadXML function . . . . . 204
MQReadAllXML function . . . . . 205
MQReadXMLCLOB function . . . . . 207
MQReadAllXMLCLOB function . . . . . 208
MQReceiveXML function . . . . . 210
MQReceiveAllXML function . . . . . 212
MQRcvAllXMLCLOB function. . . . . 214
MQReceiveXMLCLOB function . . . . . 215
MQRcvXMLCLOB function. . . . . 216
MQSENDXML function . . . . . 217
MQSENDXMLFILE function . . . . . 219
MQSendXMLFILECLOB function. . . . . 220
XML Extender MQSeries stored procedures . . . 221
XML Extender MQSeries stored procedures - Overview. . . . . 221
dxxmqGen() stored procedure . . . . . 223
dxxmqGenCLOB stored procedure . . . . . 226
dxxmqRetrieve stored procedure . . . . . 228
dxxmqRetrieveCLOB stored procedure . . . . 230
dxxmqShred stored procedure. . . . . 232
dxxmqShredAll stored procedure. . . . . 234
dxxmqShredCLOB stored procedure. . . . . 235
dxxmqShredAllCLOB stored procedure. . . . 236
dxxmqInsert stored procedure. . . . . 237
dxxmqInsertCLOB stored procedure. . . . . 239
dxxmqInsertAll stored procedure. . . . . 241
dxxmqInsertAllCLOB stored procedure. . . . 242

**Chapter 12. Extensible stylesheet language transformation (XSLT) . . . 245**
Creating an HTML document using an XSLT stylesheet. . . . . 245
XSLTransformToClob() user-defined function . . . 246
XSLTransformToFile() user-defined function . . . 247

**Chapter 13. XML Extender administration support tables . . . . 251**
DTD reference table . . . . . 251
XML usage table (XML_USAGE) . . . . . 251

**Chapter 14. Troubleshooting . . . . . 253**
Troubleshooting XML Extender . . . . . 253
Starting the trace for XML Extender . . . . . 253
Stopping the trace . . . . . 254
XML Extender UDF return codes . . . . . 254
XML Extender stored procedure return codes. . . 255
SQLSTATE codes and associated message numbers for XML Extender . . . . . 255
XML Extender messages. . . . . 260

**Appendix A. Samples . . . . . 275**
XML DTD sample . . . . . 275
XML document sample: getstart.xml. . . . . 275
Stylesheet sample: getstart.xsl . . . . . 276
Document access definition files . . . . . 277
Sample DAD file: XML column . . . . . 278
Sample DAD file: XML collection: SQL mapping 278
Sample DAD file: XML: RDB_node mapping 280

**Appendix B. Code page considerations . . . . . 283**
Terminology for XML code pages. . . . . 283
DB2 and XML Extender code page assumptions 284
Assumptions for importing an XML document 284
Assumptions for exporting an XML document 285
Encoding declaration considerations for XML Extender . . . . . 285
Legal encoding declarations . . . . . 285
Consistent encodings and encoding declarations 286
Declaring an encoding . . . . . 288
Conversion scenarios . . . . . 288
Recommendations for preventing inconsistent XML documents . . . . . 290

**Appendix C. XML Extender limits . . . 293**

**XML Extender glossary . . . . . 297**

**Notices . . . . . 305**
Trademarks . . . . . 307

**Index . . . . . 309**

**Contacting IBM . . . . . 315**

# About this guide

This section contains the following information:


- "Who should use this guide"
- "How to use this guide"
- "Highlighting conventions" on page viii


## Who should use this guide

This guide is intended for the following people:

- Those who work with XML data in DB2® applications and who are familiar with XML concepts. Readers of this document should have a general understanding of XML and DB2. To learn more about XML, see the following Web site:

  http://www.w3.org/XML

  To learn more about DB2, see the following Web site:

  http://www.ibm.com/software/data/db2/library
- DB2 database administrators who are familiar with DB2 database administration concepts, tools, and techniques.
- DB2 application programmers who are familiar with SQL and with one or more programming languages that can be used for DB2 applications.

## How to get a current version of this guide

You can get the latest version of this guide at the XML Extender Web site:

http://www.ibm.com/software/data/db2/extenders/xmlext/library.html

## How to use this guide

This guide is structured as follows:

**Part 1. Introduction**
: This part provides an overview of the XML Extender and how you can use it in your business applications. It contains a getting-started scenario that helps you get up and running.

**Part 2. Administration**
: This part describes how to prepare and maintain a DB2 database for XML data. Read this part if you need to administer a DB2 database that contains XML data.

**Part 3. Programming**
: This part describes how to manage your XML data. Read this part if you need to access and manipulate XML data in a DB2 database application program.

**Part 4. Reference**
: This part describes how to use the XML Extender administration commands, user-defined types, user-defined functions, and stored procedures. It also lists the messages and codes that the XML Extender issues. Read this part if you are familiar with the XML Extender concepts

© Copyright IBM Corp. 1999 - 2006

and tasks, but you need information about a user-defined type (UDT), user-defined function (UDF), command, message, metadata tables, control tables, or code.

**Part 5. Appendixes**
The appendixes describe the DTD for the document access definition, samples, and XML Extender limits.

## Highlighting conventions

This guide uses the following conventions:

**Bold text indicates:**

- Commands
- Field names
- Menu names
- Push buttons

**Italic text indicates**

- Variable parameters that are to be replaced with a value
- Emphasized words
- First use of a glossary term

**Uppercase letters indicate:**

- Data types
- Column names
- Table names

Example text indicates:

- System messages
- Values that you type
- Coding examples
- Directory names
- File names

# Part 1. Introduction

This part provides an overview of the XML Extender and how you can use it in your business applications.

© Copyright IBM Corp. 1999 - 2006

# Chapter 1. Introduction

## Introduction to XML Extender

DB2® XML Extender provides the ability to store and access XML documents, to generate XML documents from existing relational data, and to insert rows into relational tables from XML documents. XML Extender provides new data types, functions, and stored procedures to manage your XML data in DB2.

XML Extender is available for the following platforms. See the DB2 installation requirements to determine which operating system levels are supported.

- Windows Server on an Intel 32-bit platform
- AIX® on a 32-bit platform for XML Extender client functions
- AIX® on a 64-bit platform
- Solaris™ Operating Environment on a 32-bit platform for XML Extender client functions
- Solaris™ Operating Environment on a 64-bit platform
- Linux on a 32-bit platform with x86 architecture
- Linux on a 64-bit platform with x86 architecture
- HP-UX on a PA-RISC 32-bit platform for XML Extender client functions
- HP-UX on a PA-RISC 64-bit platform
- z/OS™
- i5/OS™

**Related concepts:**

- "XML Extender tutorial lessons" on page 6
- "Features of XML Extender" on page 5
- "Lesson: Composing an XML document" on page 17
- "Lesson: Storing an XML document in an XML column" on page 7
- "XML Documents"

## XML Documents

Because companies tend to share data between different applications, companies are continually faced with the problem of replicating, transforming, exporting, or saving their data in formats that can be imported into other applications. Many of these transforming processes tend to drop some of the data, or they at least require that users go through the tedious process of ensuring that the data remains consistent. This manual checking consumes both time and money.

One of the ways to address this problem is for application developers to write *Open Database Connectivity (ODBC)* applications, a standard application programming interface (API) for accessing data in both relational and non-relational database management systems. These applications save the data in a database management system. From there, the data can be manipulated and presented in the form in which it is needed for another application. Database applications must be written to convert the data into a form that an application requires. Applications change quickly and quickly become obsolete. Applications

© Copyright IBM Corp. 1999 - 2006

that convert data to HTML provide presentation solutions, but the data presented cannot be practically used for other purposes. A method that separates the data from its presentation is needed to provide a practical form of interchange between applications.

XML—*eXtensible Markup Language*—addresses this problem. XML is extensible because the language is a meta-language that allows you to create your own language based on the needs of your enterprise. You use XML to capture not only the data for your particular application, but also the data structure. Although it is not the only data interchange format, XML has emerged as the accepted standard. By adhering to this standard, applications can share data without first transforming it using proprietary formats.

Because XML is now the accepted standard for data interchange, many applications are emerging that will be able to take advantage of it.

Suppose that you are using a particular project management application and you want to share some of its data with your calendar application. Your project management application could export tasks in XML, which could then be imported as-is into your calendar application. In today's interconnected world, application providers have strong incentives to make an XML interchange format a basic feature of their applications.

## How XML data is handled in DB2

Although XML solves many problems by providing a standard format for data interchange, some challenges remain. When building an enterprise data application, you must answer questions such as:

- How often do I want to replicate the data?
- What kind of information must be shared between applications?
- How can I quickly search for the information I need?
- How can I make a particular action, such as a new entry being added, trigger an automatic data interchange between all of my applications?

These kinds of issues can be addressed only by a database management system. By incorporating the XML information and meta-information directly in the database, you can more efficiently obtain the XML results that your other applications need. With the XML Extender, you can take advantage of the power of DB2® in many XML applications.

With the content of your structured XML documents in a DB2 database, you can combine structured XML information with traditional relational data. Based on the application, you can choose whether to store entire XML documents in DB2 in user-defined types provided for XML data (XML data types), or you can map the XML content as base data types in relational tables. For XML data types, XML Extender adds the power to search rich data types of XML element or attribute values, in addition to the structural text search that the DB2 Database™ provides.

XML Extender provides two methods of storing and access for XML data in DB2:

**XML column method**

Stores entire XML documents as column data or externally as a file, and extracts the required XML element or attribute value and stores it in *side tables*, which are indexed tables for high-speed searching. By storing the documents as column data, you can:

- Perform fast search on XML elements or attributes that have been extracted and stored in side tables as SQL basic data types and indexed.
- Update the content of an XML element or the value of an XML attribute.
- Extract XML elements or attributes dynamically using SQL queries.
- Validate XML documents when they are inserted and updated.
- Perform structural-text search with the Net Search Extender.

**XML collection method**
Composes and decomposes contents of XML documents with one or more relational tables.

## Features of XML Extender

XML Extender provides the following features to help you manage and exploit XML data with DB2®:

- Administration resources to help you manage the integration of XML data in relational tables
- Storage and access methods for XML data within the database
- A data type definition (DTD) repository for you to store DTDs
- The ability to validate XML documents using a DTD or schema
- A mapping file called the Document Access Definition (DAD), which is used to map XML documents to relational data
- Location paths to specify the location of an element or attribute within an XML document

**Administration resources:** The XML Extender administration resources help you enable your database and table columns for XML, and map XML data to DB2 relational structures.

You can use the following resources to complete administration tasks for the XML Extender:

- The **dxxadm** command provides a command-line option for administration tasks.
- The XML Extender administration stored procedures allow you to invoke administration commands from a program.

**Storage and access methods:** XML Extender provides two storage and access methods for integrating XML documents with DB2 data structures: XML column and XML collection. These methods have very different uses, but can be used in the same application.

**XML column method**
This method helps you store intact XML documents in DB2. The XML column method works well for archiving documents. The documents are inserted into columns enabled for XML and can be updated, retrieved, and searched. Element and attribute data can be mapped to DB2 tables (side tables), which can be indexed for fast searches.

**XML collection method**
This method helps you map XML document structures to DB2 tables so

that you can either compose XML documents from existing DB2 data, or decompose XML documents, storing the untagged data in DB2 tables. This method is good for data interchange applications, particularly when the contents of XML documents are frequently updated.

**DTDs:**The XML Extender also allows you to store DTDs, the set of declarations for XML elements and attributes. When a database is *enabled* for XML, a DTD repository table (DTD_REF) is created. Each row of this table represents a DTD with additional metadata information. Users can access this table to insert their own DTDs. The DTDs are used for validating the structure of XML documents.

**DAD files:** You specify how structured XML documents are to be processed by XML Extender using a *document access definition (DAD)* file. The DAD file is an XML document that maps the XML document structure to a DB2 table. You use a DAD file when storing XML documents in a column, or when composing or decomposing XML data. The DAD file specifies whether you are storing documents using the XML column method, or defining an XML collection for composition or decomposition.

**Related concepts:**

- "XML Extender tutorial lessons"
- "Lesson: Composing an XML document" on page 17
- "Lesson: Storing an XML document in an XML column" on page 7
- "How XML data is handled in DB2" on page 4

## XML Extender tutorial lessons

This tutorial shows you how to get started using XML Extender to access and modify XML data for your applications. Three lessons are provided:

- Storing an XML document in an XML column
- Composing an XML document
- Cleaning up the database

By following the tutorial lessons, you can set up a database using provided sample data, map SQL data to an XML document, store XML documents in the database, and then search and extract data from the XML documents.

In the lessons, you will use the DB2® Command Window with XML Extender administration commands, UDFs, and stored procedures. Most of the examples in the rest of the guide draw on the sample data that is used in this section.

In these lessons, you work for ACME Auto Direct, a company that distributes cars and trucks to automotive dealerships. You have two tasks. First you will set up a system in which orders can be archived in the SALES_DB database for querying by the sales department. Next, you extract information in an existing purchase order database, SALES_DB.

**Prerequisites**

To complete the lessons in this tutorial, you must have the following prerequisites installed:

- DB2 Version 9.1 with the XML Extender feature

In addition, the DB2 database server must have been enabled by the DB2 administrator.

## Lesson: Storing an XML document in an XML column

The XML Extender provides a method of storing and accessing whole XML documents in the database . The XML column method enables you to store the document using the XML file types, index the column in side tables, and then query or search the XML document. This storage method is particularly useful for archival applications in which documents are not frequently updated.

This lesson shows you how to use the XML column storage and access method.

**The scenario:** You have the task of archiving the sales data for the service department. The sales data that you need to work with is stored in XML documents that use the same DTD.

The service department provided a recommended structure for the XML documents and specified which element data will be queried most frequently. The service department wants the XML documents stored in the SALES_TAB table in the SALES_DB database and wants to be able to search them quickly. The SALES_TAB table will contain two columns with data about each sale, and a third column will contain the XML document. This column is called ORDER.

To store this XML document in the SALES_TAB table, you will:

1. Store the document as an XMLVARCHAR data type, and determine which XML elements and attributes will be frequently queried.
2. Set up the SALES_DB database for XML.
3. Create the SALES_TAB table, and enable the ORDER column so that you can store the intact document in DB2®.
4. Insert a DTD for the XML document for validation.

When you enable the column, you will define side tables to be indexed for the structural search of the document in a document access definition (DAD) file, an XML document that specifies the structure of the side tables.

The SALES_TAB table is described in Table 1. The column to be enabled for XML, ORDER, is shown in italics.

*Table 1. SALES_TAB table*

| Column name | Data type |
|---|---|
| INVOICE_NUM | CHAR(6) NOT NULL PRIMARY KEY |
| SALES_PERSON | VARCHAR(20) |
| *ORDER* | XMLVARCHAR |

**The scripts and samples**:

For this tutorial, you use a set of scripts to set up your environment and perform the steps in the lessons. These scripts are in the *dxx_install*/samples/extenders/db2xml/cmd directory (where *dxx_install* is the directory where you installed the XML Extender files).

These scripts are:

**getstart_db.cmd**
Creates the database and populates four tables.

**getstart_prep.cmd**
Binds the database with the XML Extender stored procedures and enables the database for XML Extender.

**getstart_insertDTD.cmd**
Inserts the DTD used to validate the XML document in the XML column.

**getstart_createTabCol.cmd**
Creates an application table that will have an XML-enabled column.

**getstart_alterTabCol.cmd**
Alters the application table by adding the column that will be enabled for XML.

**getstart_enableCol.cmd**
Enables the XML column.

**getstart_createIndex.cmd**
Creates indexes on the side tables for the XML column.

**getstart_insertXML.cmd**
Inserts the XML document into the XML column.

**getstart_queryCol.cmd**
Runs a select statement on the application table and returns the XML document.

**getstart_stp.cmd**
Runs the stored procedure to compose the XML collection.

**getstart_exportXML.cmd**
Exports the XML document from the database for use in an application.

**getstart_clean.cmd**
Cleans up the tutorial environment.

**Planning how to store the document:**

Before you use the XML Extender to store your documents, you need to:

- Understand the XML document structure.
- Determine the XML user-defined type in which you will store the XML document.
- Determine the XML elements and attributes that the service department will frequently search, so that the content of these can be stored in side tables and indexed to improve performance.

The following sections will explain how to make these decisions.

**The XML document structure:**

The XML document structure for this lesson takes information for a specific order that is structured by the order key as the top level, then customer, part, and shipping information on the next level.

This lesson provides the sample DTD for you to understand and validate the XML document structure.

**Determining the XML data type for the XML column:**

The XML Extender provides XML user defined types that you can use to define a column to hold XML documents. These data types are:

**XMLVARCHAR**
Used for documents smaller than 3 kilobytes stored in DB2. The maximum size of XMLVARCHAR documents can be redefined to as large as 32672 bytes.

**XMLCLOB**
Used for documents larger than 3 kilobytes stored in DB2. The maximum document size is 2 gigabytes.

**XMLFILE**
Used for documents stored outside DB2.

In this lesson, you will store a small document in DB2, so you will use the XMLVARCHAR data type.

**Determining elements and attributes to be searched:**

When you understand the XML document structure and the needs of the application, you can determine which elements and attributes will be searched or extracted most frequently. The service department will frequently query the order key, customer name, price, and shipping date of an order, and they will need quick performance for these searches. This information is contained in elements and attributes of the XML document. Table 2 describes the location paths of each element and attribute.

*Table 2. Elements and attributes to be searched*

| Data | Location Path |
|---|---|
| order key | /Order/@Key |
| customer name | /Order/Customer/Name |
| price | /Order/Part/ExtendedPrice |
| shipping date | /Order/Part/Shipment/ShipDate |

**Mapping the XML document to the side tables:**

To map your XML documents to a side table, you must create a DAD file for the XML column. The DAD file is used to store the XML document in DB2. It also maps the XML element and attribute contents to DB2 database side tables used for indexing, which improves search performance.

After identifying the elements and attributes to be searched, you determine how they should be organized in the side tables, how many tables to use, and which columns are in what table. Organize the side tables by putting similar information in the same table. The number of side tables is also determined by whether the location path of any elements can be repeated more than once in the document. For example, in the document, the part element can be repeated multiple times, and therefore, the price and date elements can occur multiple times. Elements that can occur multiple times must each be in their own side tables.

You must also determine what DB2 database base types the element or attribute values should use, which is determined by the format of the data.

- If the data is text, use VARCHAR.
- If the data is an integer, use INTEGER.

- If the data is a date, and you want to do range searches, use DATE.

In this tutorial, the elements and attributes are mapped to either ORDER_SIDE_TAB, PART_SIDE_TAB or, SHIP_SIDE_TAB. The tables below show which table each element or attribute is mapped to.

**ORDER_SIDE_TAB**

| Column name | Data type | Location path | Multiple occurring? |
|---|---|---|---|
| ORDER_KEY | INTEGER | /Order/@Key | No |
| CUSTOMER | VARCHAR(16) | /Order/Customer/Name | No |

**PART_SIDE_TAB**

| Column name | Data type | Location path | Multiple occurring? |
|---|---|---|---|
| PRICE | DECIMAL(10,2) | /Order/Part/ExtendedPrice | Yes |

**SHIP_SIDE_TAB**

| Column name | Data type | Location path | Multiple occurring? |
|---|---|---|---|
| DATE | DATE | /Order/Part/Shipment/ShipDate | Yes |

**Creating the SALES_DB database:**

In this task, you create a SALES_DB database and enable the database for XML.

To create the database:

1. Change to the `dxx_install/samples/extenders/db2xml/cmd` directory, where `dxx_install` is the directory where you installed the XML Extender files. The sample files contain references to files that use absolute path names. Check the sample files and change these values for your directory paths.
2. On Windows® platforms, open a DB2 command window by typing the following text from a Command Prompt window:
   ```
   DB2CMD
   ```
3. Run the **getstart_db** command:
   ```
   getstart_db.cmd
   ```

**Enabling the database:**

To store XML information in the database, you need to enable it for the XML Extender. When you enable a database for XML, XML Extender:

- Creates user-defined types (UDTs), user-defined functions (UDFs), and stored procedures
- Creates and populates control tables with the necessary metadata that the XML Extender requires
- Creates the DB2XML schema and assigns the necessary privileges

To enable the database for XML:

Run the following script:

```
getstart_prep.cmd
```

This script binds the database to the XML Extender stored procedures. It also runs the **dxxadm** command option that enables the database:

```
dxxadm enable_db   SALES_DB
```

**Enabling the XML column and storing the document:**

In this lesson, you will enable a column for XML Extender and store an XML document in the column. For these tasks, you will:

1. Store the DTD in the DTD repository.
2. Create a DAD file for the XML column.
3. Create the SALES_TAB table.
4. Add the column of XML type .
5. Enable the XML column.
6. View the column and side tables.
7. Index the side tables for structural search.
8. Store the XML document.
9. Query the XML document.

**Storing the DTD in the DTD repository:**

You can use a DTD to validate XML data in an XML column. The XML Extender creates a table in the XML-enabled database, called DTD_REF. The table is known as the DTD repository and is available for you to store DTDs. When you validate XML documents, you must store the DTD in this repository. The DTD for this lesson is in

*dxx_install*`/samples/extenders/db2xml/dtd/getstart.dtd`

where *dxx_install* is the directory where you installed DB2 XML Extender.

Insert the DTD into the DTD repository using one of the following methods:

- Enter the following SQL INSERT command, all on the same DB2 command line:

```
DB2 CONNECT TO SALES_DB
INSERT into DB2XML.DTD_REF values
('dxx_install/samples/extenders/db2xml/dtd/getstart.dtd',
 DB2XML.XMLClobFromFile
('dxx_install/samples/extenders/db2xml/dtd/getstart.dtd'),
 0, 'user1',   'user1', 'user1')
```

- Run the following command file to insert the DTD:

```
getstart_insertDTD.cmd
```

**Creating a DAD file for the XML column:**

This section explains how you create a DAD file for the XML column. In the DAD file, you specify that the access and storage method you are using is XML column. In the DAD file you define the side tables and columns for indexing.

In the following steps, elements in the DAD are referred to as *tags* and the elements of your XML document structure are referred to as *elements*. To create a DAD file for use with XML column:

1. Open a text editor and name the file `getstart_xcolumn.dad`
2. Create the DAD header, with the XML and the DOCTYPE declarations.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "/dxx_install/samples/DB2XML/dtd/dad.dtd ">
```

The DAD file is an XML document. You must include XML declarations. XML tags are case sensitive.

3. Insert start and end (<DAD> and</DAD>) tags for the document. All other tags are located inside these tags.

```
<DAD>
.
.
.
</DAD>
```

4. Insert start and end (<DTDID> and</DTDID>) tags with a DTD ID to specify a DTD if the document will be validated:

```
<dtdid>dxx_install/samples/extenders/db2xml/dtd/getstart.dtd</dtdid>
```

Verify that this string matches the value used as the first parameter value when you insert the DTD in the DTD repository table. For example, the path that you use for the DTDID might be different from the string you inserted in the DTD reference table if you are working on a different system drive.

5. Insert start and end (<validation> and </validation>) tags and a keyword YES or NO to indicate whether you want XML Extender to validate the XML document structure using the DTD that you inserted into the DTD reference table. For example:

```
<validation>YES</validation>
```

The value of <validation> can be in mixed case.

6. Insert start and end (<Xcolumn> and</Xcolumn>) tags to specify that the storage method is XML column.
7. Create side tables. For each side table that you want to create:
   a. Insert start and end (<table> and </table>) tags for each side table that is to be generated, and specify the name of the side table in double quotation marks using the ″name=″ attribute as shown here:

```
<Xcolumn>
<table name="order_side_tab">
</table>
<table name="part_side_tab">
</table>
<table name="ship_side_tab">
</table>
</Xcolumn>
```

   b. Inside the table tags, insert a <column> tag for each column that you want the side table to contain. Each column has four attributes: name, type, path, and multi_occurrence:

      **name** Specifies the name of the column that is created in the side table.

      **type** Indicates the data type in the side table for each indexed element or attribute.

      **path** Specifies the location path in the XML document for each element or attribute to be indexed.

      **multi_occurrence**
      Indicates whether the element or attribute referred to by the path attribute can occur more than once in the XML document. The possible values for *multi_occurrence* are *YES* or *NO*. If the value is

*NO*, then you can mention more than one column tag in the side table. If the value is *YES*, you can mention only one column in the side table.

```
<Xcolumn>
<table name="order_side_tab">
  <column name="order_key"
          type="integer"
          path="/Order/@Key"
          multi_occurrence="NO"/>
  <column name="customer"
          type="varchar(50)"
          path="/Order/Customer/Name"
          multi_occurrence="NO"/>

</table>
<table name="part_side_tab">
  <column name="price"
          type="decimal(10,2)"
          path="/Order/Part/ExtendedPrice"
          multi_occurrence="YES"/>
</table>
<table name="ship_side_tab">
  <column name="date"
          type="DATE"
          path="/Order/Part/Shipment/ShipDate"
          multi_occurrence="YES"/>
</table>
</Xcolumn>
```

8. Ensure that you have the necessary end tags:
   - A closing </Xcolumn> tag after the last </table> tag
   - A closing </DAD> tag after the </Xcolumn> tag
9. Save the file with the following name:

   ```
   getstart_xcolumn.dad
   ```

You can compare the file that you just created with the sample file, `dxx_install/samples/extenders/db2xml/dad/getstart_xcolumn.dad` . This file is a working copy of the DAD file required to enable the XML column and create the side tables. The sample files contain references to files that use absolute path names. Check the sample files and change these values for your directory paths.

**Creating the SALES_TAB table:**

In this section you create the SALES_TAB table. Initially, it has two columns with the sale information for the order. The SALES_TAB table is described in Table 1 on page 7.

To create the table:

Enter the following CREATE TABLE statement using one of the following methods:

- Enter the following DB2 database commands:

  ```
  DB2 CONNECT TO SALES_DB

  DB2 CREATE TABLE SALES_TAB(INVOICE_NUM CHAR(6)
          NOT NULL PRIMARY KEY,
          SALES_PERSON VARCHAR(20))
  ```

- Run the following command file to create the table:

  ```
  getstart_createTabCol.cmd
  ```

**Adding the column of XML type:**

Add a new column to the SALES_TAB table. This column will contain the intact XML document that you generated earlier and must be of an XML UDT. The XML Extender provides multiple data types. In this lesson, you will store the document as XMLVARCHAR.

To add the column of XML type:

Run the SQL ALTER TABLE statement using one of the following methods:

- Enter the following SQL statement:

  ```
  DB2 ALTER TABLE SALES_TAB ADD ORDER DB2XML.XMLVARCHAR
  ```

- Run the following command file to alter the table:

  ```
  getstart_alterTabCol.cmd
  ```

**Enabling the XML column:**

After you create the column of XML type, you enable it for XML Extender. When you enable the column, XML Extender reads the DAD file and creates the side tables. Before enabling the column, you must:

- Determine whether you want to create a default view of the XML column, which contains the XML document joined with the side-table columns. You can specify the default view when you enable the XML column. In this lesson, you will specify a view with the `-v` parameter.
- Determine whether you want to specify a primary key as the *ROOT ID*, the column name of the primary key in the application table and a unique identifier that associates all side tables with the application table. If you do not specify a primary key, XML Extender adds the DXXROOT_ID column to the application table, and to the side tables.

  The ROOT_ID column is used as key to tie the application and side tables together, which allows the XML Extender to automatically update the side tables if the XML document is updated. In this lesson, you will pass the primary key column (INVOICE_NUM) with the `-r` parameter. XML Extender then uses the specified column as the ROOT_ID and add the column to the side tables.
- Determine whether you want to specify a table space or use the default table space. In this lesson, you will use the default table space.

To enable the column for XML:

Run the **dxxadm enable_column** command, using one of the following methods:

**Command line:**

- Enter the following command:

  ```
  dxxadm enable_column SALES_DB SALES_TAB ORDER getstart_xcolumn.dad
          -v SALES_ORDER_VIEW -r INVOICE_NUM
  ```

- Run the following command file to enable the column:

  ```
  getstart_enableCol.cmd
  ```

The XML Extender creates the side tables with the INVOICE_NUM column and creates the default view.

**Important:** Do not modify the side tables in any way. Updates to the side tables should only be made through updates to the XML document itself. The XML Extender will automatically update the side tables when you update the XML document in the XML column.

**Viewing the column and side tables:**

When you enabled the XML column, you created a view of the XML column and side tables. You can use this view when working with the XML column.

To view the XML column and side-table columns:

Enter the following SQL SELECT statement from the command line:

```
SELECT * FROM SALES_ORDER_VIEW
```

The view shows the columns in the side tables, as specified in the `getstart_xcolumn.dad` file.

**Indexing side tables for structural search:**

Creating indexes on side tables allows you to do fast structural searches of the XML document. In this section, you create indexes on key columns in the side tables that were created when you enabled the XML column, ORDER. The service department has specified which columns their employees are likely to query most often. Table 3 describes these columns that you will index.

*Table 3. Side-table columns to be indexed*

| Column | Side table |
|---|---|
| ORDER_KEY | ORDER_SIDE_TAB |
| CUSTOMER | ORDER_SIDE_TAB |
| PRICE | PART_SIDE_TAB |
| DATE | SHIP_SIDE_TAB |

To index the side tables:

Run the following CREATE INDEX SQL commands using one of the following methods:

**Command line:**

- Enter the following commands:

```
DB2 CREATE INDEX KEY_IDX
      ON ORDER_SIDE_TAB(ORDER_KEY)

DB2 CREATE INDEX CUSTOMER_IDX
      ON ORDER_SIDE_TAB(CUSTOMER)

DB2 CREATE INDEX PRICE_IDX
      ON PART_SIDE_TAB(PRICE)

DB2 CREATE INDEX DATE_IDX
      ON SHIP_SIDE_TAB(DATE)
```

- Run the following command file to create the indexes:

```
getstart_createIndex.cmd
```

**Storing the XML document:**

Now that you enabled a column that can contain an XML document and indexed the side tables, you can store the document using the functions that the XML Extender provides. When storing data in an XML column, you either use default

casting functions or XML Extender UDFs. Because you will be storing an object of the base type VARCHAR in a column of the XML UDT XMLVARCHAR, you will use the default casting function.

To store the XML document:

1. Open the XML document `dxx_install/samples/extenders/db2xml/xml/getstart.xml`. Ensure that the file path in the DOCTYPE matches the DTD ID specified in the DAD and when inserting the DTD in the DTD repository. You can verify they match by querying the DB2XML.DTD_REF table and by checking the DTDID element in the DAD file. If you are using a different drive and directory structure than the default, you need to change the path in the DOCTYPE declaration to match your directory structure.
2. Run the SQL INSERT command, using one of the following methods:
   - Enter the following SQL INSERT command:

     ```
     DB2 INSERT INTO SALES_TAB (INVOICE_NUM, SALES_PERSON, ORDER) VALUES
     ('123456', 'Sriram Srinivasan', DB2XML.XMLVarcharFromFile
     ('dxx_install/samples/extenders/db2xml/
     /xml/getstart.xml'))
     ```
   - Run the following command file to store the document:

     ```
     getstart_insertXML.cmd
     ```

Verify that the tables have been updated. Run the following SELECT statements for the tables from the command line.

```
SELECT * FROM SALES_TAB
SELECT * FROM PART_SIDE_TAB
SELECT * FROM ORDER_SIDE_TAB
SELECT * FROM SHIP_SIDE_TAB
```

**Querying the XML document:**

You can search the XML document with a direct query against the side tables. In this step, you will search for all orders that have a price over 2500.00.

To query the side tables:

Run the SQL SELECT statement, using one of the following methods:

- Enter the following SQL SELECT statement:

  ```
  DB2 "SELECT DISTINCT SALES_PERSON FROM SALES_TAB S,
     PART_SIDE_TAB P  WHERE PRICE > 2500.00
      AND S.INVOICE_NUM=P.INVOICE_NUM"
  ```
- Run the following command file to search the document:

  ```
  getstart_queryCol.cmd
  ```

The result set should show the names of the salespeople who sold an item that had a price greater than 2500.00. For Example:

```
SALES_PERSON

--------------------

Sriram Srinivasan
```

You have completed the getting started tutorial for storing XML documents in DB2 database tables.

**Related concepts:**

- "XML Extender tutorial lessons" on page 6
- "Introduction to XML Extender" on page 3
- "Lesson: Composing an XML document"

## Lesson: Composing an XML document

This lesson teaches you how to compose an XML document from existing DB2® data.

**The scenario:**

You have the task of taking information in an existing purchase order database, SALES_DB, and extracting requested information from it to be stored in XML documents. The service department will then use these XML documents when working with customer requests and complaints. The service department has requested specific data to be included and has provided a recommended structure for the XML documents.

Using existing data, you will compose an XML document, `getstart.xml`, from data in the tables in the SALES_DB database.

To compose an XML document, you will plan and create a DAD file that maps columns from the related tables to an XML document structure that provides a purchase order record. You will create an XML collection and associate the tables with an XML structure and a DTD. You use this DTD to define the structure of the XML document. You can also use it to validate the composed XML document in your applications.

The existing database data for the XML document is described in the following tables. The column names with an asterisk are columns that the service department has requested in the XML document structure.

**ORDER_TAB**

| Column name | Data type |
|---|---|
| ORDER_KEY * | INTEGER |
| CUSTOMER | VARCHAR(16) |
| CUSTOMER_NAME * | VARCHAR(16) |
| CUSTOMER_EMAIL * | VARCHAR(16) |

**PART_TAB**

| Column name | Data type |
|---|---|
| PART_KEY * | INTEGER |
| COLOR * | CHAR(6) |
| QUANTITY * | INTEGER |
| PRICE * | DECIMAL(10,2) |
| TAX * | REAL |
| ORDER_KEY | INTEGER |

**SHIP_TAB**

| Column name | Data type |
|---|---|
| DATE * | DATE |
| MODE * | CHAR(6) |
| COMMENT | VARCHAR(128) |
| PART_KEY | INTEGER |

**Planning:**

Before you use the XML Extender to compose your documents, you need to determine the structure of the XML document and how it corresponds to the structure of your database data. This section provides an overview of the XML document structure that the service department requested, and the DTD that you will use to define the structure of the XML document. This section shows how the data in the database columns map to data in the composed XML document.

**Determining the document structure:**

The XML document structure is driven by the requirements of the people who will use the resulting XML document. In this example, those people are in the service department. The service department needs information about the order. For example, order number, customer, part, and shipping information. You can derive all of the data from the order_tab, part_tab and ship_tab tables.

After you design the document structure, you create a DTD to describe the structure of the XML document. This lesson provides a DTD for you. Using the rules of the DTD and the hierarchical structure of the XML document, you can create a hierarchical map of your data, as shown in Figure 1 on page 19.

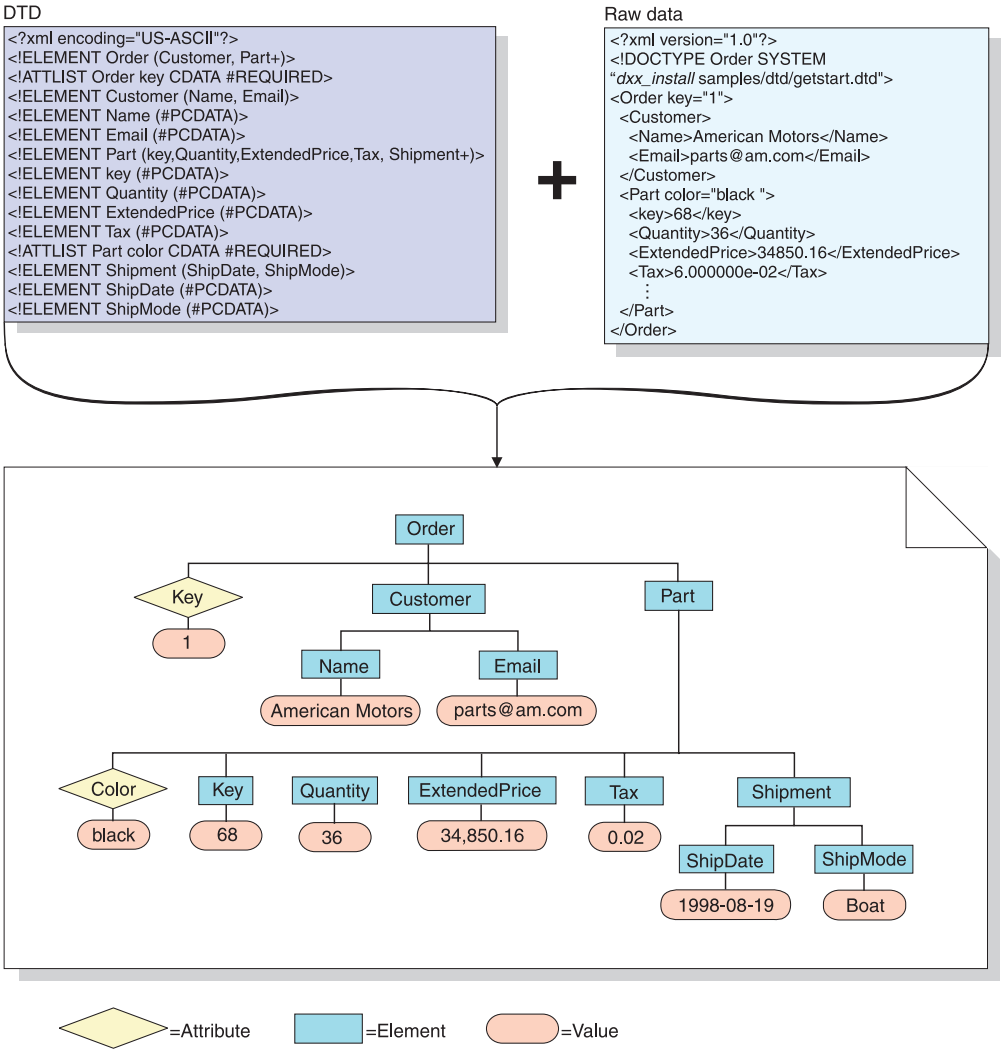



*Figure 1. The hierarchical structure of the DTD and XML document*

**Mapping the XML document and database relationship:**

After you design the structure and create the DTD, you need to show how the structure of the document relates to the DB2 database tables that you will use to populate the elements and attributes. You can map the hierarchical structure to specific columns in the relational tables, as shown in Figure 2 on page 20.

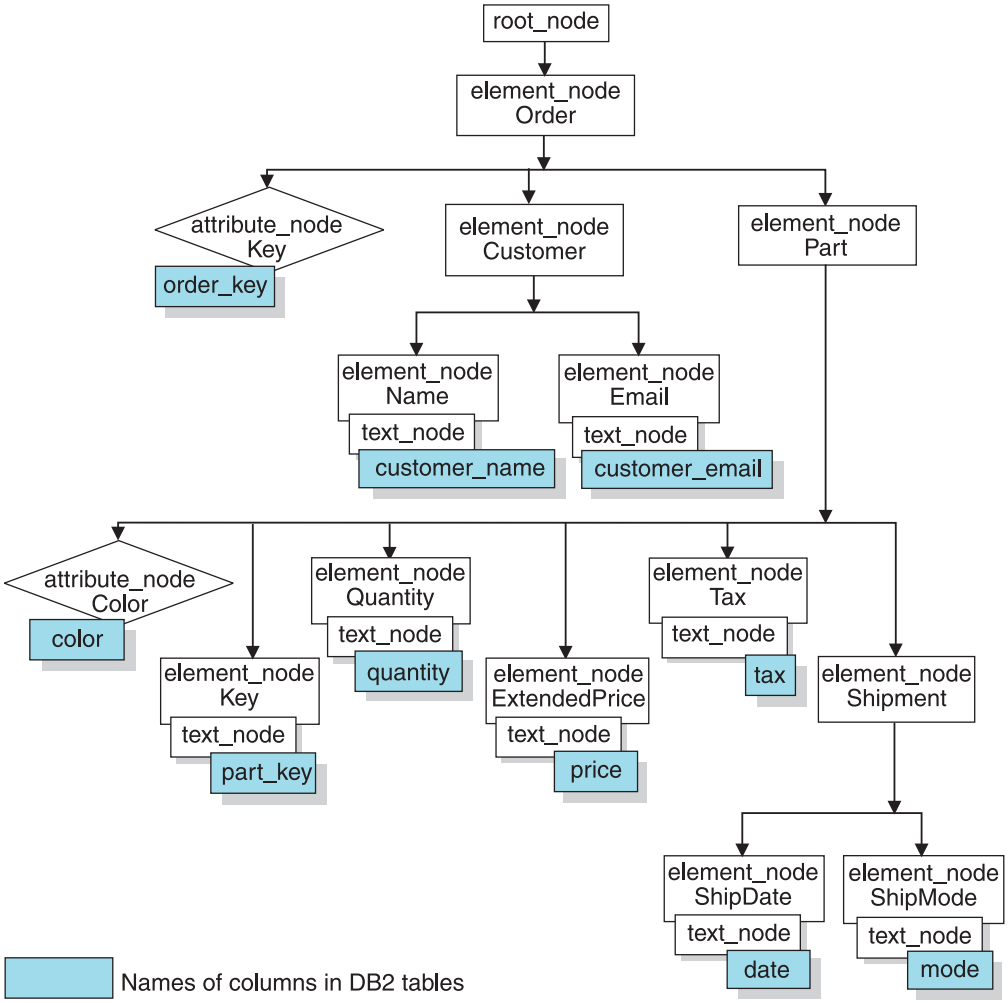



*Figure 2. XML document mapped to relational table columns*

This figure uses nodes to show elements, attributes, and text within the XML document structure. These nodes are used in the DAD file and are explained more fully in later steps.

Use this relationship description to create a DAD file that define the relationship between the relational data and the XML document structure.

To create the XML collection DAD file, you need to understand how the XML document corresponds to the database structure, as described in Figure 2, so that you can describe from what tables and columns the XML document structure derives data for elements and attributes. You will use this information to create the DAD file for the XML collection.

**The scripts and samples:**

This lesson provides a set of scripts for you to use to set up your environment. These scripts are in the *dxx_install*/samples/extenders/db2xml/xml directory (where *dxx_install* is the directory where you installed XML Extender files)

The scripts are:

**getstart_db.cmd**
Creates the database and populates four tables.

**getstart_prep.cmd**
Binds the database with the XML Extender stored procedures.

**getstart_stp.cmd**
Runs the stored procedure to compose the XML collection.

**getstart_exportXML.cmd**
Exports the XML document from the database for use in an application.

**getstart_clean.cmd**
Cleans up the tutorial environment.

**Setting up the lesson environment:**

In this section, you prepare the database for use with XML Extender. You will:

1. Create the database.
2. Enable the database.

**Creating the database:**

In this section, you use a command to set up the database. This command creates the SALES_DB database, connects to it, creates the tables to hold data, and then inserts the data.

**Important:** If you completed the XML column lesson and did not clean up your environment, you might be able to skip this step. Check to see if you have a SALES_DB database.

To create the database:

1. Change to the *dxx_install*`/samples/extenders/db2xml/xml/cmd` directory, where *dxx_install* is the directory where you installed the XML Extender files. The sample files contain references to files that use absolute path names. Check the sample files and change these values for your directory paths.
2. On Windows, type the following command into a Command Prompt window to start the DB2 command line processor:

   ```
   DB2CMD
   ```

3. Enter the following command:

   ```
   getstart_db.cmd
   ```

**Enabling the database:**

To generate an XML document from relational data in the database, you must enable the database for XML Extender. When you enable a database for XML, the XML Extender:

- Creates the user-defined types (UDTs), user-defined functions (UDFs), and stored procedures.
- Creates and populates control tables with the necessary metadata that the XML Extender requires.
- Creates the DB2XML schema and assigns the necessary privileges.

**Important:** If you completed the XML column lesson and did not clean up your environment, you might be able skip this step.

Run the following script to enable the SALES_DB database:

```
getstart_prep.cmd
```

This script binds the database to the XML Extender stored procedures. It also runs the **dxxadm** command option that enables the SALES_DB database.

**Creating the DAD file for the XML collection:**

Because the data already exists in multiple tables, you will create an XML collection, which associates the tables with the XML document. You define the collection by creating a DAD file.

In this section, you create the mapping scheme in the DAD file that specifies the relationship between the tables and the structure of the XML document.

In the following steps, elements in the DAD are referred to as *tags* and the elements of your XML document structure are referred to as *elements*. A sample of a DAD file similar to the one you will create is in *dxx_install*/samples/extenders/db2xml/dad/getstart_xcollection.dad.

It has some minor differences from the file generated in the following steps. If you use it for the lesson, note that the file paths might be different than in your environment and you might need to update the sample file.

To create the DAD file for composing an XML document:

1. From the dxx_install/samples/extenders/db2xml/xml directory, open a text editor and create a file called getstart_xcollection.dad.
2. Create the DAD header, using the following text:

   ```
   <?xml version="1.0"?>
   <!DOCTYPE DAD SYSTEM "dxx_install/samples/extenders/db2xml/dtd/dad.dtd">
   ```

   Change dxx_install to the directory where DB2 XML Extender was installed.
3. Insert the <DAD></DAD> tags. All other tags are located inside these tags.
4. Specify <validation> </validation> tags to indicate whether the XML Extender validates the XML document structure. The XML document is validated against a DTD in the DTD repository table. This lesson does not require a DTD and the value is **NO**.

   ```
   <validation>NO</validation>
   ```

   The value of the <validation> tags can be mixed case.
5. Use the <Xcollection></Xcollection> tags to define the access and storage method as XML collection. The access and storage methods define that the XML data is stored in a collection of DB2 database tables.

   ```
   <Xcollection>
   </Xcollection>
   ```
6. After the <Xcollection> tag, provide an SQL statement to specify the tables and columns used for the XML collection. This method is called SQL mapping and is one of two ways to map relational data to the XML document structure. Enter the following statement:

   ```
   <Xcollection>
   <SQL_stmt>
      SELECT o.order_key, customer_name, customer_email, p.part_key, color,
      quantity, price, tax, ship_id, date, mode from order_tab o, part_tab p,
      table (select substr(char(timestamep(generate_unique()),16)
   as ship_id, date, mode, part_key from ship_tab)  s
   ```

```
        WHERE o.order_key = 1 and
              p.price > 20000 and
              p.order_key = o.order_key and
              s.part_key = p.part_key
        ORDER BY order_key, part_key, ship_id
</SQL_stmt>
</Xcollection>
```

This SQL statement uses the following guidelines when using SQL mapping. See Figure 2 on page 20 for the document structure.

- Columns are specified in top-down order, by the hierarchy of the XML document structure. For example, the columns for the order and customer elements are first, those for the part element are second, and those for the shipment are third.
- The columns for a repeating section, or nonrepeating section, of the template that requires data from the database are grouped together. Each group has an object ID column: ORDER_KEY, PART_KEY, and SHIP_ID.
- The object ID column is the first column in each group. For example, O.ORDER_KEY precedes the columns related to the key attribute and p.PART_KEY precedes the columns for the Part element.
- The SHIP_TAB table does not have a single key conditional column, and therefore, the generate_unique DB2 built-in function is used to generate the SHIP_ID column.
- The object ID columns are then listed in top-down order in an ORDER BY statement. The columns in ORDER BY are not qualified by any schema and table name, and they match the column names in the SELECT clause.

7. Add the following prolog information to be used in the composed XML document. If you need to specify an encoding value for internationalization, add the ENCODING attribute and value.

   ```
   <prolog>?xml version="1.0"?</prolog>
   ```

   This exact text is required for all DAD files.

8. Add the <doctype></doctype> tags to be used in the XML document you are composing. The <doctype> tag contains the path to the DTD stored on the client.

   ```
   <doctype>!DOCTYPE Order SYSTEM
   "dxx_install/samples/extenders/db2xml/dtd/getstart.dtd"</doctype>
   ```

9. Define the root element of the XML document using the <root_node></root_node> tags. Inside the root_node, specify the elements and attributes that make up the XML document.
10. Map the XML document structure to the DB2 database relational table structure using the following three types of nodes:

    **element_node**
    Specifies the element in the XML document. Element_nodes can have child element_nodes.

    **attribute_node**
    Specifies the attribute of an element in the XML document.

    **text_node**
    Specifies the text content of the element and the column data in a relational table for bottom-level element_nodes.

    Figure 2 on page 20 shows the hierarchical structure of the XML document and the DB2 database table columns, and indicates what kinds of nodes are used. The shaded boxes indicate the DB2 database table column names from which the data will be extracted to compose the XML document.

To add each type of node, one type at a time:

a. Define an <element_node> tag for each element in the XML document.

```
<root_node>
<element_node name="Order">
  <element_node name="Customer">
     <element_node name="Name">
     </element_node>
     <element_node name="Email">
     </element_node>
  </element_node>
  <element_node name="Part">
     <element_node name="key">
     </element_node>
     <element_node name="Quantity">
     </element_node>
     <element_node name="ExtendedPrice">
     </element_node>
     <element_node name="Tax">
     </element_node>
     <element_node name="Shipment" multi_occurrence="YES">
        <element_node name="ShipDate">
        </element_node>
        <element_node name="ShipMode">
        </element_node>
     </element_node> <!-- end Shipment -->
  </element_node> <!-- end Part -->
</element_node> <!-- end Order -->
</root_node>
```

The <Shipment> child element has an attribute of multi_occurrence=YES. This attribute is used for elements without an attribute, that are repeated in the document. The <Part> element does not use the multi-occurrence attribute because it has an attribute of color, which makes it unique.

b. Define an <attribute_node> tag for each attribute in your XML document. These attributes are nested in the appropriate element_node. The added attribute_nodes are highlighted in bold:

```
<root_node>
<element_node name="Order">
  <attribute_node name="key">
  </attribute_node>
  <element_node name="Customer">
     <element_node name="Name">
     </element_node>
     <element_node names"Email">
     </element_node>
  </element_node>
  <element_node name="Part">
    <attribute_node name="color">
    </attribute_node>
    <element_node name="key">
    </element_node>
    <element_node name="Quantity">
    </element_node>

...

  </element_node> <!-- end Part -->
</element_node> <!-- end Order -->
</root_node>
```

c. For each bottom-level element_node, define <text_node> tags to indicate that the XML element contains character data to be extracted from DB2 database when the document is composed.

```
<root_node>
<element_node name="Order">
  <attribute_node name="key">
  </attribute_node>
  <element_node name="Customer">
    <element_node name="Name">
      <text_node>
      </text_node>
    </element_node>
    <element_node name="Email">
      <text_node>
      </text_node>
    </element_node>
  </element_node>
  <element_node name="Part">
    <attribute_node name="color">
    </attribute_node>
    <element_node name="key">
       <text_node>
       </text_node>
    </element_node>
    <element_node name="Quantity">
      <text_node>
      </text_node>
    </element_node>
    <element_node name="ExtendedPrice">
      <text_node>
      </text_node>
    </element_node>
    <element_node name="Tax">
      <text_node>
      </text_node>
    </element_node>
    <element_node name="Shipment" multi_occurrence="YES">
      <element_node name="ShipDate">
        <text_node>
        </text_node>
      </element_node>
      <element_node name="ShipMode">
        <text_node>
        </text_node>
      </element_node>
    </element_node> <!-- end Shipment -->
  </element_node> <!-- end Part -->
</element_node> <!-- end Order -->
</root_node>
```

d. For each bottom-level element_node, define a <column> tag. These tags specify from which column to extract data when composing the XML document and are typically inside the <attribute_node> or the <text_node> tags. The columns defined in the <column> tag must be in the <SQL_stmt> SELECT clause.

```
<root_node>
<element_node name="Order">
  <attribute_node name="key">
    <column name="order_key"/>
  </attribute_node>
  <element_node name="Customer">
    <element_node name="Name">
      <text_node>
        <column name="customer_name"/>
      </text_node>
    </element_node>
    <element_node name="Email">
      <text_node>
        <column name="customer_email"/>
```

```
   </text_node>
  </element_node>
</element_node>
<element_node name="Part">
  <attribute_node name="color">
     <column name="color"/>
  </attribute_node>
  <element_node name="key">
    <text_node>
      <column name="part_key"/>
    </text_node>
  <element_node name="Quantity">
    <text_node>
      <column name="quantity"/>
    </text_node>
  </element_node>
  <element_node name="ExtendedPrice">
    <text_node>
      <column name="price"/>
    </text_node>
  </element_node>
  <element_node name="Tax">
    <text_node>
      <column name="tax"/>
    </text_node>
  </element_node>
  <element_node name="Shipment" multi_occurrence="YES">
    <element_node name="ShipDate">
      <text_node>
        <column name="date"/>
      </text_node>
    </element_node>
    <element_node name="ShipMode">
      <text_node>
        <column name="mode"/>
      </text_node>
    </element_node>
  </element_node> <!-- end Shipment -->
 </element_node> <!-- end Part -->
</element_node> <!-- end Order -->
</root_node>
```

11. Ensure that you have the necessary end tags:
    - An ending </root_node> tag after the last </element_node> tag
    - An ending </Xcollection> tag after the </root_node> tag
    - An ending </DAD> tag after the </Xcollection> tag
12. Save the file as `getstart_xcollection.dad`.

You can compare the file that you created with the sample file *dxx_install*`/samples/extenders/db2xml/dad/getstart_xcollection.dad` . This file is a working copy of the DAD file required to compose the XML document. The sample file contains location paths and file path names that might need to be changed to match your environment to be run successfully.

In your application, if you will use an XML collection frequently to compose documents, you can define a collection name by enabling the collection. Enabling the collection registers it in the XML_USAGE table and helps improve performance when you specify the collection name (rather than the DAD file name) when running stored procedures. In these lessons, you do not enable the collection.

**Composing the XML document:**

In this step, you use the dxxGenXML() stored procedure to compose the XML document specified by the DAD file. This stored procedure returns the document as an XMLVARCHAR UDT.

To compose the XML document:

1. Enter the following command:

   ```
   getstart_stp.cmd
   ```

   The stored procedure composes the XML document and stores it in the RESULT_TAB table.

   If you are running XML Extender on a partitioned DB2 Enterprise Server Edition environment, check that you either created a result table with a qualified partitioning key, or that you created the result table in a table space that is in a node group with a single node.

   You can see samples of stored procedures that can be used in this step in the following files:

   - *dxx_install*/samples/extenders/db2xml/c/tests2x.sqc shows how to call the stored procedure using embedded SQL and generates the tests2x executable file, which is used by the getstart_stp.cmd.
   - *dxx_install*/samples/extenders/db2xml/cli/sql2xml.c shows how to call the stored procedure using the CLI.
2. Export the XML document from the table to a file using one of the following methods to call the XML Extender retrieval function, Content():
   - Enter the following commands:

     ```
     DB2 CONNECT TO SALES_DB

     DB2 SELECT DB2XML.Content(DB2XML.xmlVarchar(doc),
        'dxx_install/samples/extenders/db2xml/cmd/xml/getstart.xml
          ') FROM RESULT_TAB
     ```

   - Run the following command file to export the file:

     ```
     getstart_exportXML.cmd
     ```

**Tip:** This step teaches you how to generate one or more composed XML documents using DB2 database stored procedure's result set feature. Using a result set allows you to fetch multiple rows to generate more than one document. When you generate each document, you can export it to a file. This method is the simplest way to demonstrate using result sets. For more efficient ways of fetching data, see the CLI examples in *dxx_install*/samples/extenders/db2xml/cli.

**Transforming an XML document into an HTML file:**

To show the data from the XML document in a browser, you must transform the XML document into an HTML file by using a stylesheet and the XSLTransformToFile function.

Use the following steps to transform to an HTML file:

1. Generate a stylesheet using a text editor and name it getstart.xsl:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
    <xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
     <html>
     <head/>
     <body>
   ```

```
...

    </body>
   </html>
 </xsl:template>
 </xsl:stylesheet>
```

An example of this completed file can be found in the following directory:

*dxx_install*/samples/extenders/db2xml/xslt/getstart.xsl

2. For each element, create a tag using the following format:

```
<xsl:for-each select="xxxxxx">
```

This tag will be used for transforming instructions. Create a tag for each element of the hierarchy of the XML document. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
 <xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="/">
  <html>
  <head/>
  <body>

     <xsl:for-each select="Order">

    <xsl:for-each select="Customer">
          <xsl:for-each select="Name | Email">
          </xsl:for-each>
       </xsl:for-each>
<xsl:for-each select="Part">
        <xsl:for-each select="key | Quantity | ExtendedPrice | Tax">
      </xsl:for-each>

           <xsl:for-each select="Shipment">
         <xsl:for-each select="ShipDate | ShipMode">
         </xsl:for-each>
             </xsl:for-each>
         </xsl:for-each>
    </body>
   </html>
 </xsl:template>
 </xsl:stylesheet>
```

3. To format the HTML file, use a list that shows the hierarchy of the XML elements to make the data more readable. Create some additional text elements to describe the data. For example, your stylesheet file might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
 <xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="/">
  <html>
  <head/>
  <body>

     <ol style="list-style:decimal outside">
      <xsl:for-each select="Order">
       <li> Orderkey : <xsl:value-of-select="@Key"/ <br/>

       <xsl:for-each select="Customer">
        <b>Customer</b><br/>
          <xsl:for-each select="Name | Email">
      <xsl:value-of select="name()"/>
```

```
    <xsl:text> : </xsl:text>
         <xsl:value-of select="."/>
         <xsl:text>, </xsl:text>
        </xsl:for-each>
     </xsl:for-each>

       <br/><br/>
   <ol type="A">
       <xsl:for-each select="Part">
        <li><b>Parts</b><br/>
     Color : <xsl:value-of select="@color"/>
         <xsl:text>, </xsl:text>

      <xsl:for-each select="key | Quantity | ExtendedPrice | Tax">
<xsl:value-of select="name()"/>
          <xsl:text> : </xsl:text>
          <xsl:value-of select="."/>
<xsl:text>, </xsl:text>
    </xsl:for-each>

         <br/><br/>
    <ol type="a">
         <xsl:for-each select="Shipment">
      <li><b>Shipment</b><br/>
      <xsl:for-each select="ShipDate | ShipMode">
  <xsl:value-of select="name()"/>
           <xsl:text> : </xsl:text>
           <xsl:value-of select="."/>
  <xsl:text>, </xsl:text>
      </xsl:for-each>
     </li>
         </xsl:for-each>
        </ol><br/>
        </li>
        </xsl:for-each>
      </ol>
     </li>
      </xsl:for-each>
     </ol>
   </body>
  </html>
 </xsl:template>
 </xsl:stylesheet>
```

4. Use Xpath to edit the <xsl:value-of select=″xxx″> tags with data from the XML document.

   The element tags are <xsl:value-of select″.″>, where the period (″.″) is used to get data from normal elements.

   The attribute tags are <xsl:value-of select=″@attributname″> , where the ampersand (@) that is added by the attribute name will extract the value of the attribute. You can use the <xsl:value-of select=″name()″> to get the name of the XML tag.

```
<?xml version="1.0" encoding="UTF-8"?>
 <xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="/">
  <html>
  <head/>
  <body>

     <ol style="list-style:decimal outside">
      <xsl:for-each select="Order">
       <li> Orderkey : <xsl:value-of-select="@Key"/ <br/>
```

```
     <xsl:for-each select="Customer">
      <b>Customer</b><br/>
        <xsl:for-each select="Name | Email">
    <xsl:value-of select="name()"/>
    <xsl:text> : </xsl:text>
         <xsl:value-of select="."/>
         <xsl:text>, </xsl:text>
        </xsl:for-each>
     </xsl:for-each>

        <br/><br/>
  <ol type="A">
      <xsl:for-each select="Part">
       <li><b>Parts</b><br/>
    Color : <xsl:value-of select="@color"/>
         <xsl:text>, </xsl:text>

       <xsl:for-each select="key | Quantity | ExtendedPrice | Tax">
<xsl:value-of select="name()"/>
          <xsl:text> : </xsl:text>
          <xsl:value-of select="."/>
<xsl:text>, </xsl:text>
    </xsl:for-each>

         <br/><br/>
    <ol type="a">
         <xsl:for-each select="Shipment">
      <li><b>Shipment</b><br/>
      <xsl:for-each select="ShipDate | ShipMode">
  <xsl:value-of select="name()"/>
            <xsl:text> : </xsl:text>
            <xsl:value-of select="."/>
  <xsl:text>, </xsl:text>
      </xsl:for-each>
     </li>
         </xsl:for-each>
        </ol><br/>
        </li>
        </xsl:for-each>
     </ol>
     </li>
      </xsl:for-each>
     </ol>
   </body>
  </html>
 </xsl:template>
 </xsl:stylesheet>
```

5. Save the stylesheet.
6. Create the HTML file in one of the following ways:
   - Use the function XSLTransformToFile:

     ```
     SELECT DB2XML.XSLTransformToFile( CAST(doc AS CLOB(4k)),
         'dxx_install\samples\xslt\getstart.xsl',
       'dxx_install\samples\html\getstart.html')
      FROM RESULT_TAB
     ```

   - Use the following command:

     ```
     getstart_xslt.cmd
     ```

The output file can be written only to a file system that is accessible to the DB2 database server.

**Cleaning up the tutorial environment:**

If you want to clean up the lesson environment, you can run one of the provided scripts or enter the commands from the command line to:

- Disable the XML column, ORDER.
- Drop tables created in the lessons.
- Delete the DTD from the DTD repository table.

The SALES_DB database is not disabled or dropped; the database is still available for use with XML Extender. You might receive error messages if you have not completed both lessons in this section. You can ignore these errors.

To clean up the tutorial environment:

Run the cleanup command file, using one of the following methods:

- Enter the following command:

  ```
  getstart_clean.cmd
  ```

- If you want to disable the database, you can run the following XML Extender command from the command line:

  ```
  dxxadm disable_db SALES_DB
  ```

  This command drops the administration control tables DTD_REF and XML_USAGE, and removes the user-defined types and functions provided by XML Extender.

- If you want to drop the database, you can run the following command from the command line

  ```
  db2 drop database SALES_DB
  ```

  This command drops the SALES_DB.

**Related concepts:**

- "XML Extender tutorial lessons" on page 6
- "Introduction to XML Extender" on page 3
- "Lesson: Storing an XML document in an XML column" on page 7

# Part 2. Administration

This part describes how to perform administration tasks for the XML Extender.

© Copyright IBM Corp. 1999 - 2006

# Chapter 2. Administration

## Administration tools for XML Extender

The XML Extender administration tools help you enable your database and table columns for XML, and map XML data to DB2® relational structures. You must have DBADM authority to use XML Extender administration tools.

XML Extender provides the following tools to complete administration tasks:

- The XML Extender administration wizard provides a graphical user interface for administration tasks.
- The **dxxadm** command provides a command-line option for administration tasks.
- The XML Extender administration stored procedures allow you to invoke administration commands from a program.

## Preparing to administer XML Extender

To run XML Extender, you need to install the following software.

**Required software:** The XML Extender requires DB2® Database Version 9.1.

**Optional software:**

- For structural text search, the DB2 Database Net Search Extender Version 9.1, which is available with DB2 Database Version 9.1
- For XML Extender administration wizard:
  - DB2 Database Java Database Connectivity (JDBC)
  - SDK 1.1.7 or later, or JRE 1.1.1, which is available with the DB2 Control Center
  - JFC 1.1 with Swing 1.1, which is available with the DB2 Control Center

After you install XML Extender, you must complete the following tasks:

- Bind XML Extender to your DB2 database.

  You must bind XML Extender to each database. For an example see:

  *dxx_install*/samples/extenders/db2xml/cmd/getstart_prep.cmd
- View the set up instructions.
- Create a database for XML access.

## Migrating XML Extender from previous versions or releases

If you use an earlier version of DB2® XML Extender, you must migrate each database that is enabled for XML Extender before you use an existing XML-enabled database with XML Extender Version 9.1.

The XML Extender migration program makes the changes that are required for your existing data to work with DB2 Version 9.1.

© Copyright IBM Corp. 1999 - 2006

**Note:** dxxEnableColl is renamed to dxxEnableCollection, and dxxDisableColl is renamed to dxxDisableCollection. A stored procedure, db2xml.dxxDisableDB, has also been added.

When calling stored procedures, use a period (.) instead of an explanation point (!) in the procedure name. For example, use *db2xml.dxxEnableColumn* instead of *db2xml!dxxEnableColumn*.

**Procedure:**

To migrate an XML enabled database and XML enabled columns:

1. Install DB2 Version 9.1 and select the XML Extender component.
2. From the DB2 command line, enter:

```
db2 connect to database_name
db2 bind @dxxMigv.lst
dxxMigv database_name
```

Failing to do the migration step can cause problems and unpredictable results such as failure while disabling databases and inability to access new UDFs.

## XML Extender administration overview

The XML Extender provides three methods of administration: the XML Extender administration command, XML Extender stored procedures, and the XML Extender administration wizard.

- The administration command, **dxxadm**, provides options for the various administration tasks.
- Administration tasks can be executed by calling stored procedures for administration from a program.
- The XML Extender administration wizard guides you through the administration tasks. You can use it from a client workstation.

## XML Extender administration wizard

The XML Extender administration tasks consist of enabling or disabling databases and table columns for XML Extender, creating collections of table columns that are mapped into XML document trees, and importing DTDs into the database. You can use the XML Extender wizard to complete these administration tasks. Start the wizard with the `dxxadmin` command from a Command Prompt window.

After you install DB2 XML Extender, follow instructions in the readme file installed in the following path:

*dxx_install*`/db2xml/adm_gui/readme.htm`

## Access and storage methods

The XML Extender provides two access and storage methods to use DB2® as an XML repository: XML column and XML collection. You need to decide which of these methods best matches your application's needs for accessing and manipulating XML data.

**XML column**
Stores and retrieves entire XML documents as DB2 column data. The XML data is represented by an XML column.

**XML collection**
Decomposes XML documents into a collection of relational tables or composes XML documents from a collection of relational tables.

The nature of your application determines which access and storage method is most suitable, and how to structure your XML data.

You use the DAD file to associate XML data with DB2 tables through these two access and storage methods. Figure 3 shows how the DAD specifies the access and storage methods.

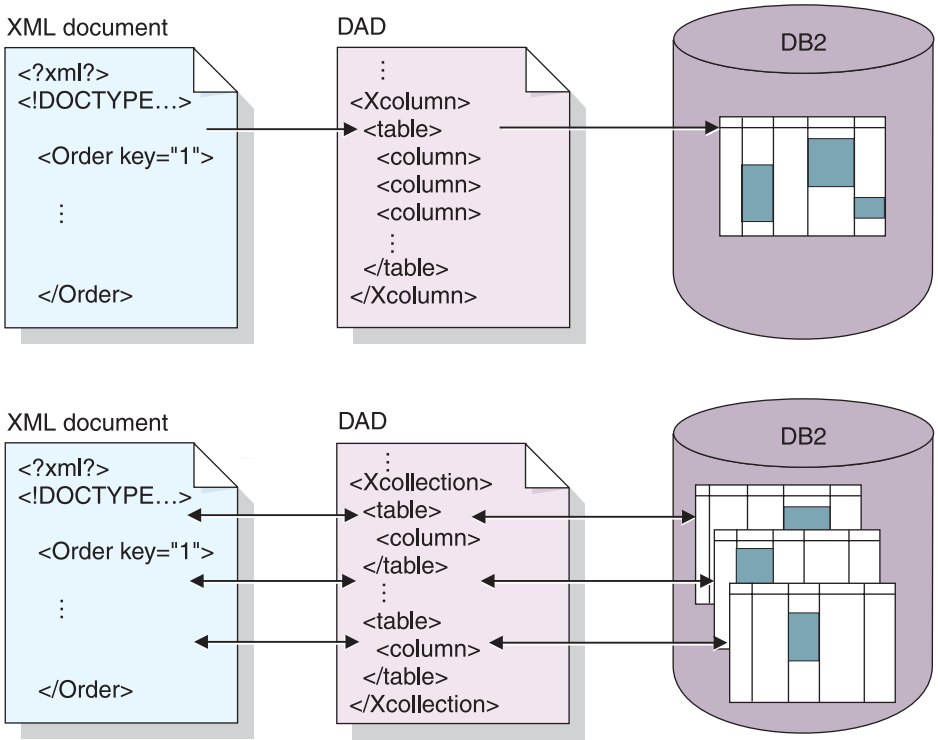



*Figure 3. The DAD file maps the XML document structure to a DB2 relational data structure and specifies the access and storage method.*

The DAD file defines the location of key files like the DTD, and specifies how the XML document structure relates to your DB2 data. Most important, it defines the access and storage method that you use in your application.

**Related concepts:**

- "When to use the XML collection method" on page 38
- "When to use the XML column method" on page 38

**Related reference:**

- "Storage functions in XML Extender overview" on page 132

## When to use the XML column method

Use XML columns in any of the following situations:

- The XML documents already exist or come from an external source and you prefer to store the documents in the native XML format. You want to store them in DB2® for integrity, archival, and auditing purposes.
- The XML documents are read frequently, but not updated.
- You want to use file name data types to store the XML documents (external to DB2 database) in the local or remote file system and use DB2 database for management and search operations.
- You need to perform range searches based on the values of XML elements or attributes, and you know what elements or attributes will frequently be used in the search arguments.
- The documents have elements with large text blocks and you want to use the DB2 Text Extender for structural text search while keeping the entire documents intact.

## When to use the XML collection method

Use XML collections in any of the following situations:

- You have data in your existing relational tables and you want to compose XML documents based on a certain DTD.
- You have XML documents that need to be stored with collections of data that map well to relational tables.
- You want to create different views of your relational data using different mapping schemes.
- You have XML documents that come from other data sources. You care about the data but not the tags, and want to store pure data in your database and you want the flexibility to decide whether to store the data in existing tables or in new tables.

## Planning for XML columns

Before you begin working with the XML Extender to store your documents, you need to understand the structure of the XML document so that you can determine how to index elements and attributes in the document. When planning how to index the document, you need to determine:

- The XML user-defined type in which you will store the XML document
- The XML elements and attributes that your application will frequently search, so that their content can be stored in side tables and indexed to improve performance
- Whether to validate XML documents in the column with a DTD
- The structure of the side tables and how they will be indexed

### XML data types for the XML columns

The XML Extender provides XML user defined types that you use to define a column to hold XML documents. These data types are described in Table 4 on page 39.

*Table 4. The XML Extender UDTs*

| User-defined type column | Source data type | Usage description |
|---|---|---|
| XMLVARCHAR | VARCHAR(*varchar_len*) | Stores an entire XML document as a VARCHAR data type within DB2®. Used for small documents (less than 3K) that are stored in DB2. |
| XMLCLOB | CLOB(*clob_len*) | Stores an entire XML document as a CLOB data type within DB2. Used for large documents (larger than 3K) that are stored in DB2. |
| XMLFILE | VARCHAR(512) | Stores the file name of an XML document in DB2, and stores the XML document in a file local to the DB2 server. Used for documents that are stored outside DB2. |

## Elements and attributes to index for XML columns

When you understand the XML document structure and the needs of your application, you can determine which elements and attributes will be searched or extracted most frequently, or those that will be the most expensive to query. The DAD file for an XML column can map the location paths of each element and attribute to relational tables (side tables) that contain these objects. The side tables are then indexed.

For example, Table 5 shows an example of types of data and location paths of elements and attributes from the Getting Started scenario for XML columns. The data was specified as information to be frequently searched and the location paths point to elements and attributes that contain the data. The DAD file can map these location paths to side tables.

*Table 5. Elements and attributes to be searched*

| Data | Location path |
|---|---|
| order key | /Order/@Key |
| customer | /Order/Customer/Name |
| price | /Order/Part/ExtendedPrice |
| shipping date | /Order/Part/Shipment/ShipDate |

## The DAD file for XML columns

For XML columns, the DAD file primarily specifies how documents that are stored in an XML column are to be indexed. The DAD file specifies a DTD to use for validating documents that are inserted into the XML column. The size of this file can be up to 2 GB.

The DAD file for XML columns provides a map of any XML data that is to be stored in side tables for indexing.

To specify the XML column access and storage method, you use the <Xcolumn> tag in the DAD file. The <Xcolumn> tag specifies that the XML data is to be stored and retrieved as entire XML documents in DB2 database columns that are enabled for XML data.

An XML-enabled column is of the XML Extender's UDT. Applications can include the column in any user table. You access the XML column data mainly through SQL statements and the XML Extender's UDFs.

**Related concepts:**

• "Planning side tables" on page 56

## Planning for XML collections

When planning for XML collections, you have different considerations for composing documents from DB2® data, decomposing XML document into relational data, or both. The following sections address planning issues for XML collections, and address composition and decomposition considerations.

### Validation

After you choose an access and storage method, you can determine whether to validate your data. You validate XML data using a DTD or a schema. Using a DTD or schema to validate ensures that the XML document is valid.

To validate using a DTD, you might need to have a DTD in the XML Extender repository.

**Important**: Make the decision whether to validate XML data before you insert XML data into DB2. The XML Extender does not validate data that is already inserted into DB2.

**Considerations:**

- You can use only one DTD for composition.
- You can use multiple schemas for composition.
- If you do not choose to validate a document, the DTD specified by the XML document is not processed. It is important that DTDs be processed to resolve entities and attribute defaults even when processing document fragments that cannot be validated.

### The DAD file for XML collections

For XML collections, the DAD file maps the structure of the XML document to the DB2 database tables from which you compose the document, or where you decompose the document.

For example, if you have an element called <Tax> in your XML document, you might need to map <Tax> to a column called TAX. You define the relationship between the XML data and the relational data in the DAD.

You specify the DAD file name when you enable a collection, or when you use the DAD file in XML collection stored procedures. If you choose to validate XML documents with a DTD, the DAD file can be associated with that DTD. When used as the input parameter of the XML Extender stored procedures, the DAD file has a data type of CLOB. This file can be up to 100 KB.

To specify the XML collection access and storage method, you use the tag in the DAD file. The <Xcollection> tag specifies that the XML data is either to be decomposed from XML documents into a collection of relational tables, or to be composed into XML documents from a collection of relational tables.

An XML collection is a virtual name for a set of relational tables that contains XML data. Applications can enable an XML collection of any user tables. These user tables can be existing tables of legacy business data or tables that the XML Extender recently created.

The DAD specifies mappings between elements or attributes of the XML document and columns of tables. It uses the following kinds of nodes to specify these mappings:

**root_node**
Specifies the root element of the document.

**element_node**
Identifies an element, which can be the root element or a child element.

**text_node**
Represents the CDATA text of an element.

**attribute_node**
Represents an attribute of an element.

Figure 4 on page 42 shows a fragment of the mapping that is used in a DAD file. The nodes map the XML document content to table columns in a relational table.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "dxx_install/samples/extenders/db2xml/dtd/dad.dtd">
<DAD>
  ...
<Xcollection>

<SQL_stmt>
 SELECT order_key, customer, part_key, color, qty
  FROM order_tab o, part_tab p
 WHERE order_key = 1 AND p.order_key = o.order_key
 ORDER BY order_key
</SQL_stmt>

<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE Order SYSTEM "dxx_install/samples/extenders/db2xml/dtd/
getstart.dtd"</doctype><root_node>
   <element_node name="Order">     --> Identifies the element <Order>
     <attribute_node name="key">   --> Identifies the attribute "key"
        <column name="order_key"/> --> Defines the name of the column,
                                       "order_key", to which the element
                                        and attribute are mapped
                                      </attribute_node>
     <element_node name="Customer">  --> Identifies a child element of
                                           <Order> as <Customer>
        <text_node>                   --> Specifies the CDATA text for
                                           the element<Customer>
          <column name="customer">    --> Defines the name of the column,
                                          "customer", to which the child
                                           element is mapped
        </text_node>
     </element_node>
        ...
   </element_node>

      ...
</root_node>
</Xcollection>
</DAD>
```

*Figure 4. Node definitions in a DAD file for an XML collection*

In the above figure, the first two columns in the SQL statement are mapped to an attribute and column.

## Mapping schemes for XML collections

If you are using an XML collection, you must select a *mapping scheme* that defines how XML data is represented in a relational database. Because XML collections must map to a hierarchical structure that is used in XML documents with a relational structure, you should understand how the two structures compare. Figure 5 on page 43 shows how the hierarchical structure can be mapped to relational table columns.

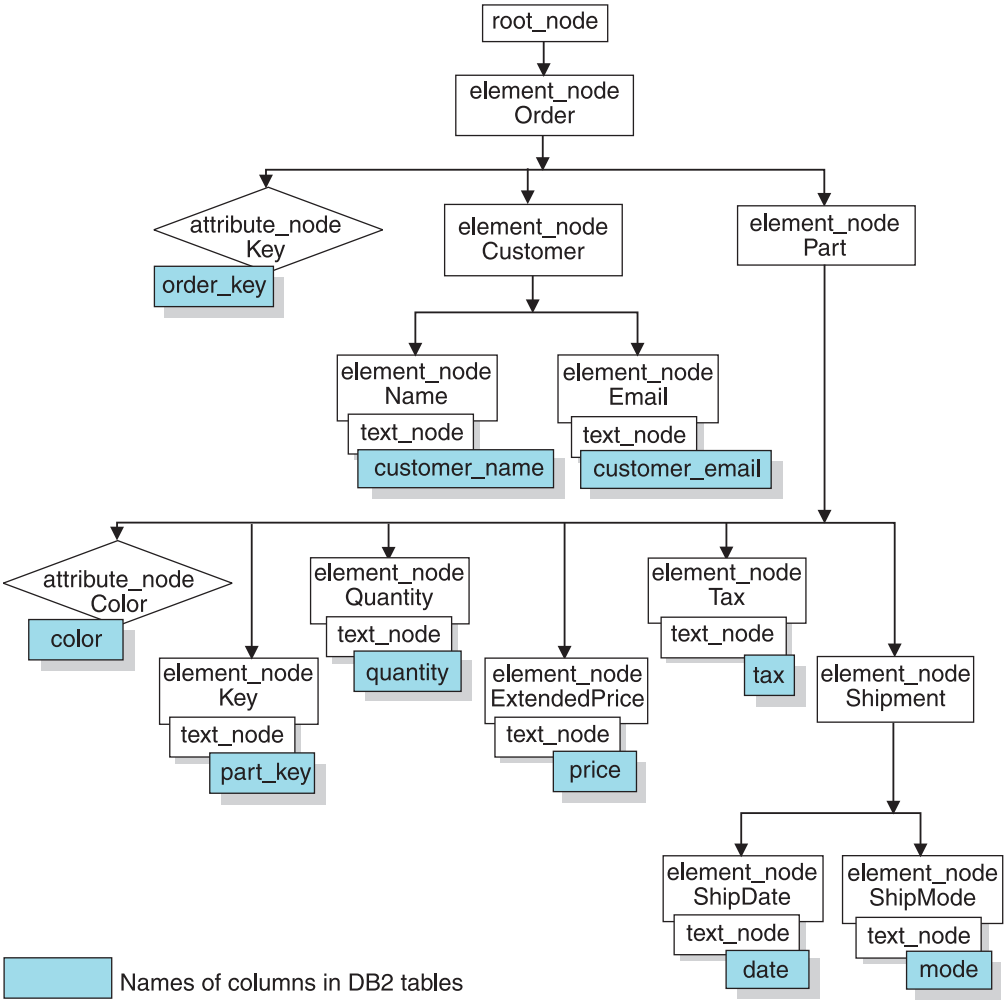



*Figure 5. XML document structure mapped to relational table columns*

The XML Extender uses the mapping scheme when composing or decomposing XML documents using relational data that is located in multiple relational tables. Before you create the DAD file, you must consider how your XML data is mapped to the XML collection.

## Types of mapping schemes

The mapping scheme is specified in the <Xcollection> element in the DAD file. The XML Extender provides two types of mapping schemes: *SQL mapping* and *relational database (RDB_node) mapping*.

**SQL mapping**

Allows direct mapping from relational data to XML documents through a single SQL statement. SQL mapping is used for composition; it is not used for decomposition. SQL mapping is defined with the SQL_stmt element in the DAD file. The content of the SQL_stmt element is a valid SQL statement. The SQL_stmt element maps the columns in the SELECT clause to XML elements or attributes that are used in the XML document. The column names in the SQL statement's SELECT clause are used to define the value of an attribute_node or the content of text_node. The FROM clause defines the tables containing the data; the WHERE clause specifies the join and search condition.

The SQL mapping gives DB2 database users the power to map the data using SQL. When using SQL mapping, you must be able to join all tables in one SELECT statement to form a query. If one SQL statement is not sufficient, consider using RDB_node mapping. To tie all tables together, the primary key and foreign key relationship is recommended among these tables.

**RDB_node mapping**
Defines the location of the content of an XML element or the value of an XML attribute so that the XML Extender can determine where to store or retrieve the XML data.

This method uses the XML Extender-provided *RDB_node*, which contains one or more node definitions for tables, optional columns, and optional conditions. The tables and columns are used to define how the XML data is to be stored in the database. The condition specifies the criteria for selecting XML data or the way to join the XML collection tables.

To define a mapping scheme, you create a DAD with an <Xcollection> element. Figure 6 on page 45 shows a fragment of a sample DAD file with an XML collection SQL mapping that composes a set of XML documents from data in three relational tables.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "dxx_install/samples/db2xml/dtd/dad.dtd">
<DAD>
  <dtdid>dxx_install/samples/db2xml/dtd/dad/
   getstart.dtd</dtdid>
  <validation>YES</validation>
  <Xcollection>
    <SQL_stmt>
       SELECT o.order_key, customer, p.part_key, quantity, price, tax, date,
              ship_id, mode, comment
         FROM order_tab o, part_tab p,
               table(select
               substr(char(timestamp(generate_unique())),16)

               as ship_id, date, node, from ship_tab)  shipid
        WHERE p.price > 2500.00 and s.date > "1996-06-01" AND
              p.order_key = o.order_key and s.part_key = p.part_key
    </SQL_stmt>
    <prolog>?xml version="1.0"?</prolog>
    <doctype>!DOCTYPE DAD SYSTEM "dxx_install
     /samples/db2xml/dtd/getstart.dtd"</doctype>
    <root_node>
    <element_node name="Order">
    <attribute_node name="key">
   <column_name="order_key"/>
    </attribute_node>
    <element_node name="Customer">
     <text_node>
      <column name="customer"/>
     </text_node>
    <element_node>

  ...

    </element_node><!-end Part->
    </element_node><!-end Order->
    </root_node>
  </Xcollection>
</DAD>
```

*Figure 6. SQL mapping scheme*

The XML Extender provides several stored procedures that manage data in an XML collection. These stored procedures support both types of mapping, but require that the DAD file follow the rules that are described in "Mapping scheme requirements."

## Mapping scheme requirements

The following sections describe requirements for each type of the XML collection mapping schemes.

### Mapping scheme requirements for SQL mapping

In this mapping scheme, you must specify the SQL_stmt element in the DAD <Xcollection> element. The SQL_stmt should contain a single SQL statement that can join multiple relational tables with the query predicate. In addition, the following clauses are required:

- **SELECT clause**
  - Ensure that the name of the column is unique. If two tables have the same column name, use the AS keyword to create an alias name for one of them.
  - Group the columns of the same table together, and use the logical hierarchical level of the relational tables. This means that you group

the columns according to how the tables map to the hierarchical structure of your XML document. In the SELECT clause, the columns of the higher-level tables should proceed the columns of lower-level tables. The following example demonstrates the hierarchical relationship among tables:

```
SELECT o.order_key, customer, p.part_key, quantity, price, tax,
       ship_id, date, mode
```

In this example, `order_key` and `customer` from table ORDER_TAB have the highest relational level because they are higher on the hierarchical tree of the XML document. The `ship_id`, `date`, and `mode` from table SHIP_TAB are at the lowest relational level.

– The composition algorithm uses the candidate key information as a hint to assist with its processing. Use a single-column candidate key to begin each table level in the select clause. If such a key is not available in a table, the query should generate one for that table using a table expression and thebuilt-in user-defined function, generate_unique(). Generated keys do not need to appear in the output document, but they do need to be in the SELECT clause. In the above example, the `o.order_key` is the primary key for ORDER_TAB, and the `part_key` is the primary key of PART_TAB. They appear at the beginning of their own group of columns that are to be selected. Because the SHIP_TAB table does not have a primary key, one must be generated, in this case, `ship_id`. This primary key is listed as the first column for the SHIP_TAB table group. Use the FROM clause to generate the primary key column, as shown in the following example.

• **FROM clause**

– Use a table expression and the built-in function, generate_unique(), to generate a single key for tables that do not have a primary single key. For example:

```
FROM order_tab as o, part_tab as p,
        table(select substr(char(timestamp
        (generate_unique())),16) as
        ship_id, date, mode from ship_tab) as s
```

In this example, a single column candidate key is generated with the function, generate_unique() and given an alias named `ship_id`.

– Use an alias name when needed to make a column distinct. For example, you could use `o` for ORDER_TAB, `p` for PART_TAB, and `s` for SHIP_TAB.

• **WHERE clause**

– Specify the join condition that ties tables in the collection together. For example:

```
WHERE p.price > 2500.00 AND s.date > '2003-06-01' AND
      p.order_key = o.order_key AND s.part_key = p.part_key
```

– Specify any other search condition in the predicate. Any valid predicate can be used.

• **ORDER BY clause**

– Define the ORDER BY clause at the end of the SQL_stmt element.

– Ensure that the column names match the column names in the SELECT clause.

– List the single-column candidate keys in the order of the hierarchical level of their corresponding tables.

- Maintain the top-down order of the hierarchy of the entities. The column specified in the ORDER BY clause must be the first column listed for each entity. Keeping the order ensures that the XML documents to be generated do not contain incorrect duplicates.
- Do not qualify the columns in ORDER BY by any schema or table name.

Although the SQL_stmt element has the preceding requirements, it is powerful because you can specify any predicate in your WHERE clause if the expression in the predicate uses the columns in the tables.

**Mapping scheme requirements for RDB_node mapping**

When using this mapping method, do not use the element SQL_stmt in the <Xcollection> element of the DAD file. Instead, use the RDB_node element as a child of the top element_node and of each attribute_node and text_node.

- **RDB_node for the top element_node**

  The top element_node in the DAD file represents the root element of the XML document. Specify an RDB_node for the top element_node based on these requirements:

  - Line ending characters are allowed in condition statements.
  - Condition elements can reference a column name an unlimited number of times.
  - Specify all tables that are associated with the XML documents. For example, the following mapping specifies three tables in the RDB_node of the element_node <Order>, which is the top element_node:

    ```
    <element_node name="Order">
        <RDB_node>
            <table name="order_tab"/>
            <table name="part_tab"/>
            <table name="ship_tab"/>
            <condition>
               order_tab.order_key = part_tab.order_key AND
               part_tab.part_key = ship_tab.part_key
            </condition>
        </RDB_node>
    ```

    There are no ordering restrictions on predicates of the root node condition.The condition element can be empty or missing if there is only one table in the collection.
  - If you are decomposing, or are enabling the XML collection specified by the DAD file, specify a primary key for each table. The primary key can consist of a single column or multiple columns, called a composite key. The primary key is specified by adding an attribute key to the table element of the RDB_node. When a composite key is supplied, the key attribute is specified by the names of key columns separated by a space. For example:

    ```
    <table name="part_tab" key="part_key price"/>
    ```

    The information specified for decomposition is ignored when composing a document.
  - Use the orderBy attribute to recompose XML documents that contain elements or attributes with multiple occurrences back to their original structure. This attribute allows you to specify the name of a column

that will be the key used to preserve the order of the document. The orderBy attribute is part of the table element in the DAD file, and it is an optional attribute.

- **RDB_node for each attribute_node and text_node**

  You need to specify an RDB_node for each attribute_node and text_node, that tells the stored procedure from which table, which column, and under which query condition to get the data. You must specify the table and column values; the condition value is optional.

  – Specify the name of the table that contains the column data. The table name must be included in the RDB_node of the top element_node. In this example, for text_node of element <Price>, the table is specified as PART_TAB.

```
<element_node name="Price">
    <text_node>
        <RDB_node>
            <table name="part_tab"/>
            <column name="price"/>
            <condition>
                 price > 2500.00
            </condition>
        </RDB_node>
    </text_node>
</element_node>
```

  – Specify the name of the column that contains the data for the element text. In the previous example, the column is specified as PRICE.

  – Specify a condition if you want XML documents to be generated using the query condition. Allowable syntax for <condition> is as follows:

    - columnname
    - operator
    - literal

    In the example above, the condition is specified as `price > 2500.00`. Only the data that meets the condition is in the generated XML documents. The condition must be a valid WHERE clause.

  – If you are decomposing a document, or are enabling the XML collection specified by the DAD file, specify the column type for each attribute_node and text_node. Column types are specified by adding the attribute type to the column element. For example:

```
<column name="order_key" type="integer"/>
```

    The information specified for decomposition is ignored when composing a document.

- **Maintain the top-down order of the hierarchy of the entities.** This means ensure the element nodes are nested properly so that the XML Extender understands the relationship between the elements when composing or decomposing. For example, using the following DAD file, that does not nest Shipment inside of Part:

```
<element_node name="Part">
    ...
    <element_node name="ExtendedPrice">
          ...
    </element_node>
          ...
</element_node> <!-- end of element Part -->

<element_node name="Shipment" multi_occurrence="YES">
```

```
            <element_node name="ShipDate">
                ...
            </element_node>
            <element_node name="ShipMode">
                ...
            </element_node>

    </element_node> <!-- end of element Shipment-->
```

Which might produce an XML file in which the Part and Shipment are sibling elements.

```
<Part color="black ">
  <key>68</key>
  <Quantity>36</Quantity>
  <ExtendedPrice>34850.16</ExtendedPrice>
  <Tax>6.000000e-2</Tax>
</Part>

<Shipment>
    <ShipDate>1998-08-19</ShipDate>
    <ShipMode>BOAT  </ShipMode>
</Shipment>
```

When you would rather have a DAD that nests Shipment inside of Part:

```
 <element_node name="Part">
     ...
     <element_node name="ExtendedPrice">
          ...
     </element_node>
     ...
    <element_node name="Shipment" multi_occurrence="YES">
          <element_node name="ShipDate">
              ...
          </element_node>
          <element_node name="ShipMode">
              ...
          </element_node>

     </element_node> <!-- end of element Shipment-->
 </element_node> <!-- end of element Part -->
```

Which produces an XML file with Shipment as a child element of Part:

```
<Part color="black ">
  <key>68</key>
  <Quantity>36</Quantity>
  <ExtendedPrice>34850.16</ExtendedPrice>
  <Tax>6.000000e-2</Tax>
  <Shipment>
    <ShipDate>1998-08-19</ShipDate>
    <ShipMode>BOAT  </ShipMode>
  </Shipment>
</Part>
```

With the RDB_node mapping approach, you don't need to supply SQL statements. However, putting complex query conditions in the RDB_node element can be more difficult.

## Decomposition table size requirements for RDB node mapping

Decomposition uses RDB_node mapping to specify how an XML document is decomposed into DB2 database tables by extracting the element and attribute values into table rows. The values from each XML document are stored in one or more DB2 tables. Each table can have a maximum of 10240 rows decomposed from each document. For example, if an XML document is decomposed into five tables, each of the five tables can have up to 10240 rows for that particular document.

Using multiple-occurring elements (elements with location paths that can occur more than once in the XML structure) affects the number of rows inserted for each document. For example, a document that contains an element <Part> that occurs 20 times, might be decomposed as 20 rows in a table. When using multiple occurring elements, consider that a maximum of 10240 rows can be decomposed into one table from a single document.

**Related concepts:**

- "DAD files for XML collections" on page 163

**Related tasks:**

- "Storing a DTD in the repository table" on page 52

## Validating XML documents automatically

After you choose an access and storage method, either XML column or XML collection, you can determine whether to *validate* the XML documents. You can also validate XML documents that are composed from XML collections.

You can have your XML data validated automatically by specifying YES for validation in the DAD file. To have a document validated when it is stored into DB2®, you must specify a DTD within the `<dtdid>` element or in the `<!DOCTYPE>` specification in the original document. To have a document validated when it is composed from an XML collection in DB2, you must specify a DTD within the `<dtdid>` element or within the `<doctype>` element in the DAD file.

The following factors should be taken into consideration when you decide whether to validate your documents.

- The DTD ID or schema is useful only if you decide to validate the XML document.

  To validate the DAD with a schema, insert the schema tags that associate the DAD file with the schema file. For example:

  ```
  <schemabindings>
  <nonamespacelocation location="path/schema_name.xsd"/>
  </schemabindings>
  ```

- You do not need a DTD to store or archive XML documents.
- It might be necessary to process the DTD to set entity values and attribute defaults regardless of whether you choose to validate.
- If you specify NO for validation in the DAD, then the DTD specified by the XML document is not processed.
- Validating your XML data has a performance impact.

## Enabling databases for XML

Before you can store or retrieve XML documents from DB2 with XML Extender, you must enable the database for XML. The XML Extender enables the database that you are connected to, using the current instance.

When you enable a database for XML, the XML Extender does these tasks:

- Creates all the user-defined types (UDTs), user-defined functions (UDFs), and stored procedures for XML Extender

- Creates and populates control tables with the necessary metadata that the XML Extender requires
- Creates the DB2XML schema in user-defined table spaces.

  The fully qualified name of an XML function is db2xml.*function-name*, where db2xml is an identifier that provides a logical grouping for SQL objects. You can use the fully qualified name anywhere that you refer to a UDF or a UDT. You can also omit the schema name when you refer to a UDF or a UDT; in this case, DB2 uses the function path to determine the function or data type.
- Assigns necessary privileges to metadata tables.

**Procedure:**

You can enable a database with the administration wizard or from a command line. To do this task from the command line, type **dxxadm** from the command line and specify the database that is to be enabled.

The following example enables an existing database called SALES_DB.

```
dxxadm enable_db SALES_DB
```

To enable a database using the administration wizard, complete the following tasks:

1. Start the administration wizard and click **Enable database** from the Launchpad window.

   If a database is already enabled, the **Disable database** button appears. If the database is disabled, the **Enable database** button appears.

   When the database is enabled, you return to the LaunchPad window.

After you enable a database, you can use the XML Extender UDTs, UDFs, and stored procedures.

**Related concepts:**

- "Migrating XML Extender from previous versions or releases" on page 35

## Creating an XML table

This task is part of the larger task of defining and enabling an XML column.

An XML table is used to store intact XML documents. To store whole documents in your database with DB2 XML Extender, you must create a table so that it contains a column with an XML user-defined type (UDT). DB2 XML Extender provides you with three user-defined types to store your XML documents as column data. These UDTs are: XMLVARCHAR, XMLCLOB, and XMLFILE. When a table contains a column of XML type, you can then enable the table for XML.

You can create a new table to add a column of XML type using the administration wizard or the command line.

**Procedure:**

To create a table with a column of XML type using the command line:

Open the DB2 command prompt and type a Create Table statement.

For example, in a sales application, you might want to store an XML-formatted line-item order in a column called ORDER of a table called SALES_TAB. This table also has the columns INVOICE_NUM and SALES_PERSON. Because it is a small order, you store the sales order using the XMLVARCHAR type. The primary key is INVOICE_NUM. The following CREATE TABLE statement creates a table with a column of XML type:

```
CREATE TABLE sales_tab(
            invoice_num    char(6)   NOT NULL PRIMARY KEY,
            sales_person   varchar(20),
            order          db2xml.XMLVarchar);
```

After you have created a table, the next step is to enable the column for XML data.

**Related concepts:**

- "Planning side tables" on page 56

## Storing a DTD in the repository table

You can use a DTD to validate XML data in an XML column or in an XML collection. DTDs can be stored in the DTD repository table, a DB2 database table called DTD_REF. The DTD_REF table has a schema name of DB2XML. Each DTD in the DTD_REF table has a unique ID. The XML Extender creates the DTD_REF table when you enable a database for XML. You can insert the DTD from the command line or by using the administration wizard.

**Procedure:**

To insert the DTD using the administration wizard:

1. Start the administration wizard and click **Import a DTD** from the Launchpad window to import an existing DTD file into the DTD repository for the current database. The Import a DTD window opens.
2. Specify the DTD file name in the **DTD file name** field.
3. Type the DTD ID in the **DTD ID** field.

   The DTD ID is an identifier for the DTD. It can also be the path that specifies the location of the DTD on the local system. The DTD ID must match the value that is specified in the DAD file for the <DTDID> element.
4. Optional: Type the name of the author of the DTD in the **Author** field.
5. Click **Finish** to insert the DTD into the DTD repository table, DB2XML.DTD_REF, and return to the Launchpad window.

To insert a DTD from the command line, issue a SQL INSERT statement from Table 6. For example:

```
DB2 INSERT into DB2XML.DTD_REF values('dxx_install
/samples/extenders/db2xml/dtd/getstart.dtd',
DB2XML.XMLClobFromFile('dxx_install/dxxsamples/dtd/getstart.dtd',
0, 'user1', 'user1', 'user1');
```

*Table 6. The column definitions for the DTD repository table*

| Column name | Data type | Description |
|---|---|---|
| DTDID | VARCHAR(128) | ID of the DTD. |
| CONTENT | XMLCLOB | Content of the DTD. |
| USAGE_COUNT | INTEGER | Number of XML columns and XML collections in the database that use this DTD. |

*Table 6. The column definitions for the DTD repository table (continued)*

| Column name | Data type | Description |
|---|---|---|
| AUTHOR | VARCHAR(128) | Author of the DTD, optional information for the user to input. |
| CREATOR | VARCHAR(128) | User ID that does the first insertion. |
| UPDATOR | VARCHAR(128) | User ID that does the last update. |

## Enabling XML columns

To store an XML document in a DB2 database, you must enable for XML the column that will contain the document. Enabling a column prepares it for indexing so that it can be searched quickly. You can enable a column by using the XML Extender Administration wizard or the command line. The column must be of XML type.

When XML Extender enables an XML column, it performs the following operations:

- Reads the DAD file to:
  - Check for the existence of the DTD in the DTD_REF table, if the DTDID was specified.
  - Create side tables on the XML column for indexing purpose.
  - Prepare the column to contain XML data.
- Optionally creates a default view of the XML table and side tables. The default view displays the application table and the side tables.
- Specifies a ROOT ID column, if one is not specified.

After you enable the XML column, you can:

- Create indexes on the side tables.
- Insert XML documents in the XML column.
- Query, update, or search the XML documents in the XML column.

You can enable XML columns using the Administration wizard or from a DB2 command line.

**Procedure (using the administration wizard):**

To enable XML columns using the administration wizard:

1. Set up and start the Administration wizard.
2. Click **Work with XML Columns** from the Launchpad window to view the tasks related to the XML Extender columns. The Select a Task window opens.
3. Click **Enable a Column** and then **Next**.
4. Specify the table and column.
   - Select the table that contains the XML column from the **Table name** field.
   - Select the column to enable from the **Column name** field.
5. Specify the DAD path and file name in the **DAD file name** field. For example:
   `dxx_install/samples/dad/getstart.dad`
6. Optional: Type the name of an existing table space in the **Table space** field.

The default table space contains side tables that XML Extender created. If you specify a table space, the side tables are created in the specified table space. If you do not specify a table space, the side tables are created in the default table space.

7. Optional: Type the name of the default view in the **Default view** field.

   If specified, the default view is automatically created when the column is enabled. The default view joins the XML table and all of the related side tables.

8. Recommended: Type the column name of the primary key for the table in the **Root ID** field.

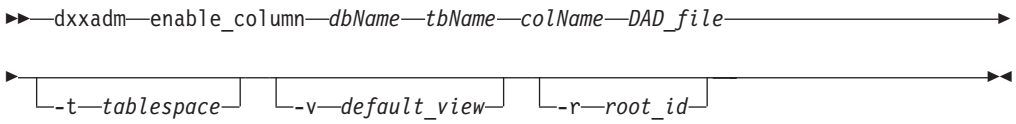
   XML Extender uses the value of **Root ID** as a unique identifier to associate all side tables with the application table. The XML Extender adds the DXXROOT_ID column to the application table and generates an identifier.

9. Click **Finish** to enable the XML column, create the side tables, and return to the Launchpad window.
   - If the column is successfully enabled, you receive the message: `column is enabled`.
   - If the column is not successfully enabled, an error message is displayed, along with prompts for you to correct the values of the entry fields until the column is successfully enabled.

**Procedure (using the command line):**

To enable an XML column using the command line, use the DXXADM enable_column command.

**Syntax:**




**Parameters:**

*dbName*
The name of the database.

*tbName*
The name of the table that contains the column to be enabled.

*colName*
The name of the XML column that is being enabled.

*DAD_file*
The name of the file that contains the document access definition (DAD).

*default_view*
Optional. The name of the default view that XML Extender created to join an application table and all of the related side tables.

*root_id* Optional, but recommended. The column name of the primary key in the application table and a unique identifier that associates all side tables with the application table. DB2 XML Extender uses the value of ROOT_ID as a unique identifier to associate all side tables with the application table. If the ROOT ID is not specified, XML Extender adds the DXXROOT_ID column to the application table and generates an identifier.

**Restriction:** If the application table has a column name of DXXROOT_ID, you must specify the *root_id* parameter; otherwise, an error occurs.

**Example:**

```
dxxadm enable_column SALES_DB sales_tab order getstart.dad
            -v sales_order_view -r INVOICE_NUMBER
```

In this example, the ORDER column is enabled in the SALES_TAB table . The DAD file is `getstart.dad`, the default view is sales_order_view, and the ROOT ID is INVOICE_NUMBER.

Using this example, the SALES_TAB table has the following columns:

| Column name | Data type |
|---|---|
| INVOICE_NUM | CHAR(6) |
| SALES_PERSON | VARCHAR(20) |
| ORDER | XMLVARCHAR |

The following side tables are created based on the DAD specification:

**ORDER_SIDE_TAB:**

| Column name | Data type | Path expression |
|---|---|---|
| ORDER_KEY | INTEGER | `/Order/@Key` |
| CUSTOMER | VARCHAR(50) | `/Order`<br>`/Customer`<br>`/Name` |
| INVOICE_NUM | CHAR(6) | N/A |

**PART_SIDE_TAB:**

| Column name | Data type | Path expression |
|---|---|---|
| PART_KEY | INTEGER | `/Order/Part/@Key` |
| PRICE | DOUBLE | `/Order/Part`<br>`/ExtendedPrice` |
| INVOICE_NUM | CHAR (6) | N/A |

**SHIP_SIDE_TAB:**

| Column name | Data type | Path expression |
|---|---|---|
| DATE | DATE | `/Order/Part/`<br>`Shipment/ShipDate` |
| INVOICE_NUM | CHAR (6) | N/A |

All of the side tables have the column INVOICE_NUM of the same type, because the ROOT ID is specified by the primary key INVOICE_NUM in the application table. After the column is enabled, the value of the INVOICE_NUM column is inserted in side tables when a row is inserted in the main table. If you specify a

default view when enabling the XML column ORDER, then XML Extender creates a default view, sales_order_view. The view joins the above tables using the following statement:

```
CREATE VIEW sales_order_view(invoice_num, sales_person, order,
                             order_key, customer, part_key, price, date)
     AS
     SELECT sales_tab.invoice_num, sales_tab.sales_person, sales_tab.order,
            order_side_tab.order_key, order_side_tab.customer,
            part_side_tab.part_key, part_side_tab.price,
            ship_tab.date
     FROM sales_tab, order_side_tab, part_side_tab, ship_side_tab
     WHERE sales_tab.invoice_num = order_side_tab.invoice_num
               AND sales_tab.invoice_num = part_side_tab.invoice_num
               AND sales_tab.invoice_num = ship_side_tab.invoice_num
```

If you specify the table space when you enabled, the side tables are created in the specified table space. If the table space is not specified, the side tables are created in the default table space.

## Planning side tables

Side tables are DB2® tables used to extract the content of an XML document that will be searched frequently. The XML column is associated with side tables that hold the contents of the XML document. When the XML document is updated in the application table, the values in the side tables are automatically updated.

Figure 7 shows an XML column with side tables.

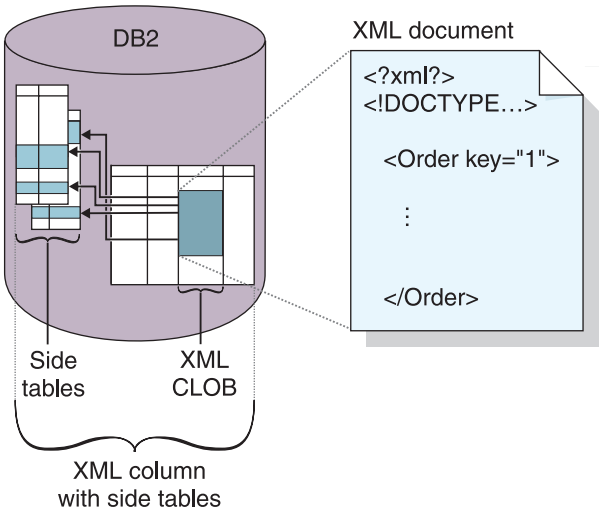



*Figure 7. An XML column whose content is mapped in side tables. There is an XML file in the column that is associated with side tables that hold the contents of the XML document.*

When planning for side tables, you must consider how to organize the tables, how many tables to create, and whether to create a default view for the side tables. Base these decisions on whether elements and attributes can occur multiple times and your requirements for query performance. Do not plan to update the side tables in any way; they will be automatically updated when the document is updated in the XML column.

**Multiple occurrence:**

When elements and attributes occur multiple times in side tables, consider the following issues in your planning:

- For elements or attributes in an XML document that have multiple occurrences, you must create a separate side table for each XML element or attribute with multiple occurrences, due to the complex structure of XML documents. This means that elements or attributes have location paths that occur multiple times and must be mapped to a table with only one column. You cannot have any other columns in the table.
- When a document has multiple occurring location paths, XML Extender adds a column named DXX_SEQNO with a type of INTEGER in each side table to track the order of elements that occur more than once. With DXX_SEQNO, you can retrieve a list of the elements in the same order as the original XML document by specifying ORDER BY DXX_SEQNO in an SQL query.

**Default views and query performance:**

When you enable an XML column, you can specify a default, read-only view that joins the application table with the side tables using a unique ID, called the ROOT ID. With the default view, you can search XML documents by querying the side tables. For example, if you have the application table SALES_TAB, and the side tables ORDER_TAB, PART_TAB and SHIP_TAB, your query might look as follows:

```
SELECT sales_person FROM sales_order_view
      WHERE price > 2500.00
```

The SQL statement returns the names of salespeople in the SALES_TAB who have orders stored in the column ORDER, and where the PRICE column is greater than 2500.00.

The advantage of querying the default view is that it provides a virtual single view of the application table and side tables. However, the more side tables that are created, the more expensive the query. Therefore, creating the default view is recommended only when the total number of side-table columns is small. Applications can create their own views that join the important side table columns.

## Indexing side tables

This task is part of the larger task of defining and enabling an XML column.

Side tables contain the XML data in the columns that you specified when you created the DAD file. After you enable an XML column and create side tables, you can index the side tables. Indexing these tables helps you improve the performance of the queries against the XML documents.

**Procedure:**

To create an index for your side tables from a DB2 database command line, use the DB2 CREATE INDEX SQL statement.

The following example creates indexes on four side tables using the DB2 command prompt.

```
DB2 CREATE INDEX KEY_IDX
      ON ORDER_SIDE_TAB(ORDER_KEY)

DB2 CREATE INDEX CUSTOMER_IDX
      ON ORDER_SIDE_TAB(CUSTOMER)
```

```
DB2 CREATE INDEX PRICE_IDX
      ON PART_SIDE_TAB(PRICE)

DB2 CREATE INDEX DATE_IDX
      ON SHIP_SIDE_TAB(DATE)
```

## Composing XML documents by using SQL mapping

You should use SQL mapping to compose XML documents if you want to use an SQL statement to define the table and columns that you will derive the data in the XML document. You can use SQL mapping only for composing XML documents. You create a DAD file to compose the XML document with SQL mapping.

**Prerequisites:**

Before you compose your documents, you must first map the relationship between your DB2 database tables and the XML document. This step includes mapping the hierarchy of the XML document and specifying how the data in the document maps to a DB2 database table.

**Procedure:**

To compose XML documents from the command line, complete the following steps:

1. Create a new document in a text editor and type the following syntax:
   ```
   <?XML version="1.0"?>
   <!DOCTYPE DAD SYSTEM "path/dad.dtd">
   ```
2. Insert the <DAD> </DAD> tags.

   The DAD element will contain all the other elements.
3. Insert the tags used for validating the DAD with a DTD or schema.
   - To validate the composed XML document with a DTD, insert the DTDID tags that associate the DAD file with the XML document DTD. For example:
     ```
     <dtdid>dtd_name</dtdid>
     ```
   - To validate the composed XML document with a schema, insert the schema tags that associate the DAD file with the schema file. For example:
     ```
     <schemabindings>
     <nonamespacelocation location="path/schema_name.xsd"/>
     </schemabindings>
     ```

   The dtd or schema is useful only if you decide to validate the XML document. Use the validation tag to indicate whether DB2 XML Extender validates the XML document:
   - If you want to validate the XML document, then type:
     ```
     <validation>YES</validation>
     ```
   - If you do not want to validate the XML document type:
     ```
     <validation>NO</validation>
     ```
4. Type <XCollection> </XCollection> tags to specify that you are using XML collections as the access and storage method for your XML data.
5. Inside the <Xcollection> </Xcollection> tags, insert the <SQL_stmt> </SQL_stmt> tags to specify the SQL statement that will map the relational data to the XML documents. This statement is used to query data from DB2 database tables. The following example shows a sample SQL query:

```
<SQL_stmt>
  SELECT o.order_key, customer_name, customer_email, p.part_key, color,
  quantity, price, tax, ship_id, date, mode from order_tab o, part_tab p,
  table (select substr(char(timestamp(generate_unique())),16)
    as ship_id, date, mode, part_key from ship_tab)  s
       WHERE o.order_key = 1 and
             p.price > 20000 and
             p.order_key = o.order_key and
             s.part_key = p.part_key
       ORDER BY order_key, part_key, ship_id
</SQL_stmt>
```

The example SQL statement for mapping the relational data to the XML document meets the following requirements:

- Columns are specified in top-down order by the hierarchy of the XML document structure.
- The columns for an entity are grouped together.
- The object ID column is the first column in each group.
- The Order_tab table does not have a single key column, and therefore, the generate_unique DB2 database built-in function is used to generate the ship_id column.
- In the ORDER BY clause, the object ID columns are listed in order corresponding to the top-down hierarchy of tables to which they belong. The columns in ORDER BY should not be qualified by any schema and the column names must match the column names in the SELECT clause.

6. Add the following prolog information to be used in the composed XML document. If you need to specify an encoding value for internationalization, add the ENCODING attribute and value.

   ```
   <prolog>?xml version="1.0"?</prolog>
   ```

7. Type the <doctype> </doctype> tag. This tag contains the DOCTYPE declaration to be inserted in each composed document. For example:

   ```
   <doctype>! DOCTYPE Order SYSTEM "dxx_install
   /samples/extenders/db2xml/dtd/getstart.dtd"</doctype>
   ```

8. Specify the root element and the elements and attributes that make up the XML document:
   a. Add the <root></root_node> tag to define the root element. All the elements and attributes that make up the XML document are specified within the root_node.
   b. Use the `<element_node>`, `<attribute_node>`, `<text_node>`, and `<column>` tags to specify the names of elements and attributes in the composed document and to map them to the columns specified in the SQL statement.

      **<element_node> tag**
      Specifies the elements in the XML document. Set the name attribute of the element_node tag to the name of the element. Each element_node can have child element_nodes.

      **<attribute_node> tag**
      Specifies the attributes of an element in the XML document. An <attribute_node> tag is a child element of <element_node>. Set the name attribute of the attribute_node tag to the name of the attribute.

      **<text_node> tag**
      Specifies the text content of the element and the column data in a relational table for bottom-level element_nodes. For each bottom-level element, specify <text_node> tags that indicate that

the element contains character data to be extracted from DB2 when the document is composed. For each bottom-level element_node, use a <column> tag to specify from which column to extract data when the XML document is composed. Column tags are typically inside the <attribute_node> or the <text_node> tags. All column names defined must be in the <SQL_stmt> SELECT clause at the beginning of the DAD file.

**<column> tag**
Specifies the column from which the data is retrieved for the element or attribute value.

9. Ensure that the ending tags are in the appropriate places:
   a. Ensure that an ending </root_node> tag is after the last </element_node> tag.
   b. Ensure that an ending </Xcollection> tag is after the </root_node> tag.
   c. Ensure that an ending </DAD> tag is after the </Xcollection> tag.
10. Save the file as *file.dad*. Where *file* is the name of your file.

The following Windows example shows a complete DAD:

```
<?xml version'"1.0"?>
<!DOCTYPE DAD SYSTEM "C:\dxx_xml\test\dtd\dad.dtd">
<DAD>
<validation>NO</validation>
<Xcollection>
<SQL_stmt> select o.order_key, customer_name, customer_email,
 p.part_key, color, qty, price, tax, ship_id, date, mode from order_tab o,
 part_tab p, (select timestamp(generate_unique()) as
ship_id, date, mode, part_key from ship_tab) s where
 o.order_key = 1 and p.price . 20000 and p.order_key
 = o.order_key and s.part_key =p.part_key ORDER BY order_key,
part_key, ship_id</SQL_stmt>
<prolog>?XML version="1.0"?</prolog>
<doctype>!DOCTYPE ORDER SYSTEM
"C:\dxx_install\samples\extenders\db2xml\dtd/Order.dtd"
 </doctype>
 <root_node>
 <element_node name="Order">
  <attribute_node name="key">
   <column name="order_key"/>
   </attribute_node>
   <element_node name="Customer">
    <element_node name="NAME">
     <text_node><column name="customer_name"/></text_node>
     </element_node>
     </element_node>
    <element_node name="Part">
     <attribute_node name="color">
      <column name="color"/>
     </attribute_node>
     <element_node name="key">
      <text_node><column name="part_key"/></text_node>
     </element_node>
     <element_node name ="Quantity">
      <text_node><column name="qty"/></text_node>
      </element_node>
     <element_node name="ExtendedPrice">
      <text_node><column name="price"/></text_node>
      </element_node>
     <element_node name="Tax">
      <text_node><column name="tax"/></text_node>
       </element_node>
      <element_node name="Shipment" multi_occurrence="YES">
```

```
 <element_node name=shipDate">
  <text_node><column name="date"/><text_node>
   <element_node>
  <element_node name="ShipMode">
   <text_node><column name="mode"/></text_node>
  </element_node>
 </element_node>
</element_node>
</root_node>
</Xcollection>
</DAD>
```

Call the dxxGenXML stored procedure to compose using the DAD. For more information on tests2x, see "Dynamically overriding values in the DAD file" on page 171.

## Composing XML collections by using RDB_node mapping

You must first create a DAD file, and then compose the XML collection using the DAD file that you created. Use <RDB_node> to specify DB2 database tables, columns, and conditions for an element or attribute node. The <RDB_node> uses the following elements:

**table** Defines the table that corresponds to the element.

**column** Defines the column that contains the corresponding element.

**condition** Optionally specifies a condition on which insertion into the column occurs.

The child elements that are used in the RDB_node element depend on the context of the node and use the following rules:

| If the node type is: | The following RDB child elements are used: | | |
|---|---|---|---|
| | **Table** | **Column** | **Condition** |
| Root element | Yes | No | Yes[1] |
| Attribute | Yes | Yes | Optional |
| Text | Yes | Yes | Optional |

[1] Required to show relationships between multiple tables.

**Restriction:** If you compose your XML collections using RDB_node mapping, all statements of a given element must map to columns in the same table.

**Procedure:**

To prepare a DAD file to use for decomposing an XML document using the RDB_node mapping:

1. Start with a XML file that includes the following items:
   - The following header, where *path*/dad.dtd is the path and file name of the DTD for the DAD:

     ```
     <?xml version="1.0"?>
     <!DOCTYPE DAD SYSTEM "path/dad.dtd">
     ```
   - The <DAD></DAD> tags. This element contains all other elements.
2. Insert the tags used for validating the DAD with a DTD or schema:

- To validate the XML document with a DTD, insert the dtdid tag that associates the DAD file with the XML document DTD. For example:

  ```
  <dtdid>dtd_name</dtdid>
  ```

- To validate the XML document with a schema, insert the schemabindings tag that associates the DAD file with the schema file. For example:

  ```
  <schemabindings>
  <nonamespacelocation location="path/schema_name.xsd"/>
  </schemabindings>
  ```

The dtdid or schema is useful only if you decide to validate the XML document. Use the validation tag to indicate whether DB2 XML Extender validates the XML document:

- If you want to validate the XML document, then type:

  ```
  <validation>YES</validation>
  ```

- If you do not want to validate the XML document type:

  ```
  <validation>NO</validation>
  ```

3. Insert <XCollection> </XCollection> tags to specify that you are using XML collections as the access and storage method for your XML data.
4. Add the following prolog information. If you need to specify an encoding value for internationalization, add the ENCODING attribute and value.

   ```
   <prolog>?xml version="1.0"?</prolog>
   ```

5. Add the <doctype></doctype> tag. For example:

   ```
   <doctype>! DOCTYPE Order SYSTEM "dxx_install
      /samples/extenders/db2xml/dtd/getstart.dtd"</doctype>
   ```

6. Define the root_node using the <root_node> </root_node> tag.
7. Inside the root_node, map the elements and attributes in the XML document to element nodes and attribute nodes that correspond to DB2 database data. These nodes provide a path from the XML data to the DB2 database data.
   a. Define a top level, root element_node. This element_node contains:
      - Table elements with a join condition to specify the collection.
      - Child elements
      - Attributes

      To specify the table and condition:

      1) Create an RDB_node element within an element_node element. For example:

         ```
         <element_node name="name">
         <RDB_node>
         </RDB_node>
         </element_node>
         ```

      2) Define a <table> element for each table that contains data to be included in the XML document. For example, if you have three tables, ORDER_TAB, PART_TAB, and SHIP_TAB, that have column data to be in the document, create a table node for each. For example:

         ```
         <RDB_node>
         <table name="ORDER_TAB">
         <table name="PART_TAB">
         <table name="SHIP_TAB">
         </RDB_node>
         ```

      3) Define a join condition for the tables in the collection. The syntax is:

         ```
         table_name.table_column = table_name.table_column AND
         table_name.table_column = table_name.table_column ...
         ```

         For example:

```
<RDB_node>
<table name="ORDER_TAB">
<table name="PART_TAB">
<table name="SHIP_TAB">
<condition>
   order_tab.order_key = part_tab.order_key AND
   part_tab.part_key = ship_tab.part_key
</condition>
</RDB_node>
```

4) Optional: Specify a primary key for each table. The primary key consists of a single column or multiple columns, called a composite key. To specify the primary key, add an attribute key to the table element of the RDB_node. The following example defines a primary key for each of the tables in the RDB_node of the root element_node Order:

```
<element_node name="Order">
    <RDB_node>
         <table name="order_tab" key="order_key"/>
         <table name="part_tab" key="part_key price"/>
         <table name="ship_tab" key="date mode"/>
         <condition>
             order_tab.order_key = part_tab.order_key AND
             part_tab.part_key = ship_tab.part_key
         </condition>
    <RDB_node>
```

The key attribute is required for decomposition and enabling a collection.

b. Define an <element_node> tag for each element in your XML document that maps to a column in a DB2 database table. For example:

```
<element_node name="name">
</element_node>
```

An element node can have one of the following types of elements:

**text_node** To specify that the element has content to a DB2 database table It does not have child elements.

**attribute_node**
To specify an attribute.

**child elements**
Children of the element_node.

The text_node contains an RDB_node to map content to a DB2 database table and column name.

For example, to fetch the content of a column called TAX to create an XML element, <tax>.

**XML document:**

```
<Tax>0.02</Tax>
```

In this case, you want the value 0.02 to be fetched in the column TAX.

In the DAD file, you specify an <RDB_node> tag to map the XML element to the DB2 database table and column.

**DAD file:**

```
<element_node name="Tax">
   <text_node>
      <RDB_node>
         <table name="part_tab"/>
```

```
        <column name="tax"/>
      </RDB_node>
    </text_node>
</element_node>
```

The <RDB_node> tag specifies that the value of the Tax element is a text value, the data is stored in the PART_TAB table in the TAX column.

c. Define an <attribute_node> tag for each attribute in your XML document that maps to a column in a DB2 database table. For example:

```
<attribute_node name="key">
</attribute_node>
```

The attribute_node has an RDB_node to map the attribute value to a DB2 database table and column.

For example, you might have an attribute `key` for an element <Order>. The value `key` must be fetched from a column PART_KEY. The XML document looks like this:

```
<Order key="1">
```

**XML document:** In the DAD file, create an attribute_node for `key` and indicate the table where the value of 1 is to be fetched.

**DAD file:**

```
<attribute_node name="key">
  <RDB_node>
    <table name="part_tab">
    <column name="part_key"/>
  <RDB_node>
</attribute_node>
```

8. Optional: Specify the column type for the RDB_node for each attribute_node and text_node. By doing so, you can use the DAD for both composition and decomposition. In composition, specifying the column type does not result in any additional data type checking To specify the column types, add the attribute type to the column element. The following example defines the column type as an INTEGER:

```
<attribute_node name="key">
    <RDB_node>
        <table name="order_tab"/>
        <column name="order_key" type="integer"/>
    </RDB_node>
</attribute_node>
```

9. Ensure that the ending tags are in the appropriate places:
   a. Ensure that an ending </root_node> tag is after the last </element_node> tag.
   b. Ensure that an ending </Xcollection> tag is after the </root_node> tag.
   c. Ensure that an ending </DAD> tag is after the </Xcollection> tag.

Call the dxxGenXML stored procedure to compose using the DAD. For information on dxxGenXML, see "dxxGenXML() stored procedure" on page 187.

**Related concepts:**

- "Mapping schemes for XML collections" on page 99
- "Requirements for RDB_Node mapping" on page 103
- "Location paths" on page 106
- "DAD files for XML collections" on page 163
- "XML Extender composition stored procedures - Overview" on page 186

**Related tasks:**

- "Decomposing an XML collection by using RDB_node mapping"
- "Managing data in XML collections" on page 88
- "Updating and deleting data in XML collections" on page 95

## Decomposing an XML collection by using RDB_node mapping

You must first create a DAD file. Use the <RDB_node> to specify DB2 database tables, columns, and conditions for an element or attribute node. The <RDB_node> uses the following elements:

**table** Defines the table that corresponds to the element.

**column** Defines the column that contains the corresponding element.

**condition** Optionally specifies a condition on which insertion into the column occurs.

The child elements that are used in the <RDB_node> depend on the context of the node and use the following rules:

| If the node type is: | RDB child element is used: | | |
|---|---|---|---|
| | **Table** | **Column** | **Condition** |
| Root element | Yes | No | Yes[1] |
| Attribute | Yes | Yes | optional |
| Text | Yes | Yes | optional |

[1] Required to show relationships between multiple tables.

**Procedure:**

To prepare a DAD file to use for decomposing an XML document using the RDB_node mapping:

1. Create a file in any text editor. Create a DAD header by typing the following syntax:

   ```
   <?xml version="1.0"?>
   <!DOCTYPE DAD SYSTEM "path/dad.dtd">
   ```

   Where *path*/dad.dtd is the path and file name of the DTD for the DAD.
2. Insert the <DAD> </DAD> tags.
3. Insert the tags used for validating the DAD with a DTD or schema.
   - To validate the XML document with a DTD, insert the dtdid tag that associates the DAD file with the XML document DTD. For example:

     ```
     <dtdid>dtd_name</dtdid>
     ```
   - To validate the XML document with a schema, insert the schemabinding tag that associates the DAD file with the schema file. For example:

     ```
     <schemabindings>
     <nonamespacelocation location="path/schema_name.xsd"/>
     </schemabindings>
     ```

   The dtdid or schema is useful only if you decide to validate the XML document. Use the validation tag to indicate whether DB2 XML Extender validates the XML document:
   - If you want to validate the XML document, then type:

```
<validation>YES</validation>
```

- If you do not want to validate the XML document type:

```
<validation>NO</validation>
```

4. Insert <XCollection> </XCollection> tags to specify that you are using XML collections as the access and storage method for your XML data.
5. Add the following prolog information. If you need to specify an encoding value for internationalization, add the ENCODING attribute and value.

```
<prolog>?xml version="1.0"?</prolog>
```

6. Add the <doctype></doctype> tags. For example:

```
<doctype>! DOCTYPE Order SYSTEM "dxx_install
   /samples/extenders/db2xml/dtd/getstart.dtd"</doctype>
```

7. Define the root_node using the <root_node> </root_node> tags.
8. Inside the root_node, map the elements and attributes in the XML document to element nodes and attribute nodes that correspond to DB2 database data. These nodes provide a path from the XML data to the DB2 database data.
   a. Define a top level, root element_node. This element_node contains:
      - Table elements with a join condition to specify the collection.
      - Child elements
      - Attributes

      To specify the table and condition:

      1) Create an RDB_node element within an element_node element. For example:

```
<element_node name="name">
<RDB_node>
</RDB_node>
</element_node>
```

      2) Define a <table> element for each table that contains data to be included in the XML document. For example, if you have three tables, ORDER_TAB, PART_TAB, and SHIP_TAB, that have column data to be in the document, create a table node for each. For example:

```
<RDB_node>
<table name="ORDER_TAB">
<table name="PART_TAB">
<table name="SHIP_TAB">
</RDB_node>
```

      3) Define a join condition for the tables in the collection. The syntax is:

```
table_name.table_column = table_name.table_column AND
table_name.table_column = table_name.table_column ...
```

      For example:

```
<RDB_node>
<table name="ORDER_TAB">
<table name="PART_TAB">
<table name="SHIP_TAB">
<condition>
   order_tab.order_key = part_tab.order_key AND
   part_tab.part_key = ship_tab.part_key
</condition>
</RDB_node>
```

      4) Optional: Specify a primary key for each table. The primary key consists of a single column or multiple columns, called a composite key. To specify the primary key, add an attribute key to the table

element of the RDB_node. The following example defines a primary key for each of the tables in the RDB_node of the root element_node Order:

```
<element_node name="Order">
    <RDB_node>
         <table name="order_tab" key="order_key"/>
         <table name="part_tab" key="part_key price"/>
         <table name="ship_tab" key="date mode"/>
         <condition>
             order_tab.order_key = part_tab.order_key AND
             part_tab.part_key = ship_tab.part_key
         </condition>
    <RDB_node>
```

The key attribute is required for decomposition and enabling a collection.

b. Define an <element_node> tag for each element in your XML document that maps to a column in a DB2 database table. For example:

```
<element_node name="name">
</element_node>
```

An element node can have one of the following types of elements:

**text_node** To specify that the element has content to a DB2 database table It does not have child elements.

**attribute_node**
To specify an attribute.

**child elements**
Children of the element_node.

The text_node contains an RDB_node to map content to a DB2 database table and column name.

For example, you might have an XML element <Tax> for which you want to store the untagged content into a column called TAX. For example, the XML document would look like this:

```
<Tax>0.02</Tax>
```

In this case, you want the value 0.02 to be stored in the column TAX.

In the DAD file, you specify an <RDB_node> tag to map an XML document or attribute values to the DB2 database table and column name.

**DAD file:**

```
<element_node name="Tax">
   <text_node>
      <RDB_node>
         <table name="part_tab"/>
         <column name="tax"/>
      </RDB_node>
   </text_node>
</element_node>
```

The <RDB_node> tag specifies that the value of the Tax element is a text value, the data is stored in the PART_TAB table in the TAX column.

c. Define an <attribute_node> tag for each attribute in your XML document that maps to a column in a DB2 database table. For example:

```
<attribute_node name="key">
</attribute_node>
```

The attribute_node has an RDB_node to map the attribute value to a DB2 database table and column.

For example, you might have an attribute key for an Order element. The value key needs to be stored in a column PART_KEY.

**XML document:**

```
<Order key="1">
```

In the DAD file, create an attribute_node for key and indicate the table where to store the value of 1.

**DAD file:**

```
<attribute_node name="key">
  <RDB_node>
    <table name="part_tab">
    <column name="part_key"/>
  <RDB_node>
</attribute_node>
```

9. Specify the column type for the RDB_node for each attribute_node and text_node. This ensures the correct data type for each column where the untagged data will be stored. To specify the column types, add the attribute type to the column element. The following example defines the column type as an INTEGER:

```
<attribute_node name="key">
    <RDB_node>
        <table name="order_tab"/>
        <column name="order_key" type="integer"/>
    </RDB_node>
</attribute_node>
```

10. Ensure that the ending tags are in the appropriate places:
    a. Ensure that an ending </root_node> tag is after the last </element_node> tag.
    b. Ensure that an ending </Xcollection> tag is after the </root_node> tag.
    c. Ensure that an ending </DAD> tag is after the </Xcollection> tag.

Call the dxxShredXML stored procedure to decompose using the DAD. For information on dxxShredXML, see "dxxShredXML() stored procedure" on page 197.

**Related concepts:**

- "XML Extender decomposition stored procedures - Overview" on page 197

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179
- "Decomposing XML documents into DB2 database data" on page 92

# Part 3. Programming

This part describes programming techniques for managing your XML data.

© Copyright IBM Corp. 1999 - 2006

# Chapter 3. XML columns

This section describes how to manage data in XML columns using DB2.

## Managing data in XML columns

When you use XML columns to store data, you store an entire XML document in its native format as column data in DB2. This access and storage method allows you to keep the XML document intact, while giving you the ability to index and search the document, retrieve data from the document, and update the document.

After you enable a database for XML, the following user-defined types (UDTs), provided by XML Extender, are available for your use:

**XMLCLOB**
Use this UDT for XML document content that is stored as a character large object (CLOB) in DB2.

**XMLVARCHAR**
Use this UDT for XML document content that is stored as a VARCHAR in DB2.

**XMLFILE**
Use this UDT for an XML document that is stored in a file on a local file system.

You can create or alter application tables to have columns of XML UDT data type. These tables are known as XML tables.

After you enable a column in a table for XML, you can create the XML column and perform the following management tasks:

- Store XML documents in DB2
- Retrieve XML data or documents from DB2
- Update XML documents
- Delete XML data or documents

To perform all of these tasks, use the user-defined functions (UDFs) provided by XML Extender. Use default casting functions to store XML documents in DB2. Default casting functions cast the SQL base type to the XML Extender user-defined types and convert instances of a data type (origin) into instances of a different data type (target).

**Related concepts:**


- "Using indexes for XML column data" on page 73
- "XML columns as a storage and access method"


## XML columns as a storage and access method

There will be times when you want to store and maintain the document structure as it currently is. XML contains all the necessary information to create a set of documents.

© Copyright IBM Corp. 1999 - 2006

For example, if you are a news publishing company that serves articles over the Web, you might want to maintain an archive of published articles. In such a scenario, XML Extender lets you store your complete or partial XML articles in a column of a DB2® table, which is the *XML column*, as shown in Figure 8.

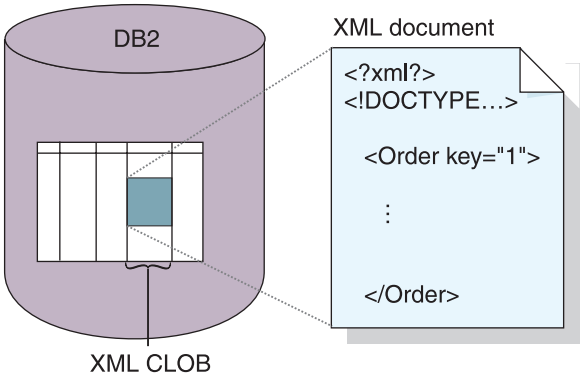



*Figure 8. Storing structured XML documents in a DB2 table column*

The XML column storage and access method allows you to manage your XML documents using DB2. You can store XML documents in a column of XML type and you can query the contents of the document to find a specific element or attribute. You can associate and store a DTD in DB2 for one or more documents. Additionally, you can map element and attribute content to DB2 tables, called *side tables*. These side tables can be indexed for improved query performance, but are not indexed automatically. The column that is used to store the document is called an XML column. It specifies that the column is used for the XML column storage and access method.

In the document access definition (DAD) file you enter <Xcolumn> and </Xcolumn> tags to denote that the storage and access method that you will use is XML column. The DAD will then map the XML element and attribute content to be stored in side tables.

Before you begin working with XML Extender to store your documents, you need to understand the structure of the XML document so that you can determine how to index elements and attributes in the document. When planning how to index the document, you need to determine:

- The XML user-defined type in which you will store the XML document
- The XML elements and attributes that your application will frequently search, so that their content can be stored in side tables and indexed to improve performance
- Whether or not you want to validate XML documents in the column with a DTD

## Defining and enabling an XML column

You use XML columns to store and access entire XML documents in the database. This storage method allows you to store documents using the XML file types, index the columns in side tables, and query or search XML documents.

Use XML columns when you want to store entire XML documents into a DB2 table column if the document is not going to be frequently updated or if you want to store intact XML documents.

If you want to map XML document structures to DB2 database tables so that you can compose XML documents from existing DB2 data or decompose XML documents into DB2 data, then you should use XML collections instead of XML columns.

**Procedure:**

To define and enable an XML column from the command line:

1. Create a document access definition (DAD) file.
2. Create a table in which the XML documents are stored.
3. Enable the column for XML data.
4. Index side tables.

The XML column is created as an XML user data type. After these tasks are complete, you will be able to store XML documents in the column. These documents can then be updated, searched, and extracted.

**Related concepts:**

- "Lesson: Storing an XML document in an XML column" on page 7
- "Using indexes for XML column data"
- "Validating XML documents automatically" on page 50
- "XML columns as a storage and access method" on page 71

**Related tasks:**

- "Creating an XML table" on page 51
- "Enabling XML columns" on page 53
- "Indexing side tables" on page 57
- "Managing data in XML columns" on page 71

## Using indexes for XML column data

An important planning decision when using XML columns, is whether to index the side tables for XML column documents. This decision should be made based on how often you need to access the data and how critical performance is during structural searches.

When using XML columns, which contain entire XML documents, you can create side tables to contain columns of XML element or attribute values, then create indexes on these columns. You must determine the elements and attributes for which you need to create the index.

XML column indexing allows frequently queried data of general data types (such as integer, decimal, or date) to be indexed using the native DB2® index support from the database engine. The XML Extender extracts the values of XML elements or attributes from XML documents and stores them in the side tables, allowing you to create indexes on these side tables. You can specify each column of a side table with a location path that identifies an XML element or attribute and an SQL data type.

The XML Extender automatically populates the side table when you store XML documents in the XML column.

For fast search, create indexes on these columns using the DB2 *B-tree indexing* technology. See the DB2 documentation for more information on *B-tree indexing*.

You must keep the following considerations in mind when creating an index:

- For elements or attributes in an XML document that have *multiple occurrences*, you must create a separate side table for each XML element or attribute with multiple occurrences due to the complex structure of XML documents.
- You can create multiple indexes on an XML column.
- You can associate side tables with the application table using the ROOT ID. **Recommendation:** Use the primary key of the application table as the ROOT ID if the primary key is not a composite key.

  If the single primary key does not exist in the application table, or for some reason you don't want to use it, XML Extender alters the application table to add a column DXXROOT_ID, which stores a unique ID that is created at the insertion time. All side tables have a DXXROOT_ID column with the unique ID. If the primary key is used as the ROOT ID, all side tables have a column with the same name and type as the primary key column in the application table, and the values of the primary keys are stored.
- If you enable an XML column for the DB2 Net Search Extender, you can also use the Net Search Extender's structural-text feature. The Net Search Extender has *"section search"* support, which extends the capability of a conventional full-text search by allowing search words to be matched within a specific document context that is specified by location paths. The *structural-text index* can be used with XML Extender's indexing on general SQL data types.

## Storing XML data

Using XML Extender, you can insert intact XML documents into an XML column. If you define side tables, XML Extender automatically updates these tables. When you store an XML document directly, XML Extender stores the base type as an XML type.

**Prerequisites:**

- Ensure that you created or updated the DAD file.
- Determine what data type to use when you store the document.
- Choose a method (casting functions or UDFs) for storing the data in the DB2® table.

Specify an SQL INSERT statement that specifies the XML table and column to contain the XML document.

The XML Extender provides two methods for storing XML documents: default casting functions and storage UDFs.

Table 7 shows when to use each method.

*Table 7. The XML Extender storage functions*

| If the DB2 database base type is ... | Store in DB2 database as ... | | | |
|---|---|---|---|---|
| | **XMLVARCHAR** | **XMLCLOB** | **XMLDBCLOB** | **XMLFILE** |
| VARCHAR | XMLVARCHAR() | N/A | N/A | XMLFile FromVarchar() |

*Table 7. The XML Extender storage functions (continued)*

| If the DB2 database base type is ... | Store in DB2 database as ... | | | |
|---|---|---|---|---|
| | **XMLVARCHAR** | **XMLCLOB** | **XMLDBCLOB** | **XMLFILE** |
| CLOB | N/A | XMLCLOB() | XMLDB CLOB, casting function | XMLFile FromCLOB() |
| FILE | XMLVarcha rFromFile() | XMLCLOB FromFile() | XMLDB CLOBFrom File, UDF | XMLFILE |

## Default casting functions for storing XML data

For each UDT, a default casting function exists to cast the SQL base type to the UDT. You can use the casting functions provided by XML Extender in your VALUES clause to insert data. Table 8 shows the provided casting functions:

*Table 8. The XML Extender default casting functions*

| Casting function | Return type | Description |
|---|---|---|
| XMLVARCHAR(VARCHAR) | XMLVARCHAR | Input from memory buffer of VARCHAR |
| XMLCLOB(CLOB) | XMLCLOB | Input from memory buffer of CLOB or a CLOB locator |
| XMLFILE(VARCHAR) | XMLFILE | Store only the file name |

For example, the following statement inserts a cast VARCHAR type into the XMLVARCHAR type:

```
EXEC SQL INSERT INTO sales_tab
VALUES('123456', 'Sriram Srinivasan', DB2XML.XMLVarchar(:xml_buff))
```

## Storage UDFs for storing XML data

For each XML Extender UDT, a storage UDF exists to import data into DB2 from a resource other than its base type. For example, if you want to import an XML file document to DB2 database as an XMLCLOB data type, you can use the function XMLCLOBFromFile().

Table 9 shows the storage functions provided by the XML Extender.

*Table 9. The XML Extender storage UDFs*

| Storage user-defined function | Return type | Description |
|---|---|---|
| XMLVarcharFromFile() | XMLVARCHAR | Reads an XML document from a file on the server and returns the value of the XMLVARCHAR data type. Optional: Specify the encoding of the file. |

*Table 9. The XML Extender storage UDFs (continued)*

| Storage user-defined function | Return type | Description |
|---|---|---|
| XMLCLOBFromFile() | XMLCLOB | Reads an XML document from a file on the server and returns the value of the XMLCLOB data type. Optional: Specify the encoding of the file. |
| XMLFileFromVarchar() | XMLFILE | Reads an XML document from memory as VARCHAR data, writes the document to an external file, and returns the value of the XMLFILE data type, which is the file name. Optional: Specify the encoding of the external file. |
| XMLFileFromCLOB() | XMLFILE | Reads an XML document from memory as CLOB data or as a CLOB locator, writes the document to an external file, and returns the value of the XMLFILE data type, which is the file name. Optional: Specify the encoding of the external file. |

For example, using the XMLCLOBFromFile() function, the following statement stores a record in an XML table as an XMLCLOB:

```
EXEC SQL INSERT INTO sales_tab(ID, NAME, ORDER)
    VALUES('1234', 'MyName',
      XMLCLOBFromFile('dxx_install/samples/extenders/db2xml/xml/getstart.xml'))
```

The example imports the XML document from the file named `dxx_install/samples/extenders/db2xml/xml/getstart.xml` into the column ORDER in the table SALES_TAB.

## Method for retrieving an XML document

Using XML Extender, you can retrieve either an entire document or the contents of elements and attributes. When you retrieve an XML column directly, the XML Extender returns the UDT as the column type. For details about retrieving data, see the following sections:

- "Retrieving an entire XML document" on page 77
- "Retrieving element contents and attribute values from XML documents" on page 79

The XML Extender provides two methods for retrieving data: default casting functions and the Content() overloaded UDF. Table 10 on page 77 shows when to use each method.

*Table 10. The XML Extender retrieval functions*

| When the XML type is ... | Retrieve from DB2 database as ... | | |
|---|---|---|---|
| | **VARCHAR** | **CLOB** | **FILE** |
| XMLVARCHAR | **VARCHAR** | N/A | Content() UDF |
| XMLCLOB | N/A | **XMLCLOB** | Content() UDF |
| XMLFILE | N/A | Content() UDF | FILE |

# Retrieving an entire XML document

**Procedure:**

To retrieve an entire XML document:

1. Ensure that you stored the XML document in an XML table and determine what data you want to retrieve.
2. Choose a method (casting functions or UDFs) for retrieving the data in the DB2 database table.
3. If you are using the overloaded Content() UDF, determine the data type of the data that is being retrieved, and which data type is to be exported.
4. The XML column from which the element or attribute is to be extracted must be defined as either an XMLVARCHAR, XMLCLOB as LOCATOR, or XMLFILE data type.

Specify an SQL query that specifies the XML table and column from which to retrieve the XML document.

## Default casting functions for retrieving XML data

The default casting function provided by DB2 database for UDTs converts an XML UDT to an SQL base type, and then operates on it. In your SELECT statement, you can use the casting functions that are provided by XML Extender to retrieve data. Table 11 shows the provided casting functions.

*Table 11. The XML Extender default cast functions*

| Casting used in SELECT clause | Return type | Description |
|---|---|---|
| varchar(XMLVARCHAR) | VARCHAR | XML document in VARCHAR |
| clob(XMLCLOB) | CLOB | XML document in CLOB |
| varchar(XMLFile) | VARCHAR | XML file name in VARCHAR |

For example, the following SQL statement retrieves the XMLVARCHAR column, order, from the SALES_TAB table, casts the column to VARCHAR, and finally passes the column as a parameter to the LENGTH function. The LENGTH function does not accept a parameter of type XMLVARCHAR, which requires casting.

```
SELECT DB2XML.XMLVarchar(order) from SALES_TAB
```

## Using the Content() UDF for retrieving XML data

Use the Content() UDF to retrieve the document content from external storage to memory, or export the document from internal storage to an external file, which is a file that is external to DB2 database on the DB2 database server.

For example, you might have your XML document stored as an XMLFILE data type. If you want to operate on it in memory, you can use the Content() UDF, which can take an XMLFILE data type as input and return a CLOB.

The Content() UDF performs two different retrieval functions, depending on the specified data type. It can:

- Retrieve a document from external storage and put it in memory.

  You can use Content() UDF to retrieve the XML document to a memory buffer or a CLOB *locator* (a host variable with a value that represents a single LOB value in the database server) when the document is stored as the external file.

  Use the following function syntax, where *xmlobj* is the XML column being queried:

  **XMLFILE to CLOB:**

  ```
  Content(xmlobj XMLFile)
  ```

- Retrieve a document from internal storage and export it to an external file.

  You can use the Content() UDF to retrieve an XML document that is stored inside DB2 database as an XMLCLOB data type and export it to a file on the database server file system. The Content() UDF returns the name of the file as a VARCHAR data type.

  Use the following function syntax:

  **XML type to external file:**

  ```
  Content(xmlobj XML type, filename varchar(512), targetencoding varchar(100))
  ```

  Where:

  *xmlobj* Is the name of the XML column from which the XML content is to be retrieved. *xmlobj* can be of type XMLVARCHAR or XMLCLOB.

  *filename*
  Is the name of the external file in which the XML data is to be stored.

  *targetencoding*
  Optional: Specifies the encoding of the output file.

In the example below, a small C program segment with embedded SQL statements (SQL statements coded within an application program) shows how an XML document is retrieved from a file to memory. This example assumes that the data type of the ORDER column is XMLFILE.

```
EXEC SQL BEGIN DECLARE SECTION;
     SQL TYPE IS CLOB_LOCATOR xml_buff;
EXEC SQL END DECLARE SECTION;
EXEC SQL CONNECT TO SALES_DB;
EXEC SQL DECLARE c1 CURSOR FOR
         SELECT Content(order) from sales_tab
EXEC SQL OPEN c1;
    do {
    EXEC SQL FETCH c1 INTO :xml_buff;
       if (SQLCODE != 0) {
          break;}
       else { /* do whatever you need to do with the XML doc in buffer */}
       }
    EXEC SQL CLOSE c1;
    EXEC SQL CONNECT RESET;
```

## Retrieving element contents and attribute values from XML documents

You can retrieve (extract) the content of an element or the value of an attribute from one or more XML documents (single document or collection document search). The XML Extender provides user-defined extracting functions that you can specify in the SQL SELECT clause for each of the SQL data types.

Retrieving element content and attribute values is useful when developing your applications, because you can access XML data as relational data. For example, you might have 1000 XML documents that are stored in the ORDER column in the SALES_TAB table. To retrieve the names of all customers who have ordered items over $2500, use the following SQL statement with the extracting UDF in the SELECT clause:

```
SELECT extractVarchar(Order, '/Order/Customer/Name') from sales_order_view
        WHERE price > 2500.00
```

In this example, the extracting UDF retrieves the content of the <customer> element from the ORDER column and stores it as a VARCHAR data type. The location path is `/Order/Customer/Name`. Additionally, the number of returned values is reduced by using a WHERE clause, which specifies that only the contents of the <customer> element with a sub-element <ExtendedPrice> that has a value greater than 2500.00.

Table 12 on page 80 shows the UDFs that you can use to extract element content and attribute values, using the following syntax as either table or scalar functions.

**Syntax:**

`extract` *`retrieved_datatype`*`(`*`xmlobj`*`,` *`path`*`)`

*retrieved_datatype*
: The data type that is returned from the extracting function; it can be one of the following types:
  - INTEGER
  - SMALLINT
  - DOUBLE
  - REAL
  - CHAR
  - VARCHAR
  - CLOB
  - DATE
  - TIME
  - TIMESTAMP

*xmlobj*
: The name of the XML column from which the element or attribute is to be extracted. This column must be defined as one of the following XML user-defined types:
  - XMLVARCHAR
  - XMLCLOB as LOCATOR
  - XMLFILE

*path*
: The location path of the element or attribute in the XML document (such as `/Order/Customer/Name`).

**Restriction:** Extracting UDFs can support location paths that have predicates with attributes, but not elements. For example, the following predicate is supported:

```
'/Order/Part[@color="black "]/ExtendedPrice'
```

The following predicate is not supported:

```
'/Order/Part/Shipment/[Shipdate < "11/25/00"]'
```

Table 12 shows the extracting functions, both in scalar and table format.

*Table 12. The XML Extender extracting functions*

| Scalar function | Table function | Returned column name (table function) | Return type |
|---|---|---|---|
| extractInteger() | extractIntegers() | returnedInteger | INTEGER |
| extractSmallint() | extractSmallints() | returnedSmallint | SMALLINT |
| extractDouble() | extractDoubles() | returnedDouble | DOUBLE |
| extractReal() | extractReals() | returnedReal | REAL |
| extractChar() | extractChars() | returnedChar | CHAR |
| extractVarchar() | extractVarchars() | returnedVarchar | VARCHAR |
| extractCLOB() | extractCLOBs() | returnedCLOB | CLOB |
| extractDate() | extractDates() | returnedDate | DATE |
| extractTime() | extractTimes() | returnedTime | TIME |
| extractTimestamp() | extractTimestamps() | returnedTimestamp | TIMESTAMP |

**Scalar function example:** In the following example, one value is inserted with the attribute key value of 1. The value is extracted as an integer and automatically converted to a DECIMAL type.

```
CREATE TABLE t1(key decimal(3,2));
INSERT into t1 values
SELECT * from table(DB2XML.extractInteger(DB2XML.XMLFile
   ('c:\xml\getstart.xml'), '/Order/@Key="1"']'));
SELECT * from t1;
```

**Table function example:** In the following example, each key value (@Key) for the sales order is extracted as an INTEGER.

```
SELECT * from table(DB2XML.extractIntegers(DB2XML.XMLFile
   ('c:\dxx_install\samples\extenders\db2xml\xml\getstart.xml'),
   '/Order/@Key')) as x;
```

## Updating XML data

With XML Extender, you can update the entire XML document by replacing the XML column data, or you can update the values of specified elements or attributes.

**Procedure**

To update XML data:

1. The XML document must be stored in an XML table.
2. You must know what data you want to retrieve.
3. You must choose a method for updating the data in the DB2 database table (casting functions or UDFs).
4. Specify an SQL query that specifies the XML table and column to update.

## Updating an entire XML document

You can update an XML document by using a default casting function, or by using a storage UDF.

### Updating with a default casting function

For each user-defined type (UDT), a default casting function exists to cast the SQL base type to the UDT. You can use the XML Extender-provided casting functions to update the XML document.

For example, the following statement updates the XMLVARCHAR type from the cast VARCHAR type, assuming that `xml_buf` is a host variable that is defined as a VARCHAR type.

```
EXEC SQL UPDATE sales_tab SET=DB2XML.XMLVarchar(:xml_buff)
```

### Updating XML documents with a storage UDF

For each of the XML Extender UDTs, a storage UDF exists to import data into DB2 database from a resource other than its base type. You can use a storage UDF to update the entire XML document by replacing it.

The following example updates the XML object from the file named `dxx_install/samples/extenders/db2xml/xml/getstart.xml` to the ORDER column in the SALES_TAB table.

```
UPDATE sales_tab
       set order = db2xml.XMLVarcharFromFile('
dxx_install/samples/extenders/db2xml/xml/getstart.xml
')
WHERE sales_person = 'MyName'
```

## Updating specific elements and attributes of an XML document

Use the Update UDF to make specific changes, rather than updating the entire document. When you use this UDF, you specify the location path of the element or attribute whose value will be replaced. You do not need to edit the XML document; XML Extender makes the change for you.

**Syntax:**

```
Update(xmlobj, path, value)
```

The syntax has the following components:

*xmlobj* The name of the XML column for which the value of the element or attribute is to be updated.

*path* The location path of the element or attribute that is to be updated.

*value* The new value that is to be updated.

For example, the following statement replaces the value of the <Customer> element with IBM:

```
UPDATE sales_tab
       set order = db2xml.Update(order, '/Order/Customer/Name', 'IBM')
   WHERE sales_person = 'Sriram Srinivasan'
```

**Multiple occurrence:** When you specify a location path in the Update UDF, the content of every element or attribute with a matching path is updated with the

supplied value. If a location path occurs in a document more than once, the Update UDF replaces all of the existing values with the value provided in the *value* parameter.

## Methods for searching XML documents

Searching XML data is similar to retrieving XML data: both techniques retrieve data for further manipulation but they search by using the content of the WHERE clause as the criteria for retrieval.

The XML Extender provides several methods for searching XML documents that are stored in an XML column. You can:

- Search document structure and return results based on element content or attribute values.
- Search a view of the XML column and its side tables.
- Search the side tables directly for better performance.
- Search using extracting UDFs with WHERE clauses.
- Use the DB2® Net Search Extender to search column data within the structural content for a text string.

With XML Extender you can use indexes to quickly search columns in side tables. These columns contain XML element content or attribute values that are extracted from XML documents. For example, in the purchase order example, you could search for all orders that have an extended price of over 2500.00.

Additionally, you can use the Net Search Extender to do structural text search or full text search. For example, you might have a column called RESUME that contains resumes in XML format. If you want to find the names of all applicants who have Java™ skills, you could use the DB2 Net Search Extender to search on the XML documents for all resumes where the <skill> element contains the character string "JAVA". For more information, see *DB2 Database Extenders: Net Search Extender Administration and Programming*.

The following section describes search methods:

- "Searching the XML document by structure"

### Searching the XML document by structure

Using XML Extender search features, you can search XML data in a column based on the document structure (the elements and attributes in the document).

**Procedures:**

To search the data, you can:

- Directly query the side tables.
- Use a *joined view.*
- Use extracting UDFs.

These search methods are described in the following examples. The SALES_TAB table has an XML column named ORDER. This column has three side tables, ORDER_SIDE_TAB, PART_SIDE_TAB, and SHIP_SIDE_TAB. A default view, sales_order_view, was specified when the ORDER column was enabled. This view joins these tables using the following CREATE VIEW statement:

```
CREATE VIEW sales_order_view(invoice_num, sales_person, order,
                             order_key, customer, part_key, price, date)
     AS
     SELECT sales_tab.invoice_num, sales_tab.sales_person, sales_tab.order,
            order_side_tab.order_key, order_side_tab.customer,
            part_side_tab.part_key, ship_side_tab.date
     FROM sales_tab, order_side_tab, part_side_tab, ship_side_tab
     WHERE sales_tab.invoice_num = order_side_tab.invoice_num
               AND sales_tab.invoice_num = part_side_tab.invoice_num
               AND sales_tab.invoice_num = ship_side_tab.invoice_num
```

## Example: searching with direct query on side tables

Direct query with subquery search provides the best performance for a structural search when the side tables are indexed.

**Procedure:**

You can use a query or subquery to search side tables correctly.

For example, the following statement uses a query and subquery to directly search a side table:

```
SELECT sales_person from sales_tab
   WHERE invoice_num in
     (SELECT invoice_num from part_side_tab
      WHERE price > 2500.00)
```

In this example, invoice_num is the primary key in the SALES_TAB table.

## Example: searching from a joined view

The XML Extender can create a default view that joins the application table and the side tables using a unique ID. You can use this default view, or any view that joins an application table and side tables, to search column data and query the side tables. This method provides a single virtual view of the application table and its side tables. However, the more side tables that are created, the longer the query takes to run.

**Tip:** You can use the root ID, or DXXROOT_ID (created by XML Extender), to join the tables when you create your own view.

For example, the following statement searches the view named SALES_ORDER_VIEW and returns the values from the SALES_PERSON column where the line item orders have a price greater than 2500.00.

```
SELECT sales_person from sales_order_view
   WHERE price > 2500.00
```

## Example: searching with extracting UDFs

You can also use XML Extender's extracting UDFs to search on elements and attributes, when you did not create indexes or side tables for the application table. Using the extracting UDFs to scan the XML data is expensive and should only be used with WHERE clauses that restrict the number of XML documents that are included in the search.

The following statement searches with an extracting XML Extender UDF:

```
SELECT sales_person from sales_tab
     WHERE extractVarchar(order, '/Order/Customer/Name')
      like '%IBM%'
    AND invoice_num > 100
```

In this example, the extracting UDF extracts </Order/Customer/Name> elements that contain the substring IBM®.

### Example: searching on elements or attributes with multiple occurrence

When you search on elements or attributes that have multiple occurrence, use the DISTINCT clause to prevent duplicate values.

The following statement searches with the DISTINCT clause:

```
SELECT sales_person from sales_tab
            WHERE invoice_num in
              (SELECT DISTINCT invoice_num from part_side_tab
               WHERE price > 2500.00 )
```

In this example, the DAD file specifies that `/Order/Part/Price` has multiple occurrence and creates a side table, PART_SIDE_TAB, for it. The PART_SIDE_TAB table might have more than one row with the same invoice_num. Using DISTINCT returns only unique values.

## Deleting XML documents

Use the SQL DELETE statement to delete the row containing an XML document from an XML column. You can specify a WHERE clause to delete specific documents.

For example, the following statement deletes all documents that have a value for <ExtendedPrice> greater than 2500.00:

```
DELETE from sales_tab
       WHERE invoice_num in
          (SELECT invoice_num from part_side_tab
           WHERE price > 2500.00)
```

The corresponding rows in the side tables are automatically deleted.

## Limitations when invoking functions from Java Database (JDBC)

When using parameter markers in functions, a JDBC restriction requires that the parameter marker for the function must be cast to the data type of the column into which the returned data will be inserted. The function selection logic does not know what data type the argument might turn out to be, and it cannot resolve the reference.

For example, JDBC cannot resolve the following code:

```
DB2XML.XMLdefault_casting_function(data_type)
```

You can use the CAST specification to provide a type for the parameter marker, such as VARCHAR, and then the function selection logic can proceed:

```
DB2XML.XMLdefault_casting_function(CAST(? AS cast_type(length))
```

**Examples:**

In the following examples, The `Sales_Tab` table has three columns.
The `invoice_num` column has a data type of Char(6), the `sales_person` column has a data type of Varchar(20), and the `order` column has a data type of XMLVarchar.

**Example 1:** In the following example, the parameter marker is cast as VARCHAR. The parameter being passed is an XML document, which is cast as VARCHAR(1000) and inserted into the column ORDER.

```
String query = "insert into sales_tab(invoice_num, sales_person, order) values
    (?,?,DB2XML.XMLVarchar(cast (? as varchar(1000))))";
```

**Example 2:** In the following example, the parameter marker is cast as VARCHAR. The parameter being passed is a file name and its contents are converted to VARCHAR and inserted into the column ORDER.

```
String query = "insert into sales_tab(invoice_num, sales_person, order) values
    (?,?,DB2XML.XMLVarcharfromFILE(cast (? as varchar(1000))))";
```

# Chapter 4. Managing data in XML collections

## XML collections as a storage and access method

Relational data is either *decomposed* from incoming XML documents or used to *compose* outgoing XML documents. Decomposed data is the untagged content of an XML document stored in one or more database tables. Or, XML documents are composed from existing data in one or more database tables. If your data is to be shared with other applications, you might want to be able to compose and decompose incoming and outgoing XML documents and manage the data as necessary to take advantage of the relational capabilities of DB2®. This type of XML document storage is called *XML collection*.

An example of an XML collection is shown in Figure 9.

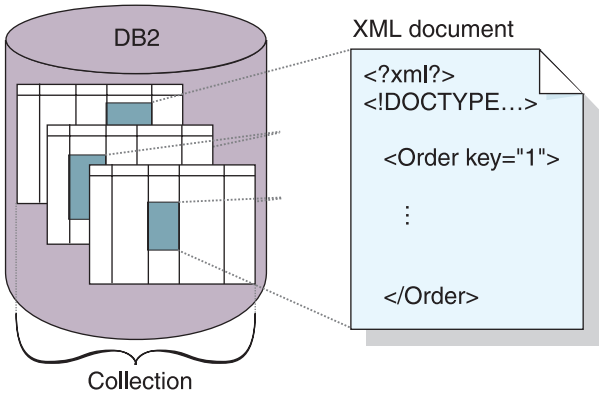



*Figure 9. Storing documents as untagged data in DB2 tables*

The XML collection is defined in a DAD file, which specifies how elements and attributes are mapped to one or more relational tables. The collection is a set of columns, associated with a DAD file, that contain the data in a particular XML document or set of XML documents. You can define a collection name by enabling it, and then refer to it by name when issuing a stored procedure to compose or decompose XML documents. It is called an enabled XML collection. The collection is given a name so that it is easily run with stored procedures that compose and decompose the XML documents.

When you define a collection in the DAD file, you use one of two types of mapping schemes, *SQL mapping* or *RDB_node mapping* that define the tables, columns, and conditions used to associate XML data with DB2 tables. SQL mapping uses SQL SELECT statements to define the DB2 tables and conditions used for the collection. RDB_node mapping uses an XPath-based relational database node, or RDB_node, which has child elements.

Stored procedures are provided to compose or decompose XML documents. Stored procedure names are qualified by DB2XML, which is the *schema name* of XML Extender.

© Copyright IBM Corp. 1999 - 2006

# Managing data in XML collections

An XML collection is a set of relational tables that contain data that is mapped to XML documents. This access and storage method lets you compose an XML document from existing data, decompose an XML document, and use XML as an interchange method.

The relational tables that make up the collection can be new tables, or existing tables that have data that is to be used with XML Extender to compose XML documents for your applications. Column data in these tables does not contain XML tags; it contains the content and values that are associated with elements and attributes, respectively. You use stored procedures to store, retrieve, update, search, and delete XML collection data.

## Preparing to compose XML documents from DB2 data

Composition is the generation of a set of XML documents from relational data in an XML collection. You can compose XML documents using stored procedures. To use these stored procedures, create a document access definition (DAD) file. A DAD file specifies the mapping between the XML document and the DB2 table structure. The stored procedures use the DAD file to compose the XML document.

**Procedure::**

Before you begin composing XML documents:

1. Map the structure of the XML document to the relational tables that contain the contents of the element and attribute values.
2. Select a mapping method: SQL mapping or RDB_node mapping.
3. Prepare the DAD file.

The XML Extender provides four stored procedures to compose XML documents:

- dxxGenXML()
- dxxGenXMLCLOB()
- dxxRetrieveXML()
- dxxRetrieveXMLCLOB

### Composing documents using dxxGenXML

Use the dxxGenXML stored procedure to compose the document. You do not have to enable a collection to use this stored procedure. The stored procedure uses a DAD file instead.

The dxxGenXML stored procedure constructs XML documents using data that is stored in XML collection tables, which are specified by the <Xcollection> element in the DAD file. This stored procedure inserts each XML document as a row into a result table. You can open a cursor on the result table and fetch the result set. The result table must be created by the application and always has one column of VARCHAR, CLOB, XMLVARCHAR, or XMLCLOB type that is used to store the XML data.

Additionally, if the value of the validation element in the DAD file is YES, XML Extender adds the column DXX_VALID of INTEGER type into the result table if the DXX_VALID column is not in the table yet. The XML Extender inserts a value of 1 for a valid XML document and 0 for an invalid document.

The stored procedure dxxGenXML lets you specify the maximum number of rows that are to be generated in the result. Specifying a maximum number of rows can shorten processing time. The stored procedure returns the actual number of rows in the table and any return codes and messages.

The corresponding stored procedure for decomposition is dxxShredXML; it also takes the DAD as the input parameter and does not require that the XML collection be enabled.

**Procedure:**

To compose an XML document using the dxxGenXML stored procedure, embed a stored procedure call in your application using the following stored procedure declaration:

```
dxxGenXML(CLOB(100K)     DAD,              /* input */
          char(32 resultTabName) resultTabName,     /* input */

          integer        overrideType,      /* input */
          varchar(varchar_value)  override,          /* input */
          integer        maxRows,           /* input */
          integer        numRows,           /* output */
          long           returnCode,        /* output */
          varchar(1024)  returnMsg)         /* output */
```

**Example:** The following example composes an XML document:

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
     SQL TYPE is CLOB(100K) dad;              /* DAD */
     SQL TYPE is CLOB_FILE  dadFile;          /* dad file */
     char                   result_tab[32];   /* name of the result table */
     char                   override[2];      /* override, will set to NULL*/
     short                  overrideType;     /* defined in dxx.h */
     short                  max_row;          /* maximum number of rows */
     short                  num_row;          /* actual number of rows */
     long                   returnCode;       /* return error code */
     char                   returnMsg[1024];  /* error message text */
     short                  dad_ind;
     short                  rtab_ind;
     short                  ovtype_ind;
     short                  ov_ind;
     short                  maxrow_ind;
     short                  numrow_ind;
     short                  returnCode_ind;
     short                  returnMsg_ind;

     EXEC SQL END DECLARE SECTION;

     /* create table */
     EXEC SQL CREATE TABLE xml_order_tab (xmlorder XMLVarchar);

     /* read data from a file to a CLOB */



strcpy(dadfile.name,"dxx_install
/samples/dad/getstart_xcollection.dad");
dadfile.name_length = strlen("dxx_install
/samples/dad/getstart_xcollection.dad");
dadfile.file_options = SQL_FILE_READ;
```

```
strcpy(dadfile.name,"dxx_install
/samples/extenders/db2xml/samples/dad/getstart_xcollection.dad");
dadfile.name_length = strlen("dxx_install
/samples/extenders/db2xml/samples/dad/getstart_xcollection.dad");

dadfile.file_options = SQL_FILE_READ;
EXEC SQL VALUES (:dadfile) INTO :dad;
strcpy(result_tab,"xml_order_tab");
override[0] = '\0';
overrideType = NO_OVERRIDE;
max_row = 500;
num_row = 0;
returnCode = 0;
msg_txt[0] = '\0';
dad_ind = 0;
rtab_ind = 0;
ov_ind = -1;
ovtype_ind = 0;
maxrow_ind = 0;
numrow_ind = -1;
returnCode_ind = -1;
returnMsg_ind = -1;

/* Call the stored procedure */
EXEC SQL CALL db2xml.dxxGenXML(:dad:dad_ind;
     :result_tab:rtab_ind,
     :overrideType:ovtype_ind,:override:ov_ind,
     :max_row:maxrow_ind,:num_row:numrow_ind,
     :returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

After the stored procedure is called, the result table contains 250 rows because the SQL query specified in the DAD file generated 250 XML documents.

## Composing documents using dxxRetrieveXML

The dxxRetrieveXML stored procedure works in the same way as the dxxGenXML stored procedure, except that it takes the name of an enabled XML collection instead of a DAD file. When an XML collection is enabled, a DAD file is stored in the XML_USAGE table. Therefore, XML Extender retrieves the DAD file and uses it to compose the document in the same way as the dxxGenXML stored procedure.

The corresponding stored procedure for decomposition is dxxInsertXML; it also takes the name of an enabled XML collection.

**Procedure:**

To compose an XML collection using the dxxRetrieveXML stored procedure, embed a stored procedure call in your application using the following stored procedure declaration:

```
dxxRetrieveXML(
   varchar(collectionName) collectionName /*input*/
   varchar(resultTabName) resultTable /*input*/
   varchar(resultColumn) resultColumn             /*input*/
   varchar(validColumn) validColumn               /*input*/
   integer overrideType                           /*input*/
   varchar(varchar_value) override                /*input*/
   integer maxRows                                /*input*/
   integer numRows                                /*output*/
   integer returnCode                             /*output*/
   varchar(1024) returnMsg                        /*output*/
```

**Example:** The following example is of a call to dxxRetrieveXML(). It assumes that a result table is created with the name of XML_ORDER_TAB and that the table has one column of XMLVARCHAR type.

```
#include "dxx.h"
#include "dxxrc.h"

            EXEC SQL INCLUDE SQLCA;
            EXEC SQL BEGIN DECLARE SECTION;
              char    collection;   /* dad buffer */
              char    result_tab[32];   /* name of the result table */
              char    override[2];      /* override, will set to NULL*/
              short   overrideType;     /* defined in dxx.h */
              short   max_row;          /* maximum number of rows */
              short   num_row;          /* actual number of rows */
              long    returnCode;       /* return error code */
              char    returnMsg[1024];  /* error message text */
              short   collection_ind;
              short   rtab_ind;
              short   ovtype_ind;
              short   ov_ind;
              short   maxrow_ind;
              short   numrow_ind;
              short   returnCode_ind;
              short   returnMsg_ind;

              EXEC SQL END DECLARE SECTION;

              /* create table */
              EXEC SQL CREATE TABLE xml_order_tab (xmlorder XMLVarchar);

              /* initialize host variable and indicators */
              strcpy(collection,"sales_ord");
              strcpy(result_tab,"xml_order_tab");
              override[0] = '\0';
              overrideType = NO_OVERRIDE;
              max_row = 500;
              num_row = 0;
              returnCode = 0;
              msg_txt[0] = '\0';
              collection_ind = 0;
              rtab_ind = 0;
              ov_ind = -1;
              ovtype_ind = 0;
              maxrow_ind = 0;
              numrow_ind = -1;
              returnCode_ind = -1;
              returnMsg_ind = -1;

              /* Call the stored procedure */
              EXEC SQL CALL db2xml.dxxRetrieveXML(:collection:collection_ind;

                     :result_tab:rtab_ind,
                     :result_colname:rcol_ind,
                     :valid_colname:vcol_ind,
                     :overrideType:ovtype_ind,:override:ov_ind,
                     :max_row:maxrow_ind,:num_row:numrow_ind,
                     :returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

**Related concepts:**

- "XML Extender composition stored procedures - Overview" on page 186
- "DAD files for XML collections" on page 163
- "Location paths" on page 106
- "Mapping schemes for XML collections" on page 99

- "XML collections as a storage and access method" on page 87

**Related tasks:**
- "Composing XML collections by using RDB_node mapping" on page 61
- "Decomposing an XML collection by using RDB_node mapping" on page 65
- "Searching XML collections" on page 97
- "Stylesheets for an XML collection" on page 106
- "Updating and deleting data in XML collections" on page 95

## Decomposing XML documents into DB2 database data

To decompose an XML document is to parse down the data inside of an XML document and store it in relational tables. The XML Extender provides stored procedures to decompose XML data from source XML documents into relational tables. To use these stored procedures, you must create a DAD file, which specifies the mapping between the XML document and DB2 database table structure. The stored procedures use the DAD file to decompose the XML document.

Before you decompose an XML document into DB2 database data:
1. Map the structure of the XML document to the relational tables that contain the contents of the elements and attributes values.
2. Prepare the DAD file, using RDB_node mapping.
3. Optional: Enable the XML collection.

**Procedure::**

Use one of the two stored procedures provided by DB2 XML Extender to decompose XML documents, dxxShredXML() or dxxInsertXML.

**dxxShredXML()**

This stored procedure is used for applications that do occasional updates or for applications that do not want the overhead of administering the XML data. The stored procedure dxxShredXML() does not required an enabled collection; it uses a DAD file instead.

The stored procedure dxxShredXML() takes two input parameters, a DAD file and the XML document that is to be decomposed; it returns two output parameters: a return code and a return message. It inserts data from an XML document into an XML collection according to the <Xcollection> specification in the input DAD file. The dxxShredXML() stored procedure then decomposes the XML document, and inserts untagged XML data into the tables specified in the DAD file. The tables that are used in the <Xcollection> of the DAD file are assumed to exist, and the columns are assumed to meet the data types specified in the DAD mapping. If this is not true, an error message is returned.

The corresponding stored procedure for composition is dxxGenXML(); it also takes the DAD as the input parameter and does not require that the XML collection be enabled.

**To decompose an XML collection with dxxShredXML()**

Embed a stored procedure call in your application using the following stored procedure declaration:

```
dxxShredXML(CLOB(100K)    DAD,          /* input */
            CLOB(1M)      xmlobj,       /* input */
            long          returnCode,   /* output */
            varchar(1024) returnMsg)    /* output */
```

**Example:** The following example is a call to dxxShredXML():

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      SQL TYPE is CLOB(100K) dad;       /* DAD*/
      SQL TYPE is CLOB_FILE dadFile;    /* DAD file*/
      SQL TYPE is CLOB(1M)  xmlDoc;     /* input XML document */
      SQL TYPE is CLOB_FILE xmlFile;    /* input XMLfile */
      long               returnCode;    /* error code */
      char               returnMsg[1024]; /* error message text */
      short              dad_ind;
      short              xmlDoc_ind;
      short              returnCode_ind;
      short              returnMsg_ind;
     EXEC SQL END DECLARE SECTION;

        /* initialize host variable and indicators */
        strcpy(dadFile.name,
"dxx_install/samples/extenders/db2xml/dad/
        getstart_xcollection.dad");
        dadFile.name_length=strlen("dxx_install
       /samples/extenders/db2xml/dad/getstart_xcollection.dad");
       dadFile.file_option=SQL_FILE_READ;
       strcpy(xmlFile.name,"dxx_install
       /samples/extenders/db2xml/xml/getstart_xcollection.xml");
       xmlFile.name_length=strlen
      ("dxx_install/samples/extenders/db2xml/xml
      /getstart_xcollection.xml");



        xmlFile.file_option=SQL_FILE_READ;
        SQL EXEC VALUES (:dadFile) INTO :dad;
        SQL EXEC VALUES (:xmlFile) INTO :xmlDoc;
        returnCode = 0;
        returnMsg[0] = '\0';
        dad_ind = 0;
        xmlDoc_ind = 0;
        returnCode_ind = -1;
        returnMsg_ind = -1;

        /* Call the stored procedure */
        EXEC SQL CALL db2xml.dxxShredXML(:dad:dad_ind;
                    :xmlDoc:xmlDoc_ind,
                    :returnCode:returnCode_ind,
                    :returnMsg:returnMsg_ind);
```

**dxxInsertXML()**

This stored procedure is used for applications that make regular updates. The stored procedure dxxInsertXML() works the same as dxxShredXML(), except that dxxInsertXML() takes an enabled XML collection as its first input parameter.

The stored procedure dxxInsertXML() inserts data from an XML document into an enabled XML collection, which is associated with a DAD file. The DAD file contains specifications for the collection tables and the mapping. The collection tables are checked or created according to the specifications

in the <Xcollection>. The stored procedure dxxInsertXML() then decomposes the XML document according to the mapping, and it inserts untagged XML data into the tables of the named XML collection.

The corresponding stored procedure for composition is dxxRetrieveXML(); it also takes the name of an enabled XML collection.

**Procedure:**

To decompose an XML collection: dxxInsertXML():

Embed a stored procedure call in your application using the following stored procedure declaration:

```
dxxInsertXML(char(
             ) collectionName,   /* input */
             CLOB(1M)       xmlobj,          /* input */
             long           returnCode,      /* output */
             varchar(1024)  returnMsg)       /* output */
```

**Example:** The following is an example of a call to dxxInsertXML():

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
  EXEC SQL BEGIN DECLARE SECTION;
   char     collection[64];       /* name of XML collection */
   SQL TYPE is CLOB_FILE xmlFile; /* input XML file */
   SQL TYPE is CLOB(1M)   xmlDoc; /* input XML doc */
   long     returnCode;           /* error code */
   char     returnMsg[1024];      /* error message text */
   short    collection_ind;
   short    xmlDoc_ind;
   short    returnCode_ind;
   short    returnMsg_ind;
EXEC SQL END DECLARE SECTION;

   /* initialize host variable and indicators */
    strcpy(collection,"sales_ord")strcpy
   (xmlobj.name,"dxx_install/samples/extenders/db2xml
   /xml/getstart_xcollection.xml");
   xmlobj.name_length=strlen("dxx_install/samples/extenders/db2xml
   /xml/getstart_xcollection.xml");

   xmlobj.file_option=SQL_FILE_READ;
   SQL EXEC VALUES (:xmlFile) INTO (:xmlDoc);
   returnCode = 0;
   returnMsg[0] = '\0';
   collection_ind = 0;
   xmlobj_ind = 0;
   returnCode_ind = -1;
   returnMsg_ind = -1;

  /* Call the stored procedure */
 EXEC SQL CALL DB2XML.dxxInsertXML
        (:collection:collection_ind;
        :xmlDoc:xmlDoc_ind,
        :returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

## Enabling an XML collection for decomposition

In most cases, you need to enable an XML collection before using the stored procedures. Cases where you must enable the collections are:

- When decomposing XML documents into new tables, an XML collection must be enabled because all tables in the XML collection are created by the XML Extender when the collection is enabled.

- When keeping the sequence of elements and attributes that have multiple occurrence is important. The XML Extender preserves only the sequence order of elements or attributes of multiple occurrence for tables that are created when a collection is enabled. When XML documents are decomposed into existing relational tables, the sequence order is not guaranteed to be preserved.

See the section about the dxxadm administration command for information about the enable_collection option.

If you want to pass the DAD file when the tables already exist in your database, you do not need to enable an XML collection.

## Decomposition table size limits

Decomposition uses RDB_node mapping to specify how an XML document is decomposed into DB2 database tables by extracting the element and attribute values and storing them in table rows. The values from each XML document are stored in one or more DB2 database tables. Each table can have a maximum of 10240 rows decomposed from each document.

For example, if an XML document is decomposed into five tables, each of the five tables can have up to 10240 rows for that particular document.

Using multiple-occurring elements (elements with location paths that can occur more than once in the XML structure) affects the number of rows. For example, a document that contains an element <Part> that occurs 20 times might be decomposed as 20 rows in a table. When using multiple occurring elements, consider that a maximum of 10240 rows can be decomposed into one table from a single document.

**Related concepts:**


- "XML Extender decomposition stored procedures - Overview" on page 197


**Related tasks:**


- "Calling XML Extender stored procedures" on page 179
- "Decomposing an XML collection by using RDB_node mapping" on page 65


**Related reference:**


- "dxxInsertXML() stored procedure" on page 199
- "dxxShredXML() stored procedure" on page 197


# Updating and deleting data in XML collections

You can update, delete, search, and retrieve XML collections. However, the purpose of using an XML collection is to store or retrieve untagged data in database tables. The data in existing database tables has nothing to do with any incoming XML documents; update, delete, and search operations consist of normal SQL access to these tables.

The XML Extender provides the ability to perform operations on the data from an XML collection view. You can run SQL UPDATE and DELETE statements to modify the data that is used for composing XML documents, and therefore, update the XML collection. Performing SQL operations on the collection tables affects the generated documents.

- To update a document, do not delete a row containing the primary key of the table, which is the foreign key row of the other collection tables. When the primary key and foreign key row is deleted, the document is deleted.
- To replace or delete elements and attribute values, you can delete and insert rows in lower-level tables without deleting the document.
- To delete a document, delete the row that composes the top element_node specified in the DAD.

## Updating data in an XML collection

The XML Extender allows you to update untagged data that is stored in XML collection tables. By updating XML collection table values, you are updating the text of an XML element, or the value of an XML attribute. Updates can also delete an instance of data from multiple-occurring elements or attributes.

From an SQL point of view, changing the value of the element or attribute is an update operation, and deleting an instance of an element or attribute is a delete operation. From an XML point of view, if the element text or attribute value of the root element_node exists, the XML document still exists and is, therefore, an update operation. SQL operations on collection tables affect documents that will be generated from the tables.

**Requirements:** When you update data in an XML collection, observe the following rules:

- Specify the primary-foreign key relationship among the collection tables when the existing tables have this relationship. If they do not, ensure that there are columns that can be joined.
- Include the join condition that is specified in the DAD file:
  - For SQL mapping, include the join condition in the <SQL_stmt> element.
  - For RDB_node mapping, include the join condition in the top <condition> element of the root element node.

### Updating element and attribute values

In an XML collection, element text and attribute values are all mapped to columns in database tables. Regardless of whether the column data previously exists or is decomposed from incoming XML documents, you replace the data using the normal SQL update technique.

To update an element or attribute value, specify a WHERE clause in the SQL UPDATE statement that contains the join condition that is specified in the DAD file.

**Example:**

```
UPDATE SHIP_TAB
    set MODE = 'BOAT'
  WHERE MODE='AIR' AND PART_KEY in
    (SELECT PART_KEY from PART_TAB WHERE ORDER_KEY=68)
```

The <ShipMode> element value is updated from AIR to BOAT in the SHIP_TAB table, where the key is 68.

### Deleting element and attribute instances

To update composed XML documents by eliminating multiple-occurring elements or attributes, delete a row containing the field value that corresponds to the element or attribute value, using the WHERE clause. If you do not delete the row

that contains the values for the top element_node, deleting element values is considered an update of the XML document.

For example, in the following DELETE statement, you are deleting a <shipment> element by specifying a unique value of one of its sub-elements.

```
DELETE from SHIP_TAB
   WHERE DATE='1999-04-12'
```

Specifying a DATE value deletes the row that matches this value. The composed document originally contained two <shipment> elements, but now contains one.

## Deleting an XML document from an XML collection

You can delete an XML document that is composed from a collection. This means that if you have an XML collection that composes multiple XML documents, you can delete one of these composed documents. Performing SQL operations on the collection tables affects the generated documents.

**Procedure:**

To delete the document, delete a row in the table that composes the top element_node that is specified in the DAD file. This table contains the primary key for the top-level collection table and the foreign key for the lower-level tables. Deleting the document with this method works only if the primary-key and foreign-key constraints are fully specified in the SQL and if the relationship of the tables shown in the DAD match those constraints exactly.

**Example:**

The following DELETE statement specifies the value of the primary key column.

```
DELETE from order_tab
   WHERE order_key=1
```

ORDER_KEY is the primary key in the table ORDER_TAB, which is the top-level table as specified in the DAD. Deleting this row deletes one XML document that is generated during composition. Therefore, from the XML point of view, one XML document is deleted from the XML collection.

# Searching XML collections

This section describes searching an XML collection in terms of generating XML documents using search criteria, and searching for decomposed XML data.

## Composing XML documents using search criteria

This task is the same as composition using a condition.

**Procedure:**

You can specify the search criteria using the following search criteria:

- Specify the condition in the text_node and attribute_node of the DAD file
- Specify the *override* parameter when using the dxxGenXML() and dxxRetrieveXML() stored procedures.

For example, if you enabled an XML collection, sales_ord, using the DAD file, order.dad, but you now want to override the price using form data derived from the Web, you can override the value of the <SQL_stmt> DAD element, as follows:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
       ...
             EXEC SQL END DECLARE SECTION;

         float    price_value;

         /* create table */
         EXEC SQL CREATE TABLE xml_order_tab (xmlorder XMLVarchar);

         /* initialize host variable and indicators */
         strcpy(collection,"sales_ord");
         strcpy(result_tab,"xml_order_tab");
         overrideType = SQL_OVERRIDE;
         max_row = 20;
         num_row = 0;
         returnCode = 0;
         msg_txt[0] = '\0';
         override_ind = 0;
         overrideType_ind = 0;
         rtab_ind = 0;
         maxrow_ind = 0;
         numrow_ind = -1;
         returnCode_ind = -1;
         returnMsg_ind = -1;

         /* get the price_value from some place, such as form data */
         price_value = 1000.00         /* for example*/

         /* specify the overwrite */
         sprintf(overwrite,
                "SELECT o.order_key, customer, p.part_key, quantity, price,
                       tax, ship_id, date, mode
                FROM order_tab o, part_tab p,
                    table
(select substr(char(timestamp(generate_unique())),16)
                    as ship_id, date, mode from ship_tab)  s
                WHERE p.price > %d and s.date >'1996-06-01' AND
                    p.order_key = o.order_key and s.part_key = p.part_key",
                    price_value);

        /* Call the stored procedure */
        EXEC SQL CALL db2xml.dxxRetrieveXML(:collection:collection_ind,
                :result_tab:rtab_ind,
                :overrideType:overrideType_ind,:overwrite:overwrite_ind,
                :max_row:maxrow_ind,:num_row:numrow_ind,
                :returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

The condition of price > 2500.00 in order.dad is overridden by price > ?, where ? is based on the input variable *price_value*.

## Searching for decomposed XML data

You can use normal SQL query operations to search collection tables. You can join collection tables, or use subqueries, and then do a structural-text search on text columns. Apply the results of the structural search to retrieve or generate the specified XML document.

## Mapping schemes for XML collections

If you are using an XML collection, you must select a *mapping scheme*, which specifies how XML data is represented in a relational database. Because XML collections must match the hierarchical structure of XML documents with a relational structure for relational databases, you should understand how the two structures compare. Figure 10 shows how the hierarchical structure can be mapped to relational table columns.

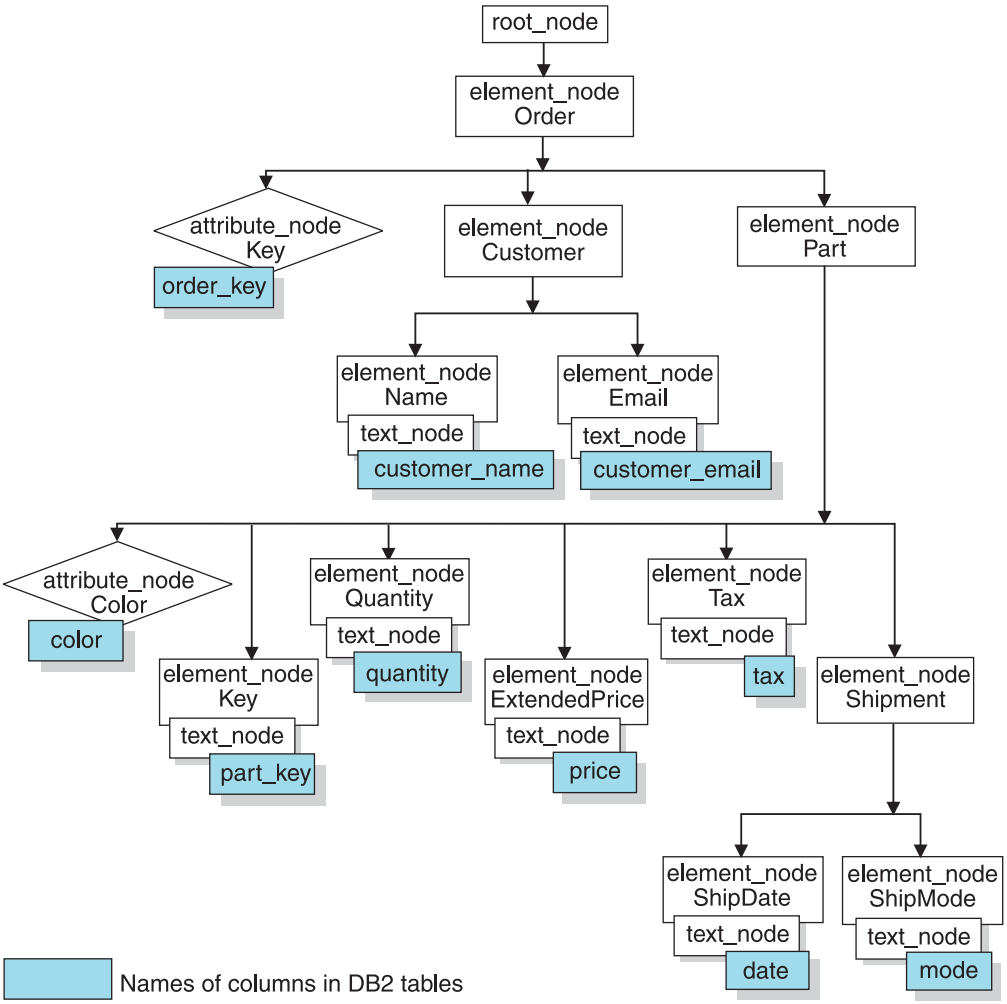



*Figure 10. XML document structure mapped to relational table columns*

The XML Extender uses a mapping scheme when composing or decomposing XML documents that are located in multiple relational tables. Before you create the DAD file, you must think about how your XML data is mapped to the XML collection.

**Types of mapping schemes:**

Use <Xcollection> to specify the mapping scheme in the DAD file. The XML Extender provides two types of mapping schemes: *SQL mapping* and *Relational Database (RDB_node) mapping*.

**SQL mapping**

This method allows direct mapping from relational data to XML

documents through a single SQL statement. SQL mapping is used for composition only. The content of the <SQL_stmt> element must be a valid SQL statement. The <SQL_stmt> element specifies columns in the SELECT clause that are mapped to XML elements or attributes later in the DAD. When defined for composing XML documents, the column names in the SELECT clause of the SQL statement are used to associate the value of an *attribute_node* or a content of *text_node* with columns that have the same *name_attribute*. The FROM clause defines the tables containing the data; the WHERE clause specifies the *join* and search *condition*.

SQL mapping gives DB2® users the power to map the data using SQL. When using SQL mapping, you must be able to join all tables in one SELECT statement to form a query. If one SQL statement is not sufficient, consider using RDB_node mapping. To tie all tables together, the *primary key* and *foreign key* relationship is recommended among these tables.

**RDB_node mapping**
Defines the location of the content of an XML element or the value of an XML attribute so that XML Extender can determine where to store or retrieve the XML data.

This method uses XML Extender-provided *RDB_node*, which contains one or more node definitions for tables, optional columns, and optional conditions. The <table> and <column> elements in the DAD define how the XML data is to be stored in the database. The condition specifies the criteria for selecting XML data or the way to join the XML collection tables.

To define a mapping scheme, you must create a DAD file with an <Xcollection> element. Figure 11 on page 101 shows a fragment of a sample DAD file with SQL mapping for an XML collection, which composes a set of XML documents from data in three relational tables.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "dxx_install/samples/extenders/db2xml/dtd/dad.dtd">
<DAD>
  <dtdid>dxx_install/samples/extenders/db2xml/dad/getstart.dtd</dtdid>
  <validation>YES</validation>
  <Xcollection>
    <SQL_stmt>
       SELECT o.order_key, customer, p.part_key, quantity, price, tax, date,
              ship_id, mode, comment
         FROM order_tab o, part_tab p,
               table(select substr(char(timestamp
                (generate_unique())),16)
                as ship_id, date, mode, from ship_tab)
         WHERE p.price > 2500.00 and s.date > "1996-06-01" AND
              p.order_key = o.order_key and s.part_key = p.part_key
    </SQL_stmt>
    <prolog>?xml version="1.0"?</prolog>
    <doctype>!DOCTYPE DAD SYSTEM
    "dxx_install/samples/extenders/db2xml/dtd/getstart.dtd
     "</doctype>
    <root_node>
    <element_node name="Order">
    <attribute_node name="Key">
   <column name="order_key"/>
    </attribute_node>
    <element_node name="Customer">
     <text_node>
      <column name="customer"/>
     </text_node>
    <element_node>

  ...

    </element_node><!-end Part->
    </element_node><!-end Order->
    </root_node>
  </Xcollection>
</DAD>
```

*Figure 11. SQL mapping scheme*

The XML Extender provides several stored procedures that manage data in an XML collection. These stored procedures support both types of mapping.

**Related concepts:**

- "Requirements for RDB_Node mapping" on page 103
- "Requirements for using SQL mapping"
- "DAD files for XML collections" on page 163

**Related tasks:**

- "Composing XML collections by using RDB_node mapping" on page 61
- "Composing XML documents by using SQL mapping" on page 58
- "Decomposing an XML collection by using RDB_node mapping" on page 65

## Requirements for using SQL mapping

### Requirements when using SQL mapping

In this mapping scheme, you must specify the <SQL_stmt> element inside the DAD <Xcollection> element. The <SQL_stmt> must contain a single SQL statement that can join multiple relational tables with the query *predicate*. In addition, the following clauses are required:

- **SELECT clause**
  - Ensure that the name of the column is unique. If two tables have the same column name, use the AS keyword to create an alias name for one of them.
  - Group columns of the same table together and order the tables according to the tree level as they map to the hierarchical structure of your XML document. The first column in each column grouping is an object ID. In the SELECT clause, the columns of the higher-level tables must precede the columns of lower-level tables. The following example demonstrates the hierarchical relationship among tables:

    ```
    SELECT o.order_key, customer, p.part_key, quantity, price, tax,
           ship_id, date, mode
    ```

    In this example, the `order_key` and `customer` columns from the ORDER_TAB table have the highest relational level because they are higher on the hierarchical tree of the XML document. The `ship_id`, `date`, and `mode` columns from the SHIP_TAB table are at the lowest relational level.
  - Use a single-column candidate key to begin each level. If such a key is not available in a table, the query should generate one for that table using a table expression and the generate_unique() function. In the above example, the `o.order_key` is the primary key for ORDER_TAB, and the `part_key` is the primary key of PART_TAB. They appear at the beginning of their own group of columns that are to be selected. The `ship_id` is generated as a primary key because the SHIP_TAB table does not have a primary key. `ship_id` is listed as the first column for the SHIP_TAB table group. Use the FROM clause to generate the primary key column, as shown in the following example.
- **FROM clause**
  - Use a table expression and the generate_unique() function to generate a single key for tables that do not have a primary single key. For example:

    ```
    FROM order_tab as o, part_tab as p,
            table(select substr
             (char(timestamp(generate_unique())),16)
              as
              ship_id, date, mode, part key from ship_tab) as s
    ```

    In this example, a single column candidate key is generated with the generate_unique() function and given an alias named `ship_id`.
  - Use an alias name when it is necessary to make a column distinct. For example, you could use `o` for columns in the ORDER_TAB table, `p` for columns in the PART_TAB table, and `s` for columns in the SHIP_TAB table.
- **WHERE clause**
  - Specify a primary and foreign key relationship as the join condition that ties tables in the collection together. For example:

    ```
    WHERE p.price > 2500.00 AND s.date > "1996-06-01" AND
          p.order_key = o.order_key AND s.part_key = p.part_key
    ```
  - Specify any other search condition in the predicate. Any valid predicate can be used.
- **ORDER BY clause**
  - Define the ORDER BY clause at the end of the SQL_stmt. Ensure that there is nothing after the column names such as ASC or DESC.

- Ensure that the column names match the column names in the SELECT clause.
- List all object ID's in the same relative order as they appear in the SELECT clause.
- An identifier can be generated using a table expression and the generate_unique() function or a user defined function.
- Maintain the top-down order of the hierarchy of the entities. The first column specified in the ORDER BY clause must be the first column listed for each entity. Keeping the order ensures that the XML documents to be generated do not contain incorrect duplicates.
- Do not qualify the columns in the ORDER BY clause with a schema or table name.

The <SQL_stmt> element is powerful because you can specify any predicate in your WHERE clause, as long as the expression in the predicate uses the columns in the tables.

**Related reference:**

• Appendix A, "Samples," on page 275

## Requirements for RDB_Node mapping

When using RDB_Node as your mapping method, do not use the <SQL_stmt>element in the <Xcollection> element of the DAD file. Instead, use the RDB_node element as a child of the top element_node and of each attribute_node and text_node.

- **RDB_node for the top element_node**

  The top element_node in the DAD file represents the root element of the XML document. Specify an RDB_node for the top element_node as follows:

  - Specify all tables that are associated with the XML collection. For example, the following mapping specifies three tables in the <RDB_node> of the<Order> element node, which is the top element node:

```
<element_node name="Order">
    <RDB_node>
        <table name="order_tab"/>
        <table name="part_tab"/>
        <table name="ship_tab"/>
        <condition>
           order_tab.order_key = part_tab.order_key AND
           part_tab.part_key = ship_tab.part_key
        </condition>
    </RDB_node>
```

    The condition element can be empty or missing if there is only one table in the collection.
  - There are no ordering restrictions on predicates of the root node condition.
  - Condition elements can reference a column name an unlimited number of times.
  - If you are enabling a collection, you must specify a primary key for each table. The primary key can consist of a single column or multiple columns, called a composite key. Specify the primary key by adding an *attribute key* to the table element of the RDB_node. When you supply a composite key, the *key* attribute will be specified by the names of key columns separated by a space. For example:

```
<table name="part_tab" key="part_key price"/>
```

The information specified for decomposition is ignored if the same DAD is used for composition.

– Use the orderBy attribute to recompose XML documents containing elements or attributes with multiple occurrence back to their original structure. This attribute allows you to specify the name of a column that will be the key used to preserve the order of the document. The orderBy attribute is part of the table element in the DAD file, and it is an optional attribute. When you decompose XML documents in an XML collection, you can lose the order of multiple-occurring elements and attribute values, unless you specify the order in the DAD file. To preserve this order, you should use the RDB_node mapping scheme and specify the orderBy attribute for the table containing the root element in its RDB_node.

Spell out the table name and the column name in the <table>tag.

- **RDB_node for each attribute_node and text_node**

  The XML Extender needs to know from where in the database to retrieve the data. XML Extender also needs to know where in the database to put the content from an XML document. You must specify an RDB_node for each attribute node and text node. You must also specify the table and column names; the condition value is optional.

  1. Specify the name of the table containing the column data. The table name must be included in the RDB_node of the top element_node. In this example, for text_node of element <Price>, the table is specified as PART_TAB.

```
<element_node name="Price">
    <text_node>
        <RDB_node>
            <table name="part_tab"/>
            <column name="price"/>
            <condition>
                 price > 2500.00
            </condition>
        </RDB_node>
    </text_node>
</element_node>
```

  2. Specify the name of the column that contains the data for the element text. In the previous example, the column is specified as PRICE.
  3. Specify a query condition if you want XML documents to be generated using that condition. Only the data meeting the condition is in the generated XML documents. The condition must be a valid WHERE clause. In the example above, the condition is specified as `price > 2500.00`, so only rows where the price is over 2500 will be included in the XML documents.
  4. If you are decomposing a document, or enabling the XML collection specified by the DAD file, you must specify the column type for each attribute node and text node. By specifying the column type for each attribute node and text node, you ensure that he correct data type for each column when new tables are created during the enabling of an XML collection. Column types are specified by adding the attribute type to the column element. For example:

```
<column name="order_key" type="integer"/>
```

     The column type specified when decomposing a document is ignored for composition.

- Maintain the top-down order of the hierarchy of the entities. Ensure that the element nodes are nested properly so that XML Extender understands the

relationship between the elements when composing or decomposing documents. For example, the following DAD file does not nest Shipment inside of Part:

```
<element_node name="Part">
    ...
    <element_node name="ExtendedPrice">
        ...
    </element_node>
        ...
</element_node> <!-- end of element Part -->

<element_node name="Shipment" multi_occurrence="YES">
        <element_node name="ShipDate">
            ...
        </element_node>
        <element_node name="ShipMode">
            ...
        </element_node>

</element_node> <!-- end of element Shipment-->
```

This DAD file produces an XML documents in which the Part and Shipment elements are siblings.

```
<Part color="black ">
  <key>68</key>
  <Quantity>36</Quantity>
  <ExtendedPrice>34850.16</ExtendedPrice>
  <Tax>6.000000e-2</Tax>
</Part>

<Shipment>
    <ShipDate>1998-08-19</ShipDate>
    <ShipMode>BOAT  </ShipMode>
</Shipment>
```

The following code shows the shipment element nested inside the Part element in the DAD file.

```
<element_node name="Part">
    ...
    <element_node name="ExtendedPrice">
        ...
    </element_node>
    ...
    <element_node name="Shipment" multi_occurrence="YES">
        <element_node name="ShipDate">
            ...
        </element_node>
        <element_node name="ShipMode">
            ...
        </element_node>

    </element_node> <!-- end of element Shipment-->
</element_node> <!-- end of element Part -->
```

Nesting the shipment element inside the part element produces an XML file with Shipment as a child element of the Part element:

```
<Part color="black ">
  <key>68</key>
  <Quantity>36</Quantity>
  <ExtendedPrice>34850.16</ExtendedPrice>
  <Tax>6.000000e-2</Tax>
  <Shipment>
    <ShipDate>1998-08-19</ShipDate>
    <ShipMode>BOAT  </ShipMode>
  </Shipment>
</Part>
```

With the RDB_node mapping approach, you don't need to supply SQL statements. However, putting complex query conditions in the RDB_node element can be more difficult.

For a subtree of the DAD with element_nodes and attribute_nodes that map to the same table, the following is true:

- Attribute nodes do not have to be the first children of the lowest common ancestor of the element nodes that map to the same table.
- Attribute nodes can appear anywhere in the subtree, as long as they are not involved in a join condition.

**Restrictions:** The limit for the number of tables allowed in a RDB_node mapping DAD is 30. The number of columns allowed per table is 500. The number of times each table or column can be specified in the join predicates of the condition statement is unlimited

## Stylesheets for an XML collection

When composing documents, XML Extender also supports processing instructions for stylesheets, using the <stylesheet> element. The processing instructions must be inside the <Xcollection> root element, located with the <doctype> and <prolog> defined for the XML document structure. For example:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dtd\dad.dtd">
<DAD>
<SQL_stmt>
      ...
</SQL_stmt>
<Xcollection>
...
<prolog>...</prolog>
<doctype>...</doctype>
<stylesheet>?xml-stylesheet type="text/css" href="order.css"?</stylesheet>
<root_node>...</root_node>
...

</Xcollection>
...
</DAD>
```

## Location paths

A *location path* defines the location of an XML element or attribute within the structure of the XML document. The XML Extender uses the location path for the following purposes:

- To locate the elements and attributes to be extracted when using extraction UDFs such as dxxRetrieveXML.
- To specify the mapping between an XML element or attribute and a DB2® column when defining the indexing scheme in the DAD for XML columns
- For structural text search, using the Net Search Extender

For example, a location path of `/Order/Part/Shipment/ShipDate` points to the ShipDate element, that is a child of the Shipment, Part, and Order elements, as shown in the following example:

```
<Order>
  <Part>
    <Shipment>
      <ShipDate>
+...
```

Figure 12 shows an example of a location path and its relationship to the structure of the XML document.

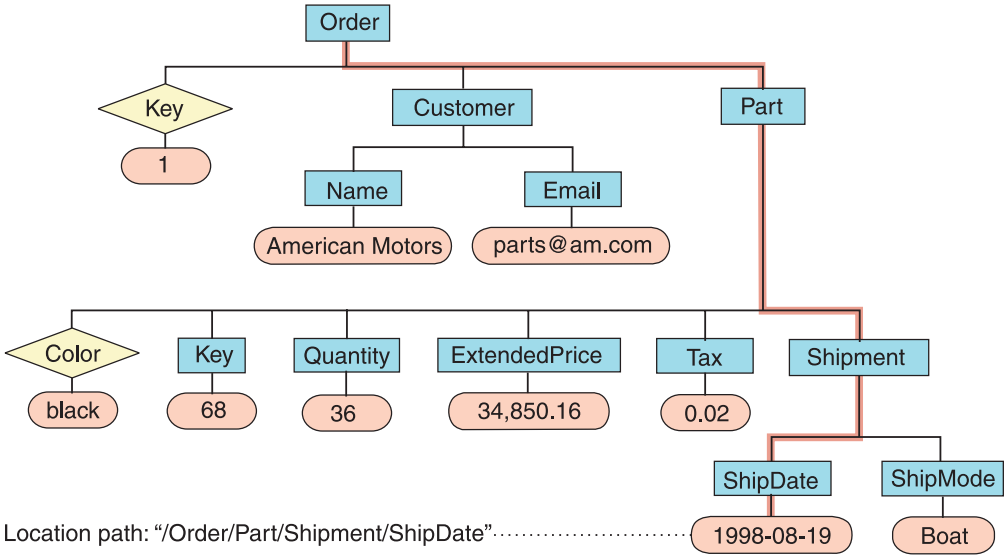



*Figure 12. Storing documents as structured XML documents in a DB2 table column*

The location path is used in the following situations:

**XML columns**

- Used to identify the elements and attributes to be extracted or updated when using the XML Extender user-defined functions.
- Also used to map the content of an XML element or attribute to a side table.

**XML collections**

Used to override values in the DAD file from a stored procedure.

To specify the location path, XML Extender uses a subset of the *XML Path Language (XPath)*, the language for addressing parts of an XML document.

For more information about XPath, see the following Web page:

http://www.w3.org/TR/xpath

**Related reference:**

- "Location path syntax"

## Location path syntax

XML Extender uses the location path to navigate the XML document structure. The following list describes the location path syntax that is supported by the XML Extender. A single slash (/) path indicates that the context is the whole document.

1. / Represents the XML root element. This is the element that contains all the other elements in the document.

2. */tag1*
   Represents the element *tag1* under the root element.

3. */tag1/tag2/..../tagn*
   Represents an element with the name *tagn* as the child of the descending chain from root, *tag1*, *tag2*, through *tagn*-1.

4. *//tagn*
   Represents any element with the name *tagn*, where double slashes (//) denote zero or more arbitrary tags.

5. */tag1//tagn*
   Represents any element with the name *tagn*, a descendent of an element with the name *tag1* under root, where double slashes (//) denote zero or more arbitrary tags.

6. */tag1/tag2/@attr1*
   Represents the attribute *attr1* of an element with the name *tag2*, which is a child of element *tag1* under root.

7. */tag1/tag2[@attr1="5"]*
   Represents an element with the name *tag2* whose attribute *attr1* has the value 5. The *tag2* element is a child of the *tag1* element under root.

8. */tag1/tag2[@attr1="5"]/.../tagn*
   Represents an element with the name *tagn*, which is a child of the descending chain from root, *tag1*, *tag2*, through *tagn*-1, where the attribute *attr1* of *tag2* has the value 5.

**Simple location path**

*Simple location path* is a type of location path used in the XML column DAD file. A simple location path is represented as a sequence of element-type names that are connected by a single slash (/). The values of each attribute are enclosed within square brackets following the element type. Table 13 summarizes the syntax for simple location path.

*Table 13. Simple location path syntax*

| Subject | Location path | Description |
|---|---|---|
| XML element | */tag1/tag2/..../tagn-1/tagn* | An element content identified by the element named *tagn* and its parents |
| XML attribute | */tag_1/tag_2/..../tag_n-1/tag_n/@attr1* | An attribute named *attr1* of the element identified by *tagn* and its parents |

**Location path usage**

The syntax of the location path is dependent on the context in which you are accessing the location of an element or attribute. Because the XML Extender uses one-to-one mapping between an element or attribute, and a DB2 column, it restricts the syntax rules for the DAD file and functions. Table 14 on page 109 describes in which contexts the syntax options are used.

*Table 14. The XML Extender's restrictions using location path*

| Use of the location path | Location path supported |
|---|---|
| Value of path attribute in the XML column DAD mapping for side tables | */tag1/tag2/..../tagn* and */tag1/tag2/@attr1* (simple location path described in Table 13 on page 108) |
| Extracting UDFs | All location paths[1] |
| Update UDF | All location paths[1] |

[1] The extracting and updating UDFs support location paths that have predicates with attributes, but not elements.

**Related concepts:**

- "Location paths" on page 106

## Enabling XML collections

Enabling an XML collection parses the DAD file to identify the tables and columns related to the XML document, and records control information in the XML_USAGE table. Enabling an XML collection is optional for:

- Decomposing an XML document and storing the data in new DB2 database tables
- Composing an XML document from existing data in multiple DB2 database tables

If the same DAD file is used for composing and decomposing, you can enable the collection for both composition and decomposition.

You can enable an XML collection with the XML Extender Administration wizard, with the **dxxadm** command with the enable_collection option, or with the XML Extender stored procedure dxxEnableCollection().

**Using the Administration wizard:**

To enable an XML collection using the wizard:

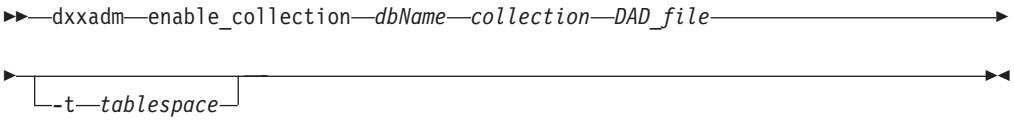
1. Set up and start the Administration wizard.
2. Click **Work with XML Collections** from the Launchpad window. The Select a Task window opens.
3. Click **Enable a Collection** and then **Next**. The Enable a Collection window opens.
4. Select the name of the collection that you want to enable in the **Collection name** field.
5. Specify the DAD file name in the **DAD file name** field.
6. Optional: Type the name of a previously created table space in the **Table space** field.

   The table space will contain new DB2 database tables generated for decomposition.
7. Click **Finish** to enable the collection and return to the Launchpad window.
   - If the collection is successfully enabled, an `Enabled collection is successful` message is displayed.
   - If the collection is not successfully enabled, an error message is displayed. Repeat the steps above until the collection is successfully enabled.

**Enabling collections using the dxxadm command:**

To enable an XML collection, enter the **dxxadm** command from a DB2 database command line:

**Syntax:**




**Parameters:**

*dbName*
The name of the database.

*collection*
The name of the XML collection. This value is used as a parameter for the XML collection stored procedures.

*DAD_file*
The name of the file that contains the document access definition (DAD).

*tablespace*
An existing table space that contains new DB2 database tables that were generated for decomposition. If not specified, the default table space is used.

**Example:** The following example enables a collection called sales_ord in the database SALES_DB using the command line. The DAD file uses SQL mapping.

```
dxxadm enable_collection SALES_DB sales_ord getstart_collection.dad
```

After you enable the XML collection, you can compose or decompose XML documents using the XML Extender stored procedures.

**Related concepts:**

- "XML collections as a storage and access method" on page 87

**Related tasks:**

- "Disabling XML collections"
- "Managing data in XML collections" on page 88

## Disabling XML collections

Disabling an XML collection removes the record in the XML_USAGE table that identifies tables and columns as part of a collection. It does not drop any data tables. You disable a collection when you want to update the DAD and need to re-enable a collection, or when you want to drop a collection.

You can disable an XML collection with the XML Extender Administration wizard, with the **dxxadm** command with the disable_collection option, or with the XML Extender stored procedure dxxDisableCollection().

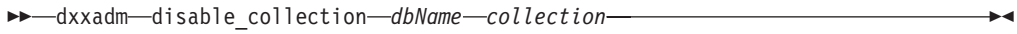
**Procedure:**

To disable an XML collection using the Administration wizard:

1. Start the Administration wizard.
2. Click **Work with XML Collections** from the Launchpad window to view the XML Extender collection related tasks. The Select a Task window opens.
3. Click **Disable an XML Collection** and then **Next** to disable an XML collection. The Disable a Collection window opens.
4. Type the name of the collection that you want to disable in the **Collection name** field.
5. Click **Finish** to disable the collection and return to the Launchpad window.
   - If the collection is successfully disabled, `Disabled collection is successful message` is displayed.
   - If the collection is not successfully disabled, an error box is displayed. Repeat the steps above until the collection is successfully disabled.

To disable an XML collection from the command line, enter the **dxxadm** command.

**Syntax:**




**Parameters:**

*dbName*
The name of the database.

*collection*
The name of the XML collection. This value is used as a parameter for the XML collection stored procedures.

**Example:**

```
dxxadm disable_collection SALES_DB sales_ord
```

**Related concepts:**


- "XML collections as a storage and access method" on page 87
- "XML Extender administration stored procedures - Overview" on page 181


**Related tasks:**


- "Managing data in XML collections" on page 88

# Chapter 5. XML schemas

The XML schema can be used in place of a DTD to define the specifications for the content of XML documents. The XML schema uses XML format or XML syntax to define the elements and attribute names of an XML document, and defines the type of content that the elements and attributes are allowed to contain.

## Advantages of using XML schemas instead of DTDs

DTDs are easier to code and validate than an XML schema. However, the advantages to using an XML schema are shown in the following list:

- XML schemas are valid XML documents that can be processed by tools such as the XSD Editor in DB2 Developer Workbench, XML Spy, or XML Authority.
- XML schemas are more powerful than DTDs. Everything that can be defined by the DTD can also be defined by schemas, but not vice versa.
- XML schemas support a set of data types, similar to the ones used in most common programming languages, and provide the ability to create additional types. You can constrain the document content to the appropriate type. For example, you can replicate the properties of fields found in DB2®.
- XML schemas support regular expressions to set constraints on character data, which is not possible if you use a DTD.
- XML schemas provide better support for XML namespaces, which enable you to validate documents that use multiple namespaces, and to reuse constructs from schemas already defined in different namespaces.
- XML schemas provide better support for modularity and reuse with include and import elements.
- XML schemas support inheritance for element, attribute, and data type definitions.

**Related tasks:**

- "Data types, elements and attributes in schemas" on page 114

**Related reference:**

- "Examples of an XML schema" on page 115

## XML schema complexType element

The XML schema element complexType is used to define an element type that can consist of sub-elements. For example, the following tags show the projection of an address in an XML document:

```
<billTo country="US">
    <name>Dan Jones</name>
    <street>My Street</street>
    <city>My Town</city>
    <state>CA</state>
    <zip>99999</zip>
</billTo>
```

The structure of this element can be defined in the XML schema as follows:

© Copyright IBM Corp. 1999 - 2006

```
1 <xsd:element name="billTo" type="USAddress"/>
2 < xsd:complexType name="USAddress">
 3  <xsd:sequence>
 4   < xsd:element name="name" type="xsd:string"/>
 5   < xsd:element name="street" type="xsd:string"/>
 6   < xsd:element name="city" type="xsd:string"/>
 7   < xsd:element name="state" type="xsd:string"/>
 8   < xsd:element name="zip" type="xsd:decimal"/>
 9  </xsd:sequence>
 10  < xsd:attribute name="country"
                          type="xsd:NMTOKEN" use="fixed"
                          value="US"/>
 12</xsd:complexType>
```

In the above example, it is assumed that the xsd prefix has been bound to the XML schema namespace. Lines 2 through 12 define the complexType USAddress as a sequence of five elements and one attribute. The order of the elements is determined by the order in which they appear in the sequence tag.

The inner elements are from data type xsd:string or xsd:decimal. Both are predefined simple data types.

Alternatively, you can use the <all> tag or the <choice> tag instead of the <sequence> tag. With the all tag, all sub-elements must appear, but do not need to appear in any particular order. With the choice tag, exactly one of the sub-elements must appear in the XML document

You can also use a user-defined data type to define other elements.

# Data types, elements and attributes in schemas

## Simple data types in XML schemas

XML schemas provide a set of simple built-in data types. You can derive other data types from them by applying constraints.

**Example 1:** The range of base type xsd:positiveInteger is limited to 0 to 100.

```
< xsd:element name="quantity">
   < xsd:simpleType>
    < xsd:restriction base="xsd:positiveInteger">
     < xsd:maxExclusive value="100"/>
    </xsd:restriction>
   </xsd:simpleType>
  </xsd:element>
```

**Example 2:** The base type xsd:string is limited by a regular expression.

```
<xsd:simpleType name="SKU">
   < xsd:restriction base="xsd:string">
    < xsd:pattern value="\d{3}-[A-Z]{2}"/>
   </xsd:restriction>
  </xsd:simpleType>
```

**Example 3:** Shows an enumerated type based on the string built-in type.

```
<xsd:simpleType name="SchoolClass">
   < xsd:restriction base="xsd:string">
    < xsd:enumeration value="WI"/>
    < xsd:enumeration value="MI"/>
    < xsd:enumeration value="II"/>
```

```
   < xsd:enumeration value="DI"/>
   < xsd:enumeration value="AI"/>
  </xsd:restriction>
 </xsd:simpleType>
```

## Elements in XML schemas

To declare an element in an XML schema you must indicate the name and type as an attribute of the element element. For example:

```
<xsd:element name="street" type="xsd:string"/>
```

Additionally, you can use the attributes minOccurs and maxOccurs to determine the maximum or minimum number of times that the element must appear in the XML document. The default value of minOccurs and maxOccurs is 1.

## Attributes in XML schemas

Attribute declarations appear at the end of an element definition. For example:

```
<xsd:complexType name="PurchaseOrderType">
< xsd:sequence>
  < xsd:element name="billTo" type="USAddress"/>
< xsd:sequence>
  < xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>
```

**Related concepts:**

- "Advantages of using XML schemas instead of DTDs" on page 113

**Related tasks:**

- "Validation functions" on page 156

**Related reference:**

- "Examples of an XML schema"
- "XML schema complexType element" on page 113

# Examples of an XML schema

It is a good strategy to write XML schemas by first designing the data structure of your XML document using a UML tool. After you design the structure, you can map the structure into your schema document. The following example shows an XML schema.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
 3
 4   <xs:element name="personnel">
 5    <xs:complexType>
 6     <xs:sequence>
 7       <xs:element ref="person" minOccurs='1' maxOccurs='unbounded'/>
 8    </xs:sequence>
 9   </xs:complexType>
10  </xs:element>
11
12   <xs:element name="person">
13    <xs:complexType>
14     <xs:sequence>
15       <xs:element ref="name"/>
16       <xs:element ref="email" minOccurs='0' maxOccurs='4'/>
17    </xs:sequence>
18     <xs:attribute name="id"  type="xs:ID" use='required'/>
```

```
19   </xs:complexType>
20  </xs:element>
21
22   <xs:element name="name">
23    <xs:complexType>
24     <xs:sequence>
25      <xs:element ref="family"/>
26      <xs:element ref="given"/>
27    </xs:sequence>
28   </xs:complexType>
29  </xs:element>
30
31   <xs:element name="family" type='xs:string'/>
32   <xs:element name="given" type='xs:string'/>
33   <xs:element name="email" type='xs:string'/>
34 </xs:schema>
```

The first two lines declare that this XML schema is XML 1.0 compatible and Unicode 8 decoded, and specify use of the XML schema standard namespace, which enables access to basic XML schema data types and structures.

Lines 4 to 10 define the personnel as a complexType that consists of a sequence of 1 to n persons. The complexType is then defined in lines 12 to 20. It consists of the complexType element name and the element email. The email element is optional (minOcccurs = '0'), and can appear up to four times (maxOccurs = '4'). The greater the number of occurrences of an element, the longer it will take to validate the schema. In contrast, in a DTD you can choose only 0, 1, or unlimited appearances of an element.

Lines 22 to 29 define the name type that is used for the person type. The name type consists of a sequence of a family and a given element.

Lines 31 to 33 define the single elements family, given, and e-mail, which contain type strings that have been declared.

## XML document instance using the schema

The following example is an XML document that is an instance of the *personalnr.xsd* schema.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <personnel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:noNamespaceSchemaLocation='personsnr.xsd'>
4
5   <person id="Big.Boss" >
6      <name><family>Boss</family><given>Big</given></name>
7     <email>chief@foo.com</email>
8   </person>
9
10   <person id="one.worker">
11   <name><family>Worker</family><given>One</given></name>
12      <email>one@foo.com</email>
13   </person>
14
15    <person id="two.worker">
16   <name><family>Worker</family><given>Two</given></name>
17      <email>two@foo.com</email>
18   </person>
19 </personnel>
```

## XML document instance using a DTD

This example shows how this XML schema would be realized as a DTD.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <!ELEMENT email (#PCDATA)>
3   <!ELEMENT family (#PCDATA)>
4   <!ELEMENT given (#PCDATA)>
5   <!ELEMENT name (family, given)>
6   <!ELEMENT person (name, email*)>
7
8   <!ATTLIST person
9 id ID #REQUIRED>
10 <!ELEMENT personnel (person+)>
```

Using a DTD you can set the maximum occurrence of email to only 1 or unlimited occurrences.

Using this DTD, the XML document instance would be the same as shown in the top example, except line 2 would be changed to:

```
<!DOCTYPE personnel SYSTEM "personsnr.dtd">
```

**Related concepts:**

- "Advantages of using XML schemas instead of DTDs" on page 113

**Related tasks:**

- "Data types, elements and attributes in schemas" on page 114
- "Validation functions" on page 156

**Related reference:**

- "XML schema complexType element" on page 113

# Part 4. Reference

This part provides syntax information for the XML Extender administration command, user-defined data types (UDTs), user-defined functions (UDFs), and stored procedures. Message text is also provided for problem determination activities.

© Copyright IBM Corp. 1999 - 2006

# Chapter 6. The dxxadm administration command

## dxxadm command overview

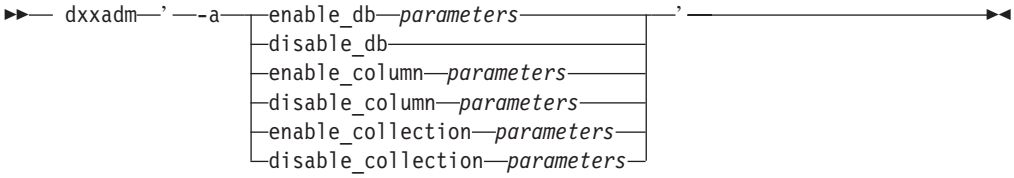
The XML Extender provides an administration command, **dxxadm**, for completing the following administration tasks:

- Enabling or disabling a database for XML Extender
- Enabling or disabling an XML column
- Enabling or disabling an XML collection

**Related concepts:**

- "Administration tools for XML Extender" on page 35
- "XML Extender administration overview" on page 36

## Syntax of the dxxadm administration command




**Parameters:**

*Table 15. dxxadm parameters*

| Parameter | Description |
|---|---|
| *enable_db* | Enables XML Extender features for a database. |
| *disable_db* | Disables XML Extender features for a database. |
| *enable_column* | Enables an XML column so that XML documents can be stored in the column. |
| *disable_column* | Disables the XML-enabled column. |
| *enable_collection* | Enables an XML collection according to the specified DAD. |
| *enable_collection* | Disables an XML-enabled collection. |

## Options for the administration command

The following **dxxadm** command options are available to system programmers:

- enable_column
- enable_collection
- enable_db
- disable_column
- disable_collection
- disable_db

© Copyright IBM Corp. 1999 - 2006

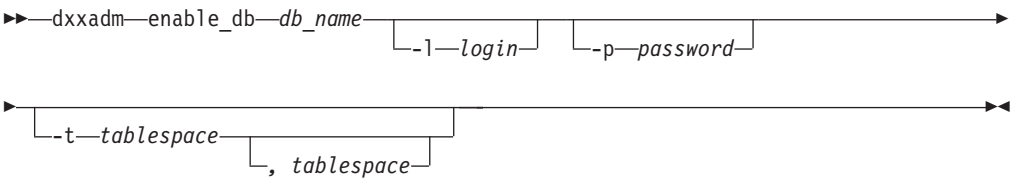
## enable_db option of the dxxadm command

**Purpose:**

Enables XML Extender features for a database. When the database is enabled, the XML Extender creates the following objects:

- The XML Extender user-defined types (UDTs).
- The XML Extender user-defined functions (UDFs).
- The XML Extender DTD repository table, DTD_REF, which stores DTDs and information about each DTD.
- The XML Extender usage table, XML_USAGE, which stores common information for each column that is enabled for XML and for each collection.

**Syntax:**




**Parameters:**

*Table 16. enable_db parameters*

| Parameter | Description |
|---|---|
| *db_name* | The name of the database in which the XML data resides. |
| -l *login* | Optional user ID, used to connect to the database. If not specified, the current user ID is used. |
| -p *password* | Optional password used to connect to the database. If not specified, the current password is used. |
| -t *tablespace* | Optional name of the existing table space to hold db2xml.XML_USAGE and db2xml.DTD_REF tables. A second table space can also be specified. |

If you are using a partitioned DB2 Enterprise Server Edition and you want to specify a table space while enabling the database, you must specify a node group when you create the table space. For example:

```
db2 "create database partition group mygroup on node (0,1)"
db2 "create regular tablespace mytb in database partition group mygroup
   managed by system using ('mytb')"
```

In the example above, you would then specify the `mytb` table space while enabling the database.

If no table space option is provided when the database is enabled, XML Extender will check whether the DXXDTDRF and DXXXMLUS table spaces exist. The `db2xml.dtd_ref` table will be created in the DXXDTDRF table space if that table space exists, and the `db2xml.xml_usage` table will be created in the DXXXMLUS table space. If one of the DXXDTDRF or DXXXMLUS table spaces does not exist, then the respective table (`db2xml.dtd_ref` or `db2xml.xml_usage`) will be created in the most appropriate table space.

If only one DXXDTDRF table space is provided when the database is enabled, both tables will be created in the specified table space. If two table spaces are provided when the database is enabled, the `db2xml.dtd_ref` table will be created in the first table space listed, and the `db2xml.xml_usage` table will be created in the second table space listed.

**Example::**

The following example enables the database SALES_DB.

```
dxxadm enable_db SALES_DB
```

**Related reference:**

- "dxxadm command overview" on page 121

## disable_db option of the dxxadm command

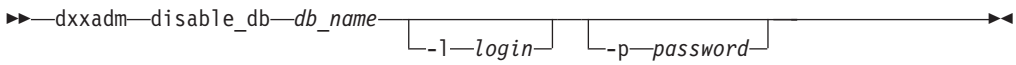
**Purpose:**

Disables XML Extender features for a database; this action is called "disabling a database." When the database is disabled, it can no longer be used by the XML Extender. When the XML Extender disables the database, it drops the following objects:

- The XML Extender user-defined types (UDTs).
- The XML Extender user-defined functions (UDFs).
- The XML Extender DTD repository table, DTD_REF, which stores DTDs and information about each DTD.
- The XML Extender usage table, XML_USAGE, which stores common information for each column that is enabled for XML and for each collection.

**Important:** You must disable all XML columns before attempting to disable a database. The XML Extender cannot disable a database that contains columns or collections that are enabled for XML Extender. You must also drop all tables that have columns defined with XML Extender user-defined types, such as XMLCLOB.

**Syntax:**




**Parameters:**

*Table 17. disable_db parameters*

| Parameter | Description |
|---|---|
| *db_name* | The name of the database in which the XML data resides |
| -l *login* | The user ID used to connect to the database. If not specified, the current user ID is used. |
| -p *password* | The password used to connect to the database. If not specified, the current password is used. |

**Example::**
The following example disables the database SALES_DB.

```
dxxadm disable_db SALES_DB
```

**Related concepts:**

- "XML Extender administration stored procedures - Overview" on page 181
- Chapter 13, "XML Extender administration support tables," on page 251

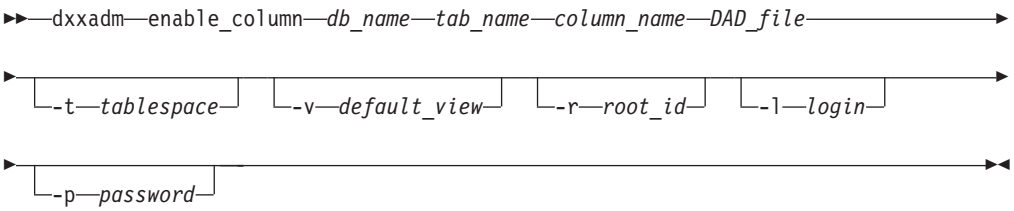
## enable_column option of the dxxadm command

**Purpose:**

Connects to a database and enables an XML column so that it can contain the XML Extender UDTs. When enabling a column, the XML Extender completes the following tasks:

- Determines whether the XML table has a primary key; if not, the XML Extender alters the XML table and adds a column called DXXROOT_ID.
- Creates side tables that are specified in the DAD file with a column containing a unique identifier for each row in the XML table. This column is either the root ID that the user specified or the DXXROOT_ID that was named by the XML Extender.
- Optionally creates a default view for the XML table and its side tables, optionally using a name that you specify.

**Syntax:**




**Parameters:**

*Table 18. enable_column parameters*

| Parameter | Description |
|---|---|
| *db_name* | The name of the database in which the XML data resides. |
| *tab_name* | The name of the table in which the XML column resides. |
| *column_name* | The name of the XML column. |
| *DAD_file* | The name of the DAD file that maps the XML document to the XML column and side tables. |
| -t *tablespace* | The table space that contains the side tables associated with the XML column. If not specified, the default table space is used. |
| -v *default_view* | The name of the default view that joins the XML column and side tables. |
| -r *root_id* | The name of the primary key in the XML column table that is to be used as the root_id for side tables. The root_id is optional. |
| -l *login* | The user ID, used to connect to the database. If not specified, the current user ID is used. |

*Table 18. enable_column parameters (continued)*

| Parameter | Description |
|---|---|
| -p *password* | The password used to connect to the database. If not specified, the current password is used. |

If you are using the partitioned IBM DB2 Enterprise Server Edition and you want to specify a table space while enabling a column, you need to have specified a node group when you created the table space. For example:

```
db2 "create database partition group mygroup on node (0,1)"
db2 "create regular tablespace mytb in database partition group mygroup
   managed by system using ('mytb')"
```

In the example above, you would then specify the `mytb` table space when you enable the column:

```
dxxadm enable_column mydatabase mytable mycolumn "dad/mydad.dad" -t mytb
```

**Example::**

The following example enables an XML column.

```
dxxadm enable_column SALES_DB SALES_TAB ORDER getstart.dad
   -v sales_order_view -r INVOICE_NUMBER
```

## disable_column option of the dxxadm command

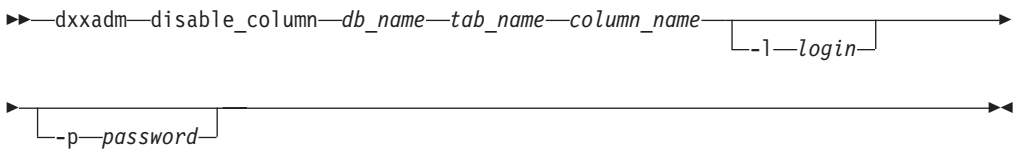
**Purpose:**

Connects to a database and disables the XML-enabled column. When the column is disabled, it can no longer contain XML data types. When an XML-enabled column is disabled, the following actions are performed:

- The XML column usage entry is deleted from the XML_USAGE table.
- The USAGE_COUNT is decremented in the DTD_REF table.
- All triggers that are associated with this column are dropped.
- All side tables that are associated with this column are dropped.

**Important:** You must disable an XML column before dropping an XML table. If an XML table is dropped but its XML column is not disabled, the XML Extender keeps both the side tables that it created and the XML column entry in the XML_USAGE table.

**Syntax:**




**Parameters:**

*Table 19. disable_column parameters*

| Parameter | Description |
|---|---|
| *db_name* | The name of the database in which the data resides. |
| *tab_name* | The name of the table in which the XML column resides. |
| *column_name* | The name of the XML column. |
| -l *login* | The user ID used to connect to the database. If not specified, the current user ID is used. |
| -p *password* | The password used to connect to the database. If not specified, the current password is used. |

**Examples:**
The following example disables an XML-enabled column.

```
dxxadm disable_column SALES_DB SALES_TAB ORDER
```

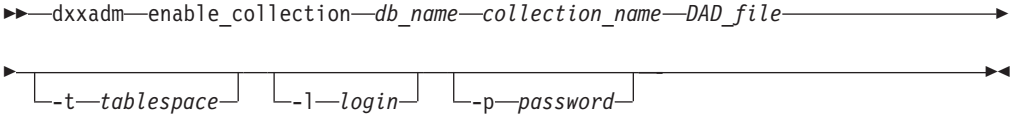
## enable_collection option of the dxxadm command

**Purpose:**

Connects to a database and enables an XML collection according to the specified DAD. When running the XML Extender on a partitioned Enterprise Server Edition environment, check that all of the tables specified in your DAD file included at least one column that is qualified to be a partitioning key. When enabling a collection, the XML Extender does the following tasks:

- Creates an XML collection usage entry in the XML_USAGE table.
- For RDB_node mapping, creates collection tables specified in the DAD if the tables do not exist in the database.

**Syntax:**




**Parameters:**

*Table 20. enable_collection parameters*

| Parameter | Description |
|---|---|
| *db_name* | The name of the database in which the data resides. |
| *collection_name* | The name of the XML collection. |
| *DAD_file* | The name of the DAD file that maps the XML document to the relational tables in the collection. |
| -t *tablespace* | The name of the table space that is associated with the collection. If not specified, the default table space is used. |
| -l *login* | The user ID used to connect to the database. If not specified, the current user ID is used. |

*Table 20. enable_collection parameters (continued)*

| Parameter | Description |
|---|---|
| -p *password* | The password used to connect to the database. If not specified, the current password is used. |

If you are using the partitioned IBM DB2 Enterprise Server Edition and you want to specify a table space while enabling a collection, you need to have specified a node group when you created the table space. For example:

```
db2 "create database partition group mygroup on node (0,1)"
db2 "create regular tablespace mytb in database partition group mygroup
   managed by system using ('mytb')"
```

In the example above, you would then specify the `mytb` table space while enabling the collection.

**Examples:**

The following example enables an XML collection.

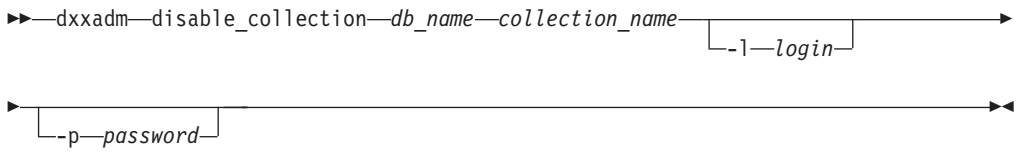
```
dxxadm enable_collection SALES_DB sales_ord
getstart_xcollection.dad -t orderspace
```

## disable_collection option of the dxxadm command

**Purpose:**

Connects to a database and disables an XML-enabled collection. The collection name can no longer be used in the composition (dxxRetrieveXML) and decomposition (dxxInsertXML) stored procedures. When an XML collection is disabled, the associated collection entry is deleted from the XML_USAGE table. Disabling the collection does not drop the collection tables that are created during when you use enable_collection option.

**Syntax:**




**Parameters:**

*Table 21. disable_collection parameters*

| Parameter | Description |
|---|---|
| *db_name* | The name of the database in which the data resides. |
| *collection_name* | The name of the XML collection. |
| -l *login* | The user ID used to connect to the database. If not specified, the current user ID is used. |
| -p *password* | The password used to connect to the database. If not specified, the current password is used. |

**Example::**
The following example disables an XML collection.

```
dxxadm disable_collection SALES_DB sales_ord
```

# Chapter 7. XML Extender user-defined types

User-defined types (UDTs) are data types created by a DB2® application or tool. The XML Extender creates the following user-defined types for use with XML columns:

- XMLVARCHAR
- XMLCLOB
- XMLFILE

The data types are used to define the column in the application table that will be used to store the XML document. You can also store XML documents as files on the file system, by specifying a file name.

All XML Extender's user-defined types have the qualifier DB2XML, which is the schema name of the DB2 XML Extender user-defined types. For example:

```
db2xml.XMLVarchar
```

The XML Extender creates UDTs for storing and retrieving XML documents. Table 22 describes the UDTs.

*Table 22. The XML Extender UDTs*

| User-defined type column | Source data type | Usage description |
|---|---|---|
| XMLVARCHAR | VARCHAR(*varchar_len*) | Stores an entire XML document as VARCHAR inside DB2. You can redefine XMLVARCHAR up to 32 KB. To change the size of an XMLVARCHAR UDT, create the UDT before you enable the database for XML Extender. |
| XMLCLOB | CLOB(*clob_len*) | Stores an entire XML document as a character large object (CLOB) inside DB2. |
| XMLFILE | VARCHAR(512) | Specifies the fully-qualified file name of the local file server. If XMLFILE is specified for the XML column, then the XML Extender stores the XML document in an external server file. You must ensure integrity between the file content, DB2, and the side table created for indexing. |

Where *varchar_len* and *clob_len* are specific to the operating system.

For XML Extender on DB2 on Linux, Unix, and Windows® operating systems, *varchar_len* = 3K and *clob_len* = 2G.

© Copyright IBM Corp. 1999 - 2006

To change the size of an XMLVARCHAR or XMLCLOB UDT, create the UDT before you enable the database for XML Extender.

Procedure:

To change the size of an XMLVARCHAR or XMLCLOB UDT of an enabled database:

1. Back up all data in the XML Extender-enabled database.
2. Drop all XML collection tables or XML column side tables.
3. Disable the database with the `dxxadm disable_db` command.
4. Create the XMLVARCHAR or XMLCLOB user-defined type.
5. Enable the database with the `dxxadm enable_db`command.
6. Re-create and reload the tables.

These UDTs are used only to specify the types of application columns; they do not apply to the side tables that the XML Extender creates.

- "Mapping schemes for XML collections" on page 99
- "Preparing to administer XML Extender" on page 35
- "XML collections as a storage and access method" on page 87
- "XML columns as a storage and access method" on page 71

# Chapter 8. XML Extender user-defined functions

A user–defined function (UDF) is a function that is defined to the database management system and can be referenced in SQL statements. This section describes user-defined functions that are provided by IBM DB2 XML Extender.

## Types of XML Extender user-defined functions

The XML Extender provides functions for storing, retrieving, searching, and updating XML documents, and for extracting XML elements or attributes.

All the UDFs have the schema name DB2XML.

The types of XML Extender functions are described in the following list:

**storage functions**
Storage functions insert intact XML documents in XML-enabled columns as XML data types.

**retrieval functions**
Retrieval functions retrieve XML documents from XML columns in a DB2® database.

**extracting functions**
Extracting functions extract and convert the element content or attribute value from an XML document to the data type that is specified by the function name. The XML Extender provides a set of extracting functions for various SQL data types.

**update function**
The Update function modifies an entire XML document or specified element content or attribute values and returns a copy of an XML document with an updated value, which is specified by the location path.

**Validation functions**
Validation functions validate XML documents against either an XML schema or a DTD.

The XML user-defined functions allow you to perform searches on general SQL data types. Additionally, you can use the DB2 Net Search Extender with XML Extender to perform structural and *full text searches* on text in XML documents. This search capability can be used, for example, to improve the usability of a Web site that publishes large amounts of readable text, such as newspaper articles or *Electronic Data Interchange (EDI)* applications, which have frequently searchable elements or attributes.

Restriction: When using parameter markers in UDFs, a Java™ database (JDBC) restriction requires that the parameter marker for the UDF must be cast to the data type of the column into which the returned data will be inserted.

© Copyright IBM Corp. 1999 - 2006

## UDT and UDF names for XML Extender

The full name of a DB2® function is *schema-name.function-name*, where *schema-name* is an identifier that provides a logical grouping for a set of SQL objects. The schema name for XML Extender UDFs and UDTs is DB2XML. In the documentation, references are made only to the function name.

You can specify UDTs and UDFs without the schema name if you add the schema name to the function path. The function path is an ordered list of schema names. DB2 database uses the order of schema names in the list to resolve references to functions and UDTs. You can specify the function path by specifying the SQL statement SET CURRENT FUNCTION PATH. This statement sets the function path in the CURRENT FUNCTION PATH special register.

**Recommendation:** Add the DB2XML schema name to the function path. By adding this schema name, you can enter XML Extender UDF and UDT names without having to qualify them with DB2XML. The following example shows how to add the DB2XML schema to the function path:

```
SET CURRENT FUNCTION PATH = DB2XML, CURRENT FUNCTION PATH
```

**Restriction:** Do not add DB2XML as the first schema in the function path if you log on with a user ID of DB2XML. DB2XML is automatically set as the first schema when you log on as DB2XML. If you add DB2XML as the first schema in the function path, you will receive an error condition because the function path will start with two DB2XML schemas.

## Storage functions

### Storage functions in XML Extender overview

Use storage functions to insert XML documents into a DB2 database. You can use the default casting functions of a UDT directly in INSERT or SELECT statements. Additionally, XML Extender provides UDFs to take XML documents from sources other than the UDT base data type and convert them to the specified UDT.

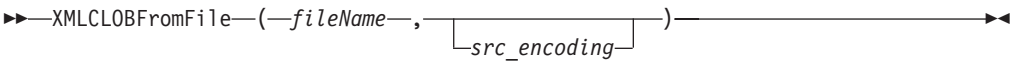
Before you use UDFs to read or write from the server file system, you must set environment variables as described in Appendix C, "XML Extender limits," on page 293.

### XMLCLOBFromFile() function

**Purpose:**

Reads an XML document from a server file and returns the document as an XMLCLOB type.

**Syntax:**




**Parameters:**

*Table 23. XMLCLOBFromFile parameter*

| Parameter | Data type | Description |
|---|---|---|
| *fileName* | VARCHAR(512) | The fully qualified server file name. |
| *src_encoding* | VARCHAR(100) | The encoding of the source file. |

**Results:**
XMLCLOB as LOCATOR
**Example:**
The following example reads an XML document from a file on a server and inserts it into an XML column as an XMLCLOB type. The encoding of the server file is explicitly specified as iso-8859-1.

```
EXEC SQL INSERT INTO sales_tab(ID, NAME, ORDER)
   VALUES('1234', 'Sriram Srinivasan',
           XMLCLOBFromFile('dxx_install/samples/extenders/db2xml
           /xml/getstart.xml
           ', 'iso-8859-1'))
```

where *dxx_install* is the directory where XML Extender is installed.

The column ORDER in the SALES_TAB table is defined as an XMLCLOB type.

## XMLFileFromCLOB() function

**Purpose:**

Reads an XML document as CLOB locator, writes it to an external server file, and returns the file name and path as an XMLFILE type.

**Syntax:**

**Parameters:**

*Table 24. XMLFileFromCLOB() parameters*

| Parameters | Data type | Description |
|---|---|---|
| *buffer* | CLOB as LOCATOR | The buffer containing the XML document. |
| *fileName* | VARCHAR(512) | The fully qualified server file name. |
| *targetencoding* | VARCHAR(100) | The encoding of the output file. |

**Results:**

XMLFILE

**Example:**

The following example reads an XML document as CLOB locator (a host variable with a value that represents a single LOB value in the database server), writes it to

an external server file, and inserts the file name and path as an XMLFILE type in an XML column. The function will encode the output file in ibm-808.

```
EXEC SQL BEGIN DECLARE SECTION;
     SQL TYPE IS CLOB_LOCATOR xml_buff;
EXEC SQL END DECLARE SECTION;

EXEC SQL INSERT INTO sales_tab(ID, NAME, ORDER)
   VALUES('1234', 'Sriram Srinivasan',
           XMLFileFromCLOB(:xml_buff, 'dxx_install/samples/extenders/db2xml
           /xml/getstart.xml', 'ibm-808'))
```

where *dxx_install* is the directory where XML Extender is installed.

The column ORDER in the SALES_TAB table is defined as an XMLFILE type. If you have an XML document in your buffer, you can store it in a server file.

## XMLFileFromVarchar() function

**Purpose:**

Reads an XML document from memory as VARCHAR, writes it to an external server file, and returns the file name and path as an XMLFILE type.

**Syntax:**

**Parameters:**

*Table 25. XMLFileFromVarchar parameters*

| Parameter | Data type | Description |
|---|---|---|
| *buffer* | VARCHAR(3K) | The buffer containing the XML document. |
| *fileName* | VARCHAR(512) | The fully qualified server file name. |
| *targetencoding* | VARCHAR(100) | The encoding of the output file. |

**Results:**

XMLFILE

**Example:**

The following examples reads an XML document from memory as VARCHAR, writes it to an external server file, and inserts the file name and path as an XMLFILE type in an XML column. The function will encode the output file in iso-8859-1.

```
EXEC SQL BEGIN DECLARE SECTION;
      struct { short len; char data[3000]; } xml_buff;
EXEC SQL END DECLARE SECTION;

EXEC SQL INSERT INTO sales_tab(ID, NAME, ORDER)
  VALUES('1234', 'Sriram Srinivasan',
        XMLFileFromVarchar(:xml_buff, 'dxx_install/samples/extenders/db2xml
           /xml/getstart.xml', 'iso-8859-1'))
```

where *dxx_install* is the directory where XML Extender is installed.

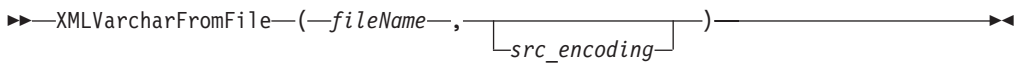
The column ORDER in the SALES_TAB table is defined as an XMLFILE type.

## XMLVarcharFromFile() function

**Purpose:**

Reads an XML document from a server file and returns the document as an XMLVARCHAR type.

**Syntax:**




**Parameters:**

*Table 26. XMLVarcharFromFile parameter*

| Parameter | Data type | Description |
|---|---|---|
| *fileName* | VARCHAR(512) | The fully qualified server file name. |
| *src_encoding* | VARCHAR(100) | The encoding of the source file. |

**Results:**

XMLVARCHAR

**Example:**

The following example reads an XML document from a server file and inserts it into an XML column as an XMLVARCHAR type. The encoding of the server file is explicitly specified as ibm-808.

```
EXEC SQL INSERT INTO sales_tab(ID, NAME, ORDER)
   VALUES('1234', 'Sriram Srinivasan',
           XMLVarcharFromFile('dxx_install/samples/extenders/db2xml
           /xml/getstart.xml', 'ibm-808'))
```

where *dxx_install* is the directory where XML Extender is installed.

In this example, a record is inserted into the SALES_TAB table. The function XMLVarcharFromFile() imports the XML document from a file that is explicitly specified to be encoded in ibm-808 into DB2 database and stores it as a XMLVARCHAR.

# Retrieval functions

## Retrieval functions in XML Extender

The XML Extender provides an overloaded function Content(), which is used for retrieval. This overloaded function refers to a set of retrieval functions that have the same name, but behave differently based on where the data is being retrieved. You can also use the default casting functions to convert an XML UDT to the base data type.

The Content() functions provide the following types of retrieval:

- **Retrieval from external storage at the server to a host variable at the client.**

  You can use Content() to retrieve an XML document to a memory buffer when it is stored as an external server file. You can use Content(): retrieve from XMLFILE to a CLOB for this purpose.
- **Retrieval from internal storage to an external server file**

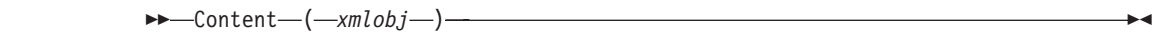
  You can also use Content() to retrieve an XML document that is stored inside DB2 and store it to a server file on the DB2 database server's file system. The following Content() functions are used to store information on external server files:
  - Content(): retrieve from XMLVARCHAR to an external server file
  - Content(): retrieval from XMLCLOB to an external server file

## Content(): retrieve from XMLFILE to a CLOB

**Purpose:**

Retrieves data from a server file and stores it in a CLOB LOCATOR.

**Syntax:**




**Parameters:**

*Table 27. XMLFILE to a CLOB parameter*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLFILE | The XML document. |

**Results:**
CLOB (*clob_len*) as LOCATOR

*clob_len* for DB2 database is 2G.

**Example:**

The following example retrieves data from a server file and stores it in a CLOB locator.

```
char    subsystem[20];
long   retcode = 0, reason = 0;
extern "OS" { int DSNALI(char * functn, ...); }

extern "OS" short DSNTIAR(struct sqlca *sqlca,
                          error_struct *error_message,
                          long *data_len);

EXEC SQL BEGIN DECLARE SECTION;
     SQL TYPE IS CLOB_LOCATOR xml_buff;
EXEC SQL END DECLARE SECTION;

/* Attach to subsystem */
rc = DSNALI("OPEN        ", subsystem, "PLANNAME",
     &retcode, &reason);
if ( retcode != 0 )
{
   /* print error message */
   goto exit;
```

```
}

EXEC SQL DECLARE c1 CURSOR FOR

     SELECT Content(order) from sales_tab
     WHERE sales_person = 'Sriram Srinivasan'

EXEC SQL OPEN c1;

do {
   EXEC SQL FETCH c1 INTO :xml_buff;
   if (SQLCODE != 0) {
      break;
      }
   else {
     /* do with the XML doc in buffer */
     }
   }

EXEC SQL CLOSE c1;

/* Detach from sybsystem */
DSNALI("CLOSE       ", "SYNC", &retcode, &reason);
if ( retcode != 0 ) {
   /* print error message */
}
```

The column ORDER in the SALES_TAB table is of an XMLFILE type, so the Content() UDF retrieves data from a server file and stores it in a CLOB locator.

**Related tasks:**

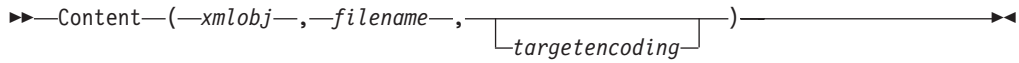
- "Updating and deleting data in XML collections" on page 95

## Content(): retrieve from XMLVARCHAR to an external server file

**Purpose:**

Retrieves the XML content that is stored as an XMLVARCHAR type and stores it in an external server file.

**Syntax:**




**Important:** If a file with the specified name already exists, the content function overrides its content.

**Parameters:**

*Table 28. XMLVarchar to external server file parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR | The XML document. |
| *filename* | VARCHAR(512) | The fully qualified server file name. |

*Table 28. XMLVarchar to external server file parameters (continued)*

| Parameter | Data type | Description |
|---|---|---|
| *targetencoding* | VARCHAR(100) | The encoding of the output file. |

**Results:**

VARCHAR(512)

**Example:**

The following example retrieves the XML content that is stored as XMLVARCHAR type and stores it in an external file located on the server. The UDF encodes the file in 'ibm-808'.

```
CREATE table app1 (id int NOT NULL, order DB2XML.XMLVarchar);
INSERT into app1 values (1, '<?xml version="1.0"?>
   <!DOCTYPE SYSTEM "dxx_install/samples/extenders/db2xml/dtd/getstart.dtd"->

     <Order key="1">
        <Customer>
          <Name>American Motors</Name>
          <Email>parts@am.com</Email>
        </Customer>
        <Part color="black">
          <key>68</key>
          <Quantity>36</Quantity>
          <ExtendedPrice>34850.16</ExtendedPrice>
          <Tax>6.000000e-02</Tax>
          <Shipment>
              <ShipDate>1998-08-19</ShipDate>
              <ShipMode>AIR </ShipMode>
          </Shipment>
          <Shipment>
              <ShipDate>1998-08-19</ShipDate>
              <ShipMode>BOAT </ShipMode>
          </Shipment>
          </Part>
     </Order>');

SELECT DB2XML.Content(order,
   'dxx_install/samples/extenders/db2xml/dad/getstart_column.dad'
   , 'ibm-808')
FROM app1 where ID=1;
```

**Related tasks:**

- "Method for retrieving an XML document" on page 76

**Related reference:**

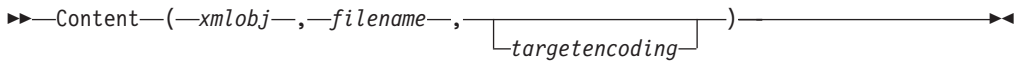
- "Retrieval functions in XML Extender" on page 135

## Content(): retrieval from XMLCLOB to an external server file

**Purpose:**

Retrieves the XML content that is stored as an XMLCLOB type and stores it in an external server file.

**Syntax:**




**Important:** If a file with the specified name already exists, the content function overrides its content.

**Parameters:**

*Table 29. XMLCLOB to external server file parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLCLOB as LOCATOR | The XML document. |
| *filename* | VARCHAR(512) | The fully qualified server file name. |
| *targetencoding* | VARCHAR(100) | The encoding of the output file. |

**Results:**

VARCHAR(512)

**Example:**

The following example retrieves the XML content that is stored as an XMLCLOB type and stores it in an external file located on the server. The UDF encodes the file in 'ibm-808'.

```
CREATE table app1 (id int NOT NULL, order DB2XML.XMLCLOB not logged);

INSERT into app1 values (1, '<?xml version="1.0"?>
  <!DOCTYPE SYSTEM "dxx_install/samples/extenders/db2xml/dtd/getstart.dtd"
->
    <Order key="1">
      <Customer>
          <Name>American Motors</Name>
          <Email>parts@am.com</Email>
      </Customer>
      <Part color="black">
          <key>68</key>
          <Quantity>36</Quantity>
          <ExtendedPrice>34850.16</ExtendedPrice>
          <Tax>6.000000e-02</Tax>
          <Shipment>
              <ShipDate>1998-08-19</ShipDate>
              <ShipMode>AIR </ShipMode>
          </Shipment>
          <Shipment>
              <ShipDate>1998-08-19</ShipDate>
              <ShipMode>BOAT </ShipMode>
          </Shipment>
      </Part>
    </Order>');

SELECT DB2XML.Content(order,
'dxx_install/samples/extenders/db2xml/xml/getstart.xml', 'ibm-808')
 from app1 where ID=1;
```

# Extraction functions

## Extracting functions in XML Extender

The extracting functions extract the element content or attribute value from an XML document and return the requested SQL data types. The XML Extender provides a set of extracting functions for various SQL data types. The extracting functions take two input parameters. The first parameter is the XML Extender UDT, which can be one of the XML UDTs. The second parameter is the location path that specifies the XML element or attribute. Each extracting function returns the value or content that is specified by the location path.

Because some element or attribute values have multiple occurrence, the extracting functions return either a scalar or a table value; the former is called a scalar function, the latter is called a table function.

The examples assume you are using the DB2 database command shell, in which you do not need to type "DB2" at the beginning of each command.

## extractInteger() and extractIntegers()

**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as INTEGER type.

**Syntax:**

**Scalar function:**




**Table function:**




**Parameters:**

*Table 30. extractInteger and extractIntegers function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
INTEGER
**Return codes:**
returnedInteger
**Examples:**
**Scalar function example:**

In the following example, one value is returned when the attribute value of key = ″1″. The value is extracted as an INTEGER. The examples assume that you are using the DB2 command shell, in which you do not need to type “DB2” at the beginning of each command.

```
CREATE TABLE t1(key INT);
INSERT INTO t1 values (
    DB2XML.extractInteger(DB2XML.XMLFile('/samples/extenders/db2xml
    /xml/getstart.xml
    '),
    '/Order/Part[@color="black "]/key'));
SELECT * from t1;
```

**Table function example:**

In the following example, each order key for the sales orders is extracted as INTEGER. The examples assume that you are using the DB2 command shell, in which you do not need to type “DB2” at the beginning of each command.

```
SELECT *
FROM TABLE(
  DB2XML.extractIntegers
   (DB2XML.XMLFile('/samples/extenders/db2xml/xml/getstart.xml'),
   '/Order/Part/key')) AS X;
```

**Related concepts:**

- “UDT and UDF names for XML Extender” on page 132
- “Types of XML Extender user-defined functions” on page 131

**Related reference:**

- “Extracting functions in XML Extender” on page 140

## extractSmallint() and extractSmallints()

**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as SMALLINT type.

**Syntax:**

**Scalar function:**

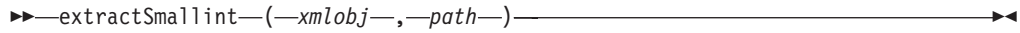
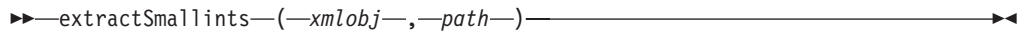



**Table function:**




**Parameters:**

*Table 31. extractSmallint and extractSmallints function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |

*Table 31. extractSmallint and extractSmallints function parameters (continued)*

| Parameter | Data type | Description |
|---|---|---|
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
SMALLINT
**Return codes:**
returnedSmallint

**Examples:**

**Scalar function example:**

In the following example, the value of key in all sales orders is extracted as SMALLINT. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
CREATE TABLE t1(key INT);
INSERT INTO t1 values (
    DB2XML.extractSmallint(db2xml.xmlfile('dxx_install
     /samples/extenders/db2xml/xml/getstart.xml'),
    '/Order/Part[@color="black "]/key'));
SELECT * from t1;
```

**Table function example:**

In the following example, the value of key in all sales orders is extracted as SMALLINT. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
SELECT *
  FROM TABLE(
      DB2XML.extractSmallints(DB2XML.XMLFile('dxx_install
      /samples/extenders/db2xml/xml/getstart.xml'),
      '/Order/Part/key'))   AS X;
```

**Related concepts:**

- "UDT and UDF names for XML Extender" on page 132
- "Types of XML Extender user-defined functions" on page 131

**Related reference:**

- "Extracting functions in XML Extender" on page 140

## extractDouble() and extractDoubles()

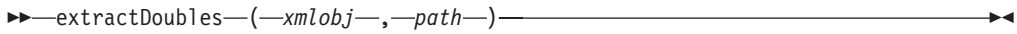
**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as DOUBLE type.

**Syntax:**

**Scalar function:**

**Table function:**




**Parameters:**

*Table 32. extractDouble and extractDoubles function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned Type:**
DOUBLE
**Return codes:**
returnedDouble
**Examples: Scalar function example:**
The following example automatically converts the price in an order from a DOUBLE type to a DECIMAL. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
CREATE TABLE t1(price DECIMAL(9,2));
INSERT INTO t1 values (
    DB2XML.extractDouble(DB2XML.xmlfile('dxx_install
    /samples/extenders/db2xml/xml/getstart.xml'),
    '/Order/Part[@color="black "]/ExtendedPrice'));
SELECT * from t1;
```

**Table function example:**

In the following example, the value of ExtendedPrice in each part of the sales order is extracted as DOUBLE. The examples assume that you are using the DB2 command shell, in which you do not need to type DB2 UDB at the beginning of each command.

```
SELECT CAST(RETURNEDDOUBLE AS DOUBLE)
  FROM TABLE(
      DB2XML.extractDoubles(DB2XML.XMLFile('dxx_install
       /samples/extenders/db2xml/xml/getstart.xml'),
      '/Order/Part/ExtendedPrice'))   AS X;
```

**Related concepts:**

- "UDT and UDF names for XML Extender" on page 132

**Related reference:**

- "Extracting functions in XML Extender" on page 140

## extractReal() and extractReals()

**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as REAL type.

**Syntax:**

**Scalar function:**

**Table function:**

**Parameters:**

*Table 33. extractReal and extractReals function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
REAL
**Return codes:**
returnedReal
**Examples:**
**Scalar function example:**

In the following example, the value of ExtendedPrice is extracted as a REAL. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
CREATE TABLE t1(price DECIMAL(9,2));
INSERT INTO t1 values (
    DB2XML.extractReal(DB2XML.xmlfile('dxx_install
    /samples/extenders/db2xml/xml/getstart.xml'),
    '/Order/Part[@color="black"]/ExtendedPrice'));
SELECT * from t1;
```

**Table function example:**

In the following example, the value of ExtendedPrice is extracted as a REAL. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
SELECT CAST(RETURNEDREAL AS REAL)
  FROM TABLE(
      DB2XML.extractReals(DB2XML.XMLFile('dxx_install
      /samples/extenders/db2xml/xml/getstart.xml'),
      '/Order/Part/ExtendedPrice'))   AS X;
```

**Related concepts:**

- "UDT and UDF names for XML Extender" on page 132
- "Types of XML Extender user-defined functions" on page 131

**Related reference:**

- "Extracting functions in XML Extender" on page 140
- "XML Extender UDF return codes" on page 254

## extractChar() and extractChars()

**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as CHAR type.

**Syntax:**

**Scalar function:**

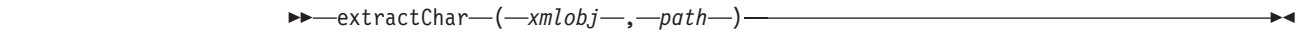
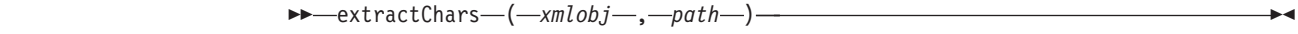



**Table function:**




**Parameters:**

*Table 34. extractChar and extractChars function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
CHAR
**Return codes:**
returnedChar
**Examples:**
**Scalar function example:**

In the following example, the value of Name is extracted as CHAR. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
CREATE TABLE t1(name char(30));
INSERT INTO t1 values (
    DB2XML.extractChar(DB2XML.xmlfile('dxx_install
    /samples/extenders/db2xml/xml/getstart.xml'),
    '/Order/Customer/Name'));
SELECT * from t1;
```

**Table function example:**

In the following example, the value of Color is extracted as CHAR. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
SELECT *
  FROM TABLE(
      DB2XML.extractChars(DB2XML.XMLFile('dxx_install
       /samples/extenders/db2xml/xml/getstart.xml'),
      '/Order/Part/@color'))   AS X;
```

**Related reference:**

- "Extracting functions in XML Extender" on page 140

## extractVarchar() and extractVarchars()

**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as VARCHAR type.

**Syntax:**

**Scalar function:**

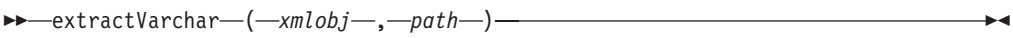
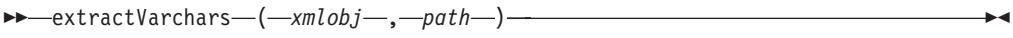



**Table function:**




**Parameters:**

*Table 35. extractVarchar and extractVarchars function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
VARCHAR(4K)
**Return codes:**
returnedVarchar
**Examples:**
**Scalar function example:**

In a database with more than 1000 XML documents that are stored in the column ORDER in the SALES_TAB table, you might want to find all the customers who have ordered items that have an ExtendedPrice greater than 2500.00. The following SQL statement uses the extracting UDF in the SELECT clause:

```
SELECT extractVarchar(Order, '/Order/Customer/Name') from sales_order_view
   WHERE price > 2500.00
```

The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command. The UDF extractVarchar() takes the column ORDER as the input and the location path `/Order/Customer/Name` as the select identifier. The UDF returns the names of the customers. With the WHERE clause, the extracting function evaluates only those orders with an ExtendedPrice greater than 2500.00.

**Scalar function example:**

In the following example, the value of Name is extracted as VARCHAR. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
CREATE TABLE t1(name varchar(30));
INSERT INTO t1 values (
    DB2XML.extractVarchar(DB2XML.xmlfile('dxx_install
     /samples/extenders/db2xml/xml/getstart.xml'),
    '/Order/Customer/Name'));
SELECT * from t1;
```

**Table function example:**

In the following example, the value of Color is extracted as VARCHAR. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
SELECT*
  FROM TABLE(
      DB2XML.extractVarchars(DB2XML.XMLFile('dxx_install
      /samples/xml/getstart.xml'),
      '/Order/Part/@color'))   AS X;
```

**Related concepts:**

- "UDT and UDF names for XML Extender" on page 132
- "Types of XML Extender user-defined functions" on page 131

**Related reference:**

- "Extracting functions in XML Extender" on page 140
- "XML Extender UDF return codes" on page 254

## extractCLOB() and extractCLOBs()

**Purpose:**

Extracts a fragment of XML documents, with element and attribute markup and content of elements and attributes, including sub-elements. This function differs from the other extract functions, which return only the content of elements and attributes. The extractClob(s) functions are used to extract document fragments, whereas extractVarchar(s) and extractChar(s) are used to extract simple values.

**Syntax:**

**Scalar function:**

**Table function:**

**Parameters:**

*Table 36. extractCLOB and extractCLOBs function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
CLOB(10K)
**Return codes:**
returnedCLOB
**Examples:**
**Scalar function example:**

In this example, all name element content and tags are extracted from a purchase order. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
CREATE TABLE t1(name DB2XML.xmlclob);
INSERT INTO t1 values (
    DB2XML.extractClob(DB2XML.xmlfile('dxx_install
     /samples/extenders/db2xml/xml/getstart.xml'),
    '/Order/Customer/Name'));
SELECT * from t1;
```

**Table function example:**

In this example, all of the color attributes are extracted from a purchase order. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
SELECT *
  FROM TABLE(
      DB2XML.extractCLOBs(DB2XML.XMLFile('dxx_install
     /samples/extenders/db2xml/xml/getstart.xml'),
      '/Order/Part/@color'))   AS X;
```

**Related concepts:**

- "Types of XML Extender user-defined functions" on page 131

**Related reference:**

- "Extracting functions in XML Extender" on page 140

## extractDate() and extractDates()

**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as DATE type. The date must be in the format: YYYY-MM-DD.

**Syntax:**

**Scalar function:**

**Table function:**

**Parameters:**

*Table 37. extractDate and extractDates function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
DATE
**Return codes:**
returnedDate
**Examples:**
**Scalar function example:**

In the following example, the value of ShipDate is extracted as DATE. The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

```
CREATE TABLE t1(shipdate DATE);
INSERT INTO t1 values (
    DB2XML.extractDate(DB2XML.xmlfile('dxx_install
    /samples/extenders/db2xml/xml/getstart.xml'),
    '/Order/Part[@color="red   "]/Shipment/ShipDate'));
SELECT * from t1;
```

**Table function example:**

In the following example, the value of ShipDate is extracted as DATE.

```
SELECT *
  FROM TABLE(
      DB2XML.extractDates(DB2XML.XMLFile('dxx_install
    /samples/extenders/db2xml/xml/getstart.xml'),
      '/Order/Part[@color="black "]/Shipment/ShipDate'))   AS X;
```

**Related concepts:**

- "Types of XML Extender user-defined functions" on page 131

**Related reference:**

- "Extracting functions in XML Extender" on page 140
- "XML Extender UDF return codes" on page 254

## extractTime() and extractTimes()

**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as TIME type.

**Syntax:**

**Scalar function:**

**Table function:**

**Parameters:**

*Table 38. extractTime and extractTimes function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
TIME
**Return codes:**
returnedTime
**Examples:**
The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

**Scalar function example:**

```
CREATE TABLE t1(testtime  TIME);
INSERT INTO t1 values (
    DB2XML.extractTime(DB2XML.XMLCLOB(
         '<stuff><data>11.12.13</data></stuff>'), '//data'));
SELECT * from t1;
```

**Table function example:**

```
select *
from table(
    DB2XML.extractTimes(DB2XML.XMLCLOB(
        '<stuff><data>01.02.03</data><data>11.12.13</data></stuff>'),
        '//data')) as x;
```

**Related concepts:**

- "UDT and UDF names for XML Extender" on page 132
- "Types of XML Extender user-defined functions" on page 131

**Related reference:**

- "Extracting functions in XML Extender" on page 140

## extractTimestamp() and extractTimestamps()

**Purpose:**

Extracts the element content or attribute value from an XML document and returns the data as TIMESTAMP type.

**Syntax:**

**Scalar function:**

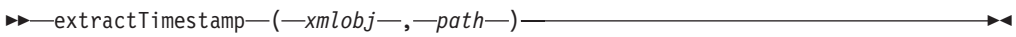
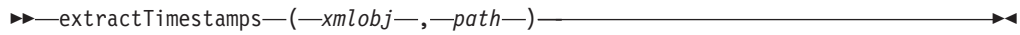



**Table function:**




**Parameters:**

*Table 39. extractTimestamp and extractTimestamps function parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLFILE, or XMLCLOB | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |

**Returned type:**
TIMESTAMP
**Return codes:**
returnedTimestamp
**Examples:**
The examples assume that you are using the DB2 command shell, in which you do not need to type "DB2" at the beginning of each command.

**Scalar function example:**

```
CREATE TABLE t1(testtimestamp  TIMESTAMP);
INSERT INTO t1 values (
    DB2XML.extractTimestamp(DB2XML.XMLCLOB(
        '<stuff><data>2003-11-11-11.12.13.888888</data></stuff>'),
        '//data'));
SELECT * from t1;
```

**Table function example:**

```
select * from
table(DB2XML.extractTimestamps(DB2XML.XMLClob(
    '<stuff><data>2003-11-11-11.12.13.888888
    </data><data>2003-12-22-11.12.13.888888</data></stuff>'),
    '//data')) as x;
```

XML Extender will automatically normalize timestamps extracted from XML documents to comply with the DB2 timestamp format if needed. Timestamps are normalized to the `yyyy-mm-dd-hh.mm.ss.nnnnnn` format or the `yyyy-mm-dd-hh mm.ss.nnnnnn` format. For example:

```
2003-1-11-11.12.13
```

will be normalized to:

```
2003-01-11-11.12.13.000000
```

**Related concepts:**

- "UDT and UDF names for XML Extender" on page 132
- "Types of XML Extender user-defined functions" on page 131

**Related reference:**

- "Extracting functions in XML Extender" on page 140
- "XML Extender UDF return codes" on page 254

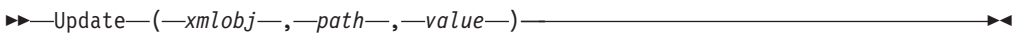
## Update functions in XML Extender

The Update() function updates a specified element or attribute value in one or more XML documents stored in the XML column. You can also use the default casting functions to convert an SQL base type to the XML UDT.

**Purpose**

Takes the column name of an XML UDT, a location path, and a string of the update value and returns an XML UDT that is the same as the first input parameter. With the Update() function, you can specify the element or attribute that is to be updated.

**Syntax**




**Parameters**

*Table 40. The UDF Update parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | XMLVARCHAR, XMLCLOB as LOCATOR | The column name. |
| *path* | VARCHAR | The location path of the element or attribute. |
| *value* | VARCHAR | The update string.<br><br>**Restriction:** The Update function does not have an option to disable output escaping; the output of an extractClob (which is a tagged fragment) cannot be inserted using this function. Use textual values only. |

**Restriction:** Note that the Update UDF supports location paths that have predicates with attributes, but not elements. For example, the following predicate is supported:

```
'/Order/Part[@color="black "]/ExtendedPrice'
```

The following predicate is not supported:

```
'/Order/Part/Shipment/[Shipdate < "11/25/00"]'
```

**Return type**

| Data type | Return type |
|---|---|
| XMLVARCHAR | XMLVARCHAR |
| XMLCLOB as LOCATOR | XMLCLOB |

**Examples**

The following example updates the purchase order handled by the salesperson Sriram Srinivasan.

```
UPDATE sales_tab
   set order = db2xml.update(order, '/Order/Customer/Name', 'IBM')
   WHERE sales_person = 'Sriram Srinivasan'
```

In this example, the content of `/Order/Customer/Name` is updated to `IBM`.

The following examples show instances of an XML document before and after an update.

*Table 41. XML documents before and after an update*

**Example 1:**

**Before:**

```
<?xml version='1.0'   encoding='utf-8'    standalone="yes"?>
<!DOCTYPE book PUBLIC "public.dtd" "system.dtd">
<?pitarget option1='value1' option2='value2'?>
<!-- comment -->
<book>
  <chapter id="1" date='07/01/1997'>
    <!-- first section -->
    <section>This is a section in Chapter
       One.</section>
  </chapter>
  <chapter id="2" date="01/02/1997">
    <section>This is a section in Chapter
       Two.</section>
    <footnote>A footnote in Chapter Two is
        here.</footnote>
  </chapter>
  <price date="12/22/1998" time="11.12.13"
       timestamp="1998-12-22-11.12.13.888888">
       38.281</price>
</book>
```

- Contains white space in the XML declaration
- Specifies a processing instruction
- Contains a comment outside of the root node
- Specifies PUBLIC ExternalID
- Contains a comment inside of root note

**After:**

```
<?xml version='1.0' encoding='utf-8' standalone='yes'?>
<!DOCTYPE book PUBLIC "public.dtd" "system.dtd">
<?pitarget option1='value1' option2='value2'?>
<book>
  <chapter id="1" date="07/01/2003">
        <!-- first section -->
        <section>This is a section in Chapter
         One.</section>
  </chapter>
  <chapter id="2" date="01/02/2003">
        <section>This is a section in Chapter
         Two.</section>
        <footnote>A footnote in Chapter Two
         is here.</footnote>
  </chapter>
  <price date="12/22/2003" time="11.12.13"
       timestamp="2003-12-22-11.12.13.888888">
       60.02</price>
</book>
```

- White space inside of markup is eliminated
- Processing instruction is preserved
- Comment outside of the root node is not preserved
- PUBLIC ExternalID is preserved
- Comment inside of root node is preserved
- Changed value is the value of the <price> element

*Table 41. XML documents before and after an update (continued)*

**Example 2:**

**Before:**

```
<?xml version='1.0'     ?>
<!DOCTYPE book>
<!-- comment -->
<book>
  ...
</book>
```

Contains DOCTYPE declaration without an ExternalID or an internal DTD subset. Not supported.

**After:**

```
<?xml version='1.0'?>
<book>
  ...
</book>
```

DOCTYPE declaration is not reported by the XML parser and not preserved.

**Example 3:**

**Before:**

```
<?xml version='1.0'     ?>
<!DOCTYPE book [ <!ENTITY myDog "Spot"> ]>
<!-- comment -->
<book>
  <chapter id="1" date='07/01/1997'>
       <!-- first section -->
       <section>This is a section in Chapter
         One about my dog &;myDog;.</section>
       ...
  </chapter>
  ...
</book>
```

- Contains white space in markup
- Specifies internal DTD subset
- Specifies entity in text node

**After:**

```
<?xml version='1.0'?>
<!DOCTYPE book>
<book>
  <chapter id="1" date="07/01/1997">
       <!-- first section -->
       <section>This is a section in Chapter
         One about my dog Spot.</section>
          ...
  </chapter>
  ...
</book>
```

- White space in markup is eliminated
- Internal DTD subset is not preserved
- Entity in text node is resolved and replaced

### Usage

When you use the Update function to change a value in one or more XML documents, it replaces the XML documents within the XML column. Based on output from the XML parser, some parts of the original document are preserved,

while others are lost or changed. The following sections describe how the document is processed and provide examples of how the documents look before and after updates.

## How the Update() function processes the XML document

When the Update() function replaces XML documents, it must reconstruct the document based on the XML parser output. Table 42 describes how the parts of the document are handled, with examples.

*Table 42. Update function rules*

| Item or node type | XML document code example | Status after update |
|---|---|---|
| XML declaration | `<?xml  version='1.0'`<br>`encoding='utf-8'`<br>`standalone='yes' >` | The XML declaration is preserved:<br>• Version information is preserved.<br>• Encoding declaration is preserved and appears when specified in the original document.<br>• Standalone declaration is preserved and appears when specified in the original document.<br>• After update, single quotation marks are used to delineate values. |
| DOCTYPE Declaration | `<!DOCTYPE books SYSTEM`<br>`"http://dtds.org/books.dtd" >`<br>`<!DOCTYPE books PUBLIC`<br>`"local.books.dtd"  "http://dtds.org/books.dtd" >`<br>`<!DOCTYPE books>`<br>`-Any of`<br>`<!DOCTYPE books`<br>`( S ExternalID ) ?`<br>`[ internal-dtd-subset ] >`<br>`-Such as`<br>`<!DOCTYPE books`<br>`[ <!ENTITY mydog "Spot"> ] >?`<br>`[ internal-dtd-subset ] >` | The document type declaration is preserved:<br>• Root element name is supported.<br>• Public and system ExternalIDs are preserved and appear when specified in the original document.<br>• Internal DTD subset is *not* preserved. Entities are replaced; defaults for attributes are processed and appear in the output documents.<br>• After the update, double quotation marks are used to delineate public and system URI values.<br>• The current XML4c parser does not report an XML declaration that does not contain an ExternalID or internal DTD subset. After the update, the DOCTYPE declaration would be missing in this case. |

*Table 42. Update function rules (continued)*

| Item or node type | XML document code example | Status after update |
|---|---|---|
| Processing Instructions | `<?xml-stylesheet title="compact" href="datatypes1.xsl" type="text/xsl"?>` | Processing instructions are preserved. |
| Comments | `<!-- comment -->` | Comments inside the root element are preserved.<br><br>Comments outside the root element are discarded. |
| Elements | `<books>`<br>*content*<br>`</books>` | Elements are preserved. |
| Attributes | `id='1' date="01/02/2003"` | Attributes of elements are preserved.<br>• After update, double quotation marks are used to delineate values.<br>• Data within attributes is lost.<br>• Entities are replaced. |
| Text Nodes | `This section is about my dog &mydog;.` | Text nodes (element content) are preserved.<br>• Data within text nodes is lost.<br>• Entities are replaced. |

### Multiple occurrence

When a location path is provided in the Update() UDF, the content of every element or attribute with a matching path is updated with the supplied value. This means that if a document has multiple occurring location paths, the Update() function replaces the existing values with the value provided in the *value* parameter.

You can specify a predicate in the *path* parameter to provide distinct locations paths to prevent unintentional updates. The Update() UDF supports location paths that have predicates with attributes, but not elements.

## Validation functions

DB2 XML Extender offers two user defined functions (UDFs) that validate XML documents against either an XML schema or a DTD. An element in an XML document is valid according to a given schema if the associated element type rules are satisfied. If all elements are valid, the whole document is valid.

The validation functions return 1 if the document is valid, or they return 0 and write an error message in the trace file if the document is invalid. The functions are:

**db2xml.svalidate:**
Validates an XML document instance against the specified schema.

**db2xml.dvalidate:**
Validates an XML document instance against the specified DTD.

# SVALIDATE() function

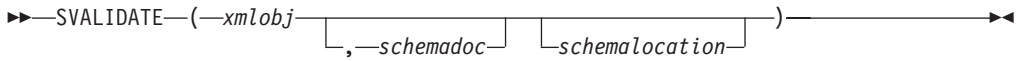
This function validates an XML document against a specified schema (or the one named in the XML document) and returns 1 if the document is valid or 0 if not. This function assumes that an XML document and a schema exist on the file system or as a CLOB in DB2.

Before you run the SVALIDATE function, ensure that XML Extender is enabled with your database by running the following command:

```
dxxadm enable_db mydbname
```

If the XML document fails the validation, an error message is written to the XML Extender trace file. Enable the trace before executing the SVALIDATE command.

## Syntax




## Parameters

*Table 43. The SVALIDATE parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj*<br><br>Required. Only one xmlobj allowed. | VARCHAR(256) | The fully-qualified file name of the XML document to be verified. |
| | CLOB(2G) | XML column that contains the document to be verified. |
| *schemadoc*<br><br>Optional. No default. Only one schemadoc allowed. | VARCHAR(256) | The fully-qualified file name of the schema document (with no target namespace) with which to validate the XML document. |
| | CLOB(2G) | XML column that contains the schema (with no target namespace) with which to validate the XML document. |
| *schemalocation*<br><br>Optional. No default. Not allowed if nonamespace schemadoc is data type CLOB. | VARCHAR(256) | Namespace URI and fully-qualified schema document file name pairs to use for validation. You can specify one or more pairs separated by a space. |

## Examples

**Example 1:** This example validates `equiplog2001.xml` against the schema that is specified within the document.

```
db2 values db2xml.svalidate("/home/jean/xml/equiplog2001.xml")
```

**Example 2:** This example validates an XML document using the specified schema, and both the document and schema are stored in DB2 database tables.

```
db2 select db2xml.svalidate(doc,schema) from equiplogs where id=1
```

**Example 3:**

This example validates the XML document in CLOB column xml in a_table against the specified nonamespace schema schema1.xsd and the specified list of namespace schemas, schema2.xsd and schema3.xsd, that are associated with target namespaces http://ns1 and http://ns2:

```
select db2xml.svalidate(
     xml,
     'xsd/schema1.xsd',
     'http://ns1 /dxxsamples/schema2.xsd
      http://ns2 /dxxsamples/schema3.xsd')
from a_table where id=2
```

**Example 4:**

This example validates test.xml against the specified list of schemas, schema1.xsd and schema2.xsd, that are associated with target namespaces, http://ns1 and http://ns2.

```
db2 values db2xml.svalidate(
     '/dxxsamples/xml/test.xml',
     ' ',
     'http://ns1 /dxxsamples/xsd/schema1.xsd http://ns2
      /dxxsamples/xsd/schema2.xsd')
```

# DVALIDATE() function

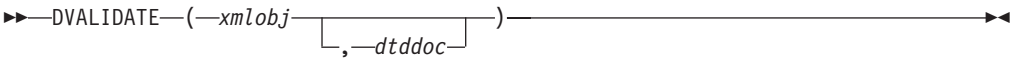
This function validates an XML document against a specified DTD (or the one named in the XML document) and returns 1 if the document is valid or 0 if not. This function assumes that an XML document and a DTD exist on the file system or as a CLOB in DB2.

Before executing the DVALIDATE function, ensure that XML Extender is enabled with your database by running the following command:

```
dxxadm enable_db mydbname
```

If the XML document fails the validation, an error message is written to the XML Extender trace file. Enable the trace before executing the DVALIDATE command.

## Syntax




## Parameters

*Table 44. The DVALIDATE parameters*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | VARCHAR(256) | Fully-qualified path of the XML document to be verified. |
| | CLOB(2G) | XML column that contains the document to be verified. |

*Table 44. The DVALIDATE parameters (continued)*

| Parameter | Data type | Description |
|---|---|---|
| *dtddoc* | VARCHAR(256) | Fully-qualified path of the DTD document. |
| | CLOB(2G) | XML column that contains the DTD, which is either from the DTD_REF table or from a regular table. |

## Examples

**Example 1:** This example validates `equiplog2001.xml` against the DTD that is specified within the document.

```
db2 values db2xml.dvalidate(/home/jean/xml/equiplog2001.xml)
```

**Example 2:** This example validates an XML document using the specified DTD, and both the document and DTD are in the file system.

```
db2 values db2xml.dvalidate (c:/xml/equiplog.xml,c:/xml/dtds/equip.dtd)
```

**Example 3:** This example validates an XML document using the specified DTD, and both the document and DTD are stored in DB2 database tables.

```
db2 values db2xml.dvalidate (doc,dtdid) from equiplogs, db2xml.dtd_ref \
    where dtdid="equip.dtd"
```

**Related reference:**

- "Starting the trace for XML Extender" on page 253

# Chapter 9. Document access definition (DAD) files

## Creating a DAD file for XML columns

This task is part of the larger task of defining and enabling an XML column.

To access your XML data and enable columns for XML data in an XML table, you need to define a document access definition (DAD) file. This file defines the attributes and key elements of your data that need to be searched within the column. For XML columns, the DAD file primarily specifies how documents stored within it are to be indexed. The DAD file also specifies a DTD or schema to use for validating documents that are inserted into the XML column. DAD files are stored as a CLOB data type, and their size limit is 100 KB.

**Prerequisites:**

Before you create the DAD file, you need to:

- Decide which elements or attributes you expect to use often in your search. The elements or attributes that you specify are extracted into the side tables for fast searches by the XML Extender.
- Define the location path to represent each element or attribute indexed in a side table. You must also specify the type of data that you want the element or attribute to be converted to.

**Procedure:**

To create a DAD file:

1. Create a new document in a text editor and type the following syntax:

   ```
   <?XML version="1.0"?>
   <!DOCTYPE DAD SYSTEM <"path/dtd/dad.dtd">.
   ```

   *"path/dtd/dad.dtd "* is the path and file name of the DTD for the DAD file. A DTD is provided in *dxx_install*\samples\extenders\db2xml\dtd
2. Insert DAD tags after the lines from step 1.

   ```
   <DAD>
   </DAD>
   ```

   This element will contain all the other elements.
3. Specify validation for the document and the column:
   - If you want to validate your entire XML document against a DTD or schema before it is inserted into the database:
     - Insert the following tag to validate the document using a DTD:

       ```
       <dtdid>dtd_name.dtd</dtdid>
       ```
     - Insert the following tag to validate the document using a schema:

       ```
       <schemabindings>
       <nonamespacelocation location="path/schema_name.xsd"/>
       </schemabindings>
       ```
     - To validate the document insert the following tag:

       ```
       <validation>YES</validation>
       ```

© Copyright IBM Corp. 1999 - 2006

Otherwise, insert the following tag:

```
<validation>NO</validation>
```

4. Insert <Xcolumn> </Xcolumn> tags to specify that you are using XML columns as the access and storage method for your XML data.
5. Specify side tables. For each side table that you want to create:
   a. Specify a <table></table> tag. For example:

      ```
      <table name="person_names">
      </table>
      ```

   b. Inside the table tags, insert a <column> tag for each column that you want the side table to contain. Each column has four attributes: name, type, path and, multi_occurrence.

      **Example:**

      ```
      <table name="person_names">>
      <column name ="fname"
              type="varchar(50)"
              path="/person/firstName"
              multi_occurrence="NO"/>
      <column name ="lname"
              type="varchar(50)"
              path="/person/lastName"
              multi_occurrence="NO"/>
      </table>
      ```

      Where:

      **name** Specifies the name of the column that is created in the side table.

      **type** Indicates the SQL data type in the side table for each indexed element or attribute

      **path** Specifies the location path in the XML document for each element or attribute to be indexed

      **multi_occurrence**
      Indicates whether the element or attribute referred to by the path attribute can occur more than once in the XML document. The possible values for **multi_occurrence** are **YES** or **NO**. If the value is **NO**, then multiple columns can be specified per table. If the value is **YES**, you can specify only one column in the side table.
6. Save your file with a DAD extension.

The following example shows a complete DAD file:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx_install\samples\extenders\db2xml\dtd\dad.dtd">
<DAD>
<dtdid>C:\SG246130\code\person.dtd</dtdid>
<validation>YES</validation>
<Xcolumn>
  <table name="person_names">
   <column name="fname"
     type="varchar(50)"
     path="/person/firstName"
     multi_occurrence="NO"/>
   <column name="lname"
     type="varchar(50)"
     path="/person/lastName"
     multi_occurrence="NO"/>
</table>
<table name="person_phone_number">
   <column name="pnumber"
```

```
     type="varchar(20)"
     path="/person/phone/number"
     multi_occurrence="YES"/>
</table>
<table name="person_phone_number">
   <column name="pnumber"
     type="varchar(20)"
     path="/person/phone/number"
     multi_occurrence="YES"/>
</table>
<table name="pesron_phone_type">
   <column name="ptype"
     type="varchar(20)"
     path="/person/phone/type"
     multi_occurrence="YES"/>
</table>
<Xcolumn>
</DAD>
```

Now that you created a DAD file, the next step to defining and enabling an XML column is to create the table in which your XML documents will be stored.

**Related concepts:**

- "DAD files for XML collections"
- "XML collections as a storage and access method" on page 87

## DAD files for XML collections

For XML collections, the DAD file maps the structure of the XML document to the DB2® tables from which you compose the document. You can also decompose documents to the DB2 tables using the DAD file.

For example, if you have an element called <Tax> in your XML document, you need to map <Tax> to a column called TAX. You use the DAD file to define the relationship between the XML data and the relational data.

You must specify the DAD file either while enabling a collection, or when you are using the DAD file in stored procedures for XML collections. The DAD is an XML-formatted document, residing at the client. If you choose to validate XML documents with a DTD, the DAD file can be associated with that DTD. When used as the input parameter of the XML Extender stored procedures, the DAD file has a data type of CLOB. This file can be up to 100 KB.

To specify the XML collection access and storage method, use the <Xcollection> tag in your DAD file.

**<Xcollection>**

Specifies that the XML data is either to be decomposed from XML documents into a collection of relational tables, or to be composed into XML documents from a collection of relational tables.

An XML collection is a set of relational tables that contains XML data. Applications can enable an XML collection of any user tables. These user tables can be tables of existing business data or tables that XML Extender recently created.

The DAD specifies mappings between elements or attributes of the XML document and columns of tables. It uses the following kinds of nodes to specify these mappings:

**root_node**
Specifies the root element of the document.

**element_node**
Identifies an element, which can be the root element or a child element.

**text_node**
Represents the CDATA text of an element.

**attribute_node**
Represents an attribute of an element.

Figure 13 shows a fragment of the mapping that is used in a DAD file. The nodes map the XML document content to table columns in a relational table.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "dtd\dad.dtd">
<DAD>
  ...
<Xcollection>
<SQL_stmt>
       ...
</SQL_stmt>
<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE Order SYSTEM
                 "dxx_install\samples\db2xml\dtd\getstart.dtd"</doctype>
<root_node>
 <element_node name="Order">      --> Identifies the element <Order>
  <attribute_node name="key">     --> Identifies the attribute "key"
   <column name="order_key"/>     --> Defines the name of the column,
                                      "order_key", to which the
                                      element and attribute are
                                      mapped
  </attribute_node>
  <element_node name="Customer">  --> Identifies a child element of
                                      <Order> as <Customer>
   <text_node>                    --> Specifies the CDATA text for
                                      the element <Customer>
    <column name="customer">      --> Defines the name of the column,
                                      "customer", to which the child
                                      element is mapped
   </text_node>
  </element_node>
        ...
 </element_node>

      ...
</root_node>
</Xcollection>
</DAD>
```

*Figure 13. Node definitions for the XML document as mapped to the XML collection table*

In this example, the first two columns have elements and attributes mapped to them.

The XML Extender also supports processing instructions for stylesheets, using the <stylesheet> element. It must be inside the root node of the DAD file, with the doctype and prolog defined for the XML document. For example:

```
<Xcollection>
...
<prolog>...</prolog>
<doctype>...</doctype>
<stylesheet>?xml-stylesheet type="text/css" href="order.css"?</stylesheet>
```

```
<root_node>...</root_node>
...

</Xcollection>
```

Use any text editor to create and update a DAD file.

**Related concepts:**

- "Mapping schemes for XML collections" on page 99

## SQL composition

You can compose XML documents using columns with the same name. Selected columns with the same name, even if from diverse tables, must be identified by a unique alias so that every variable in the select clause of the SQL statement is different. The following example shows how you would give unique aliases to columns that have the same names.

```
<SQL_stmt>select o.order_key as oorder_key,
                 key, customer_name, customer_email,
                 p.part_key,  p.order_key as porder_key,
                 color, qty, price, tax, ship_id, date, mode
            from order_tab o.part_tab p
            order by oorder_key, part_key</SQL_stmt>
```

You can also compose XML documents using columns with generated random values. If an SQL statement in a DAD file has a random value, you must give the random value function an alias to use it in the ORDER BY clause. This requirement is necessary because the value is not associated with any column in a given table. See the alias for generate_unique at the end of the ORDER BY clause in the following example.

```
<SQL_stmt>select o.order_key, customer_name,customer_email,
                 p.part_key,color,qty,price,tax,ship_id,
                 date, mode
          from order_tab o,part_tab p,
               table(select substr(char(timestamp(generate_unique())),16)
as ship_id, date, mode,
                           part_key
                      from ship_tab) s
          where o.order_key=1 and p.price>2000 and
                o.order_key=o.order_key and s.part_key
          order by order_key, part_key,ship_id</SQL_stmt>
```

## RDB node composition

The following restrictions apply to RDB node composition:

- The condition associated with any non-root_node RDB node DAD file must compare against a literal.
- The condition associated with any lower level RDB node in the DAD must compare against a literal.
- The condition associated with a root_node describes the relationship between the tables involved in the RDB node composition. An example is a primary foreign key relationship.
- Each equality in the condition associated with a top-level RDB_node specifies the join relationship between columns of two tables and is applied separately from the other equalities. That is, all the predicates connected by AND do not apply simultaneously for a single join condition; they simulate an outer join

when the document is composed. The parent-child relationship between each pair of tables is determined by their relative nesting in the DAD file. For example:

```
<condition>order_tab.order_key=part_tab.order_key AND
part_tab.part_key=ship_tab.part_key</condition>
```

## Composition from rows that have null values

You can use columns that have null values to compose XML documents.

The following example illustrates how you can generate an XML document from a table *MyTable* which has a row containing a null value in column *Col 1*. The DAD used in the example is *nullcol.dad*.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx\dtd\dad.dtd">
<DAD>
<validation>NO validation>NO>
<Xcollection>
<SQL_stmt>SELECT 1 as X, Col1 FROM MyTable order by X, Col1<\SQL_stmt>
<prolog>?xml version="1.0"?prolog>?xml version="1.0"?>
<doctype>!DOCTYPE Order SYSTEM "e:\t3xml\x.dtd">
<root_node>
<element_node name="MyColumn">
<element_node name="Column1" multi_occurrence="YES">
  <text_node>
  <column name="Col1"/>
  </text_node>
 </element_node>
</element_node>
</rootт_node>
</Xcollection>
</DAD>
```

```
MyTable
_________
| Col 1 |
|_______|
|  1    |
|_______|
|  3    |
|_______|
|  -    |
|_______|
```

Run tests2x mydb nullcol.dad result_tab or use dxxGenXML to produce the following document. There is no third Column1 element because that row in the table had a null value.

```
<?xml version="1.0"?>
<!DOCTYPE Order SYSTEM "e:\t3xml\x.dtd">
<MyColumn>
 <Column1>1</Column1>
 <Column1>3</Column1>
</MyColumn>
```

# DTD for the DAD file

This topic describes the document type declarations (DTD) for the document access definition (DAD) file. The DAD file itself is a tree-structured XML document and requires a DTD. The DTD file name is `dad.dtd`. The following example shows the DTD for the DAD file:

```
<?xml encoding="US-ASCII"?>

<!ELEMENT DAD ((schemabindings |dtdid)?, validation, (Xcolumn | Xcollection))>
<!ELEMENT dtdid (#PCDATA)>
<!ELEMENT schemabindings (nonamespacelocation)>
<!ELEMENT nonamespacelocation EMPTY>
<!ATTLIST nonamespacelocation location CDATA #REQUIRED>
<!ELEMENT validation (#PCDATA)>
<!ELEMENT Xcolumn (table*)>
<!ELEMENT table (column*)>
<!ATTLIST table name CDATA #REQUIRED
                key CDATA #IMPLIED
                orderBy CDATA #IMPLIED>
<!ELEMENT column EMPTY>
<!ATTLIST column
                    name CDATA #REQUIRED
                    type  CDATA #IMPLIED
                    path  CDATA #IMPLIED
                    multi_occurrence CDATA #IMPLIED>
<!ELEMENT Xcollection (SQL_stmt?, prolog, doctype?, stylesheet?, root_node)>
<!ELEMENT SQL_stmt  (#PCDATA)>
<!ELEMENT prolog  (#PCDATA)>
<!ELEMENT stylesheet  (#PCDATA)>
<!ELEMENT doctype  (#PCDATA)>
<!ELEMENT root_node (element_node)>
<!ELEMENT element_node (RDB_node?,
                        attribute_node*,
                        text_node?,
                        element_node*,
                        namespace_node*,
                        process_instruction_node*,
                        comment_node*)>
<!ATTLIST element_node
                   name             CDATA #REQUIRED
                   ID               CDATA #IMPLIED
                   multi_occurrence CDATA "NO"
                   BASE_URI         CDATA #IMPLIED> <!--ignored  -->
<!ELEMENT attribute_node (column | RDB_node)>
<!ATTLIST attribute_node
                   name CDATA #REQUIRED>
<!ELEMENT text_node (column | RDB_node)>
<!ELEMENT RDB_node (table+, column?, condition?)>
<!ELEMENT condition  (#PCDATA)>
<!ELEMENT comment_node  (#PCDATA)>  <!-- ignored  -->
<!ELEMENT namespace_node EMPTY> <!-- ignored  -->
<!ATTLIST namespace_node
                    name      CDATA #IMPLIED
                    value      CDATA #IMPLIED>
<!ELEMENT process_instruction_node (#PCDATA)> <!-- ignored  -->
```

The DAD file has four major elements:

- DTDID
- validation
- Xcolumn
- Xcollection

Xcolumn and Xcollection have child element and attributes that aid in the mapping of XML data to relational tables in DB2. The following list describes the major elements and their child elements and attributes. Syntax examples are taken from the previous example.

**DTDID element**

DTDs that are provided to XML Extender are stored in the DTD_REF table. Each DTD is identified by a unique ID that is provided in the `DTDID` tag of

the DAD file. The DTDID points to the DTD that validates the XML documents, or guides the mapping between XML collection tables and XML documents. For XML collections, this element is required only for validating input and output XML documents. For XML columns, this element is needed only to validate input XML documents. The DTDID must be the same as the SYSTEM ID specified in the doctype of the XML documents.

*Syntax:* `<!ELEMENT dtdid (#PCDATA)>`

**validation element**
Indicates whether the XML document is to be validated with the DTD for the DAD. If YES is specified, then the DTDID must also be specified.

**Syntax:** `<!ELEMENT validation(#PCDATA)>`

**Xcolumn element**
Defines the indexing scheme for an XML column. It is composed of zero or more tables.

**Syntax:** `<!ELEMENT Xcolumn (table*)>`Xcolumn has one child element, table.

**table element**
Defines one or more relational tables created for indexing elements or attributes of documents stored in an XML column.

**Syntax:**

```
<!ELEMENT table (column+)>
 <!ATTLIST table name CDATA #REQUIRED
       key CDATA #IMPLIED
       orderBy CDATA #IMPLIED>
```

The table element has one mandatory and two implied attributes:

**name attribute**
Specifies the name of the side table.

**key attribute**
The primary single key of the table.

**orderBy attribute**
The names of the columns that determine the sequence order of multiple-occurring element text or attribute values when generating XML documents.

The table element has one child element:

**column element**
Maps an attribute of a CDATA node from the input XML document to a column in the table.

**Syntax:**

```
<!ATTLIST column
                     name CDATA #REQUIRED
                     type  CDATA #IMPLIED
                     path  CDATA #IMPLIED
                     multi_occurrence CDATA #IMPLIED>
```

The column element has the following attributes:

**name attribute**
Specifies the name of the column. It is the alias name of the location path that identifies an element or attribute.

**type attribute**
Defines the data type of the column. It can be any SQL data type.

**path attribute**
Shows the location path of an XML element or attribute and must be the simple location path as specified in Table 3.1.a.

**multi_occurrence attribute**
Specifies whether this element or attribute can occur more than once in an XML document. Values can be YES or NO.

**Xcollection**
Defines the mapping between XML documents and an XML collection of relational tables.

**Syntax:**

```
<!ELEMENT Xcollection(SQL_stmt?, prolog, doctype, root_node)>
```

`Xcollection` has the following child elements:

**SQL_stmt**
Specifies the SQL statement that XML Extender uses to define the collection. Specifically, the statement selects XML data from the XML collection tables, and uses the data to generate the XML documents in the collection. The value of this element must be a valid SQL statement. It is only used for composition, and only a single `SQL_stmt` is allowed.

**Syntax:** `<!ELEMENT SQL_stmt  #PCDATA >`

**prolog**
The text for the XML prolog. The same prolog is supplied to all documents in the entire collection. The value of `prolog` is fixed.

**Syntax:** `<!ELEMENT prolog  #PCDATA>`

**doctype**
Defines the text for the XML document type definition.

**Syntax:**

```
<!ELEMENT doctype (#PCDATA | RDB_node)*>
```

`doctype` is used to specify the DOCTYPE of the resulting document. Define an explicit value. This value is supplied to all documents in the entire collection.

`doctype` has one child element:

**root_node**
Defines the virtual root node. `root_node` must have one required child element, `element_node`, which can be used only once. The `element_node` under the `root_node` is actually the `root_node` of the XML document.

**Syntax:** `<!ELEMENT root_node(element_node)>`

**RDB_node**
Defines the DB2 database table where the content of an XML element or value of an XML attribute is to be stored or from where

it will be retrieved. rdb_node is a child element of element_node, text_node, and attribute_node and has the following child elements:

**table** Specifies the table in which the element or attribute content is stored.

**column**
Specifies the column in which the element or attribute content is stored.

**condition**
Specifies a condition for the column. Optional.

**element_node**
Represents an XML element. It must be defined in the DAD specified for the collection. For the `RDB_node` mapping, the root `element_node` must have an `RDB_node` to specify all tables containing XML data for itself and all of its child nodes. It can have zero or more `attribute_nodes` and child `element_nodes`, as well as zero or one `text_node`. For elements other than the root element no RDB_node is needed.

**Syntax:**

An `element_node` is defined by the following child elements:

**RDB_node**
(Optional) Specifies tables, column, and conditions for XML data. The RDB_node for an element needs to be defined only for the RDB_node mapping. In this case, one or more tables must be specified. The column is not needed because the element content is specified by its text_node. The condition is optional, depending on the DTD and query condition.

**child nodes**
Optional: An `element_node` can also have the following child nodes:

**element_node**
Represents child elements of the current XML element.

**attribute_node**
Represents attributes of the current XML element.

**text_node**
Represents the CDATA text of the current XML element.

**attribute_node**
Represents an XML attribute. It is the node that defines the mapping between an XML attribute and the column data in a relational table.

**Syntax:**

The `attribute_node` must have definitions for a `name` attribute, and either a `column` or a `RDB_node` child element. `attribute_node` has the following attribute:

**name** The name of the attribute.

`attribute_node` has the following child elements:

**column**
Used for the SQL mapping. The column must be specified in the SELECT clause of `SQL_stmt`.

**RDB_node**
Used for the RDB_node mapping. The node defines the mapping between this attribute and the column data in the relational table The table and column must be specified. The condition is optional.

**text_node**
Represents the text content of an XML element. It is the node that defines the mapping between an XML element content and the column data in a relational table.

**Syntax:** It must be defined by a `column` or an `RDB_node` child element:

**column**
Needed for the SQL mapping. In this case, the column must be in the SELECT clause of `SQL_stmt`.

**RDB_node**
Needed for the RDB_node mapping. The node defines the mapping between this text content and the column data in the relational table. The `table` and `column` must be specified. The condition is optional.

**Related concepts:**

- "DAD files for XML collections" on page 163

**Related tasks:**

- "Dynamically overriding values in the DAD file"

## Dynamically overriding values in the DAD file

**Procedure:**

For dynamic queries you can use two optional parameters to override conditions in the DAD file: *override* and *overrideType*. Based on the input from *overrideType*, the application can override the <SQL_stmt> tag values for SQL mapping or the conditions in RDB_nodes for RDB_node mapping in the DAD.

These parameters have the following values and rules:

*overrideType*
This parameter is a required input parameter (IN) that flags the type of the *override* parameter. The *overrideType* parameter has the following values:

**NO_OVERRIDE**
Specifies not to override a condition in the DAD file.

**SQL_OVERRIDE**
Specifies to override the SQL statement in the DAD file.

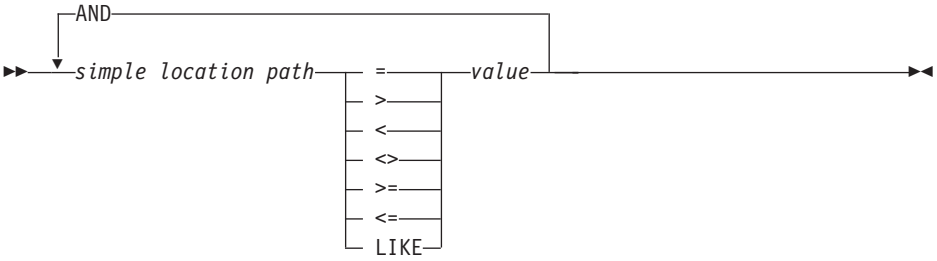
**XML_OVERRIDE**
Specifies to override a condition in the DAD file with a Location Path-based condition.

*override*
This parameter is an optional input parameter (IN) that specifies the override condition for the DAD file. The syntax of the input value corresponds to the value specified on the *overrideType* parameter:

- If you specify NO_OVERRIDE, the input value is a NULL string.
- If you specify SQL_OVERRIDE, the input value is a valid SQL statement.

  To use SQL_OVERRIDE, you must use the SQL mapping scheme in the DAD file. The input SQL statement overrides the SQL statement specified by the <SQL_stmt> element in the DAD file.
- If you specify XML_OVERRIDE, the input value is a string that contains one or more expressions.

  If you use XML_OVERRIDE and an expression, you must use the RDB_node mapping scheme in the DAD file. The input XML expression overrides the RDB_node condition specified in the DAD file. The expression uses the following syntax:




This syntax has the following components:

*simple location path*
Specifies a simple location path, using syntax defined by XPath.

**operators**
The SQL operators shown in the syntax diagram can have a space to separate the operator from the other parts of the expression.

Spaces around the operators are optional. Spaces are mandatory around the LIKE operator.

*value*
A numeric value or a string enclosed in single quotation marks.

**AND**
And is treated as a logical operator on the same location path. If a simple location path is specified more than once in the override string, then all the predicates for that simple location path are applied simultaneously.

If you specify XML_OVERRIDE, the condition for the RDB_node in the text_node or attribute_node that matches the simple location path is overridden by the specified expression.

XML_OVERRIDE is not completely XPath compliant. The simple location path is used only to identify the element or attribute that is mapped to a column.

The following examples use SQL_OVERRIDE and XML_OVERRIDE to show dynamic override.

**Example 1:** A stored procedure using SQL_OVERRIDE. In this example, the <xcollection> element in the DAD file must have an <SQL_stmt> element. The *override* parameter overrides the value of <SQL_stmt>, by changing the price to be greater than 50.00, and the date to be greater than 1998-12-01.

```
include "dxx.h"
include "dxxrc.h"

       EXEC SQL INCLUDE SQLCA;
       EXEC SQL BEGIN DECLARE SECTION;
         char    collection[32];    /* dad buffer */
         char    result_tab[32];    /* name of the result table */
         char    override[256];     /* override, SQL_stmt */
         short   overrideType;      /* defined in dxx.h */
         short   max_row;           /* maximum number of rows */
         short   num_row;           /* actual number of rows */
         long    returnCode;        /* return error code */
         char    returnMsg[1024];   /* error message text */
         short   rtab_ind;
         short   collection_ind;

         short   ovtype_ind;
         short   ov_ind;
         short   maxrow_ind;
         short   numrow_ind;
         short   returnCode_ind;
         short   returnMsg_ind;

         EXEC SQL END DECLARE SECTION;

         /* create table */
         EXEC SQL CREATE TABLE xml_order_tab (xmlorder XMLVarchar);

         /* initialize host variable and indicators */
         strcpy(collection,"sales_ord");
         strcpy(result_tab,"xml_order_tab");
         sprintf(override,"%s %s %s %s %s %s %s",
                   "SELECT o.order_key, customer, p.part_key,
                    quantity, price,", "tax, ship_id, date, mode ",
                   "FROM order_tab o, part_tab p,",
                   "table(select substr(char(timestamp
                    (generate_unique())),16)",
                   "as ship_id, date, mode from ship_tab)  s",
                   "WHERE p.price > 50.00 and s.date >'1998-12-01' AND",
                   "p.order_key = o.order_key and s.part_key = p.part_key");
        overrideType = SQL_OVERRIDE;
         max_row = 500;
         num_row = 0;
         returnCode = 0;
         msg_txt[0] = '\0';
         collection_ind = 0;
         rtab_ind = 0;
         ov_ind = 0;
         ovtype_ind = 0;
         maxrow_ind = 0;
         numrow_ind = -1;
         returnCode_ind = -1;
         returnMsg_ind = -1;

         /* Call the stored procedure */
         EXEC SQL CALL db2xml.dxxRetrieveXML(:collection:collection_ind;
                :result_tab:rtab_ind,
                :overrideType:ovtype_ind,:override:ov_ind,
                :max_row:maxrow_ind,:num_row:numrow_ind,
                :returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

**Example 2:** A stored procedure using XML_OVERRIDE. In this example, the <collection> element in the DAD file has an RDB_node for the root element_node. The *override* value is XML-content based. The XML Extender converts the simple location path to the mapped DB2 database column.

```
include "dxx.h"
include "dxxrc.h"

       EXEC SQL INCLUDE SQLCA;
       EXEC SQL BEGIN DECLARE SECTION;
         char    collection[32];  /* dad buffer */
         char    result_tab[32];  /* name of the result table */
         char    override[256];   /* override, XPATH condition */
         short   overrideType;    /* defined in dxx.h */
         short   max_row;         /* maximum number of rows */
         short   num_row;         /* actual number of rows */
         long    returnCode;      /* return error code */
         char    returnMsg[1024]; /* error message text */
         short   dadbuf_ind;
         short   rtab_ind;
         short   collection_ind;
         short   ovtype_ind;
         short   ov_ind;
         short   maxrow_ind;
         short   numrow_ind;
         short   returnCode_ind;
         short   returnMsg_ind;

         EXEC SQL END DECLARE SECTION;

         /* create table */
         EXEC SQL CREATE TABLE xml_order_tab (xmlorder XMLVarchar);

         /* initialize host variable and indicators */
         strcpy(collection,"sales_ord");
         strcpy(result_tab,"xml_order_tab");
         sprintf(override,"%s %s",
                   "/Order/Part/Price > 50.00 AND ",
                   "Order/Part/Shipment/ShipDate > '1998-12-01'");
        overrideType = XML_OVERRIDE;
         max_row = 500;
         num_row = 0;
         returnCode = 0;
         msg_txt[0] = '\0';
         collection_ind = 0;
         rtab_ind = 0;
         ov_ind = 0;
         ovtype_ind = 0;
         maxrow_ind = 0;
         numrow_ind = -1;
         returnCode_ind = -1;
         returnMsg_ind = -1;

         /* Call the stored procedure */
         EXEC SQL CALL db2xml.dxxRetrieveXML(:collection:collection_ind;
                :result_tab:rtab_ind,
                :overrideType:ovtype_ind,:override:ov_ind,
                :max_row:maxrow_ind,:num_row:numrow_ind,
                :returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

### Multiple Overrides

XML Extender supports multiple overrides on the same path. All overrides specified to the RDB node will be accepted.

You can specify multiple XML overrides on the same location path to refine set conditions in your search. In the following example, an XML document is composed from the two tables using the test.dad file.

*Table 45. Department Table*

| Department Number | Department Name |
|---|---|
| 10 | Engineering |
| 20 | Operations |
| 30 | Marketing |

*Table 46. Employee Table*

| Employee Number | Department Number | Salary |
|---|---|---|
| 123 | 10 | $98,000.00 |
| 456 | 10 | $87,000.00 |
| 111 | 20 | $65,000.00 |
| 222 | 20 | $71,000.00 |
| 333 | 20 | $66,000.00 |
| 500 | 30 | $55,000.00 |

The DAD file `test.dad` illustrated below contains a condition comparing the variable *deptno* with the value 10. To expand the search to greater than 10 and less than 30, you must override this condition. You must set the override parameter when calling dXXGenXML as follows:

```
/ABC.com/Department>10 AND /ABC.com/Department<30

<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "C:\dxx_xml\test\dtd\dad.dtd">
<DAD>
<dtdid>E:\dtd\lineItem.dtd</dtdid>
<validation>NO</validation>
<Xcollection>
<porlog>?xmol version="1.0"?</prolog>
<doctype>!DOCTYPE Order SYSTEM "C:\dxx_xml\test\dtd\LineItem.dtd"</doctype>
<root_node>
<element_node name="ABC.com">
<TDB_node>
<table name="dept" key="deptno"/>
<table name="empl" key="emplno"/>
<condition>dept deptno=empl.deptno</condition>
</RDB_node>

<element_node name="Department" multi_occurrence="YES">
<text_node>
<RDB_node>
<table name="dept"/>
<column name="deptno">
<condition>deptno=10</condition><RDB_node></RDB_node><text_node></text_node>
<element_node name="Employees" multi_occurrence="YES">

<text_node>

<RDB_node>

<table name="dept"><column name="deptnot"><condition>deptno=10</condition>
</table></RDB_node></text_node>
<element_node name="Employees" multi_occurence="YES">
```

```
<element_node name="EmployeeNo">

<text_node>

<RDB_node>

<table name="empl"><column name="emplno"><condition>emplno<500</condition>
</table></RDB_node></text_node></element_node>
<element_node name="Salary">

<text_node>

<RDB_node>

<table name="empl"><column name="salary"><condition>salary>5000.00</condition>
</table></RDB_node></text_node></element_node></element_node></element_node>
```

To compose an XML document without an override, enter tests2x mydb test.dad result_tab or you can invoke dxxGenXML without setting an override. This will generate a document similar to this:

```
<?xml version="1.0">
<!DOCTYPE Order SYSTEM "C:\dxx_xml\test\dtd\LineItem.dtd">
<ABC.com>
<Department>10
<Employees>
<EmployeeNo>123</EmployeeNO>
<Salary>98,000.00</Salary>
</Employees>
<Employees>
<EmployeeNo>456</EmployeeNo>
<Salary>87,000.00</Salary>
</Employees>
</Department>
</ABC.COM>
```

To override the DAD file you can invoke dxxGenXML as mentioned above, or you can run the test2x program with the specified conditions:

tests2x mydb test.dad result_tab -o 2 ″/ABC.com/Department>10 AND /ABC.com/Department<30″

```
<?xml version="1.0">
<!DOCTYPE Order SYSTEM "C:\dxx_xml\test\dtd\LineItem.dtd">
<ABC.com>
<Department>20
<Employees>
<EmployeeNo>111</EmployeeNo>
<Salary>65,000.00</Salary>
</Employees>
<EmployeeNo>222</EmployeeNo>
<Salary>71,000.00</Salary>
</Employees>
<Employees>
<EmployeeNo>333</EmployeeNo>
<Salary>66,000.00</Salary>
</Employees>
</Department>
</ABC.com>
```

**Related concepts:**

- “DAD files for XML collections” on page 163

**Related reference:**

- "DTD for the DAD file" on page 166

# Chapter 10. XML Extender stored procedures

## XML Extender stored procedures - Overview

The XML Extender provides stored procedures for administration and management of XML columns and collections. These stored procedures can be called from the DB2 client. The client interface can be embedded in SQL, ODBC, or JDBC. See the section on stored procedures in the *IBM DB2 Administration Guide* for details about how to call stored procedures.

The stored procedures use the schema DB2XML, which is the schema name of the XML Extender.

The XML Extender provides three types of stored procedures:

**Administration stored procedures**
assist users in completing administrative tasks

**Composition stored procedures**
generate XML documents using data in existing database tables

**Decomposition stored procedures**
break down or shred incoming XML documents and store data in new or existing database tables

Ensure that you include the XML Extender external header files in the program that calls stored procedures. The header files are located in the *"dxx_install*\dxx\samples\extenders\db2xml\include" directory, where *dxx_install* is the directory where you installed DB2 XML Extender. The header files are:

**dxx.h** The XML Extender defined constant and data types

**dxxrc.h** The XML Extender return code

The syntax for including these header files is:

```
#include "dxx.h"
#include "dxxrc.h"
```

Make sure that the path of the include files is specified in your makefile with the compilation option.

## Calling XML Extender stored procedures

You can use XML Extender in different operating systems from a single client application, by writing the stored procedure names in both uppercase and lowercase. To call the stored procedures in this way, use the `result_colname` and `valid_colname` versions of the composition stored procedures. Using this method gives you the following benefits:

- You can use these stored procedures in all DB2 Database environments because you can include many columns in the result table. The versions of the stored procedures that do not support `result_colname` and `valid_colname` require exactly one column in the result table.

© Copyright IBM Corp. 1999 - 2006

- You can use a declared temporary table as your result table. Your temporary table is identified by a schema that is set to "session". Declared temporary tables enable you to support multi-user client environments.

Use uppercase when you call the DB2 XML Extender stored procedures to access the stored procedures consistently across platforms.

**Prerequisites:** Bind your database with the XML Extender stored procedure and DB2 CLI bind files. You can use a sample command file, `getstart_prep.cmd`, to bind the files. This command file is in the "c:\cmd" directory. To bind:

1. Connect to the database. For example:
   ```
   db2 "connect to SALES_DB"
   ```
2. Change to the "c:\bnd" directory and bind the XML Extender to the database.
   ```
   db2 "bind @dxxbind.lst"
   ```
3. Change to the "c:\bnd" directory and bind the CLI to the database.
   ```
   db2 "bind @db2cli.lst"
   ```
4. Terminate the connection.
   ```
   db2 "terminate"
   ```

**Procedure:**

Call XML Extender using the following syntax:

```
CALL DB2XML.function_entry_point
```

Where:

*function_entry_point*
Specifies the name of the function.

In the CALL statement, the arguments that are passed to the stored procedure must be host variables, not constants or expressions. The host variables can have null indicators.

See samples for calling stored procedures in the `dxx_install/samples/extenders/db2xml/c` and `dxx_install/samples/extenders/db2xml/cli` directories. In the `dxx_install/samples/extenders/db2xml/c` directory, SQX code files are provided to call XML collection stored procedures using embedded SQL. In the `dxx_install/samples/extenders/db2xml/cli` directory, the sample files show how to call stored procedures using the Call Level Interface (CLI).

## Stored Procedures that return CLOBs

If you have CLOB files that are larger than 1 MB, you can redefine the stored procedure parameter by creating and running a file containing the following commands:

```
drop procedure db2xml.dxxGenXMLClob;

create procedure db2xml.dxxGenXMLClob(
          in     dadBuf        clob(100K),
          in     overrideType  integer,
          in     override      varchar(32672),
          out    resultDoc     clob(1M),
          out    valid         integer,
          out    numDocs       integer,
          out    returnCode    integer,
```

```
          out    returnMsg     varchar(1024)
          )
          external name 'db2xml!dxxGenXMLClob'
          specific DB2XML.DXXGENXMLCLOB
          language C
          parameter style SQL
          not deterministic
          fenced
          null call;

drop procedure db2xml.dxxRetrieveXMLClob;

create procedure db2xml.dxxRetrieveXMLClob(
          in   collectionName    varchar(128),
          in   overrideType      integer,
          in   override          varchar(32672),
          out  resultDoc         clob(1M),
          out  valid             integer,
          out  numDocs           integer,
          out  returnCode        integer,
          out  returnMsg         varchar(1024)
          )
          external name 'db2xml!dxxRetrieveXMLClob'
          specific DB2XML.DXXRETRIEVEXMLCLOB
          language C
          parameter style SQL
          not deterministic
          fenced
```

**To specify the CLOB length:** Open the file in an editor and modify the *resultDoc* parameter shown in the following example:

```
out resultDoc clob(clob_size),
```

If more than one document is generated, the stored procedure returns the first document.

**Size recommendation:** The size limit of the *resultDoc* parameter depends on your system setup. Be aware that the amount specified in this parameter is the amount allocated by JDBC, regardless of the size of the document. The size should accommodate your largest XML files, but should not exceed 1.5 gigabytes.

To run the command file from the DB2 command line and directory where the file is located, enter:

```
db2 -tf crtgenxc.db2
```

**Related tasks:**


- "Calling XML Extender stored procedures" on page 179


# XML Extender administration stored procedures

## XML Extender administration stored procedures - Overview

These stored procedures are used for administration tasks, such as enabling or disabling an XML column or collection. They are called by the XML Extender administration wizard and the administration command **dxxadm**.

- dxxEnableDB()
- dxxDisableDB()
- dxxEnableColumn()

- dxxDisableColumn()
- dxxEnableCollection()
- dxxDisableCollection()

## dxxEnableDB() stored procedure

**Purpose:**

Enables the database. When the database is enabled, the XML Extender creates the following objects:

- The XML Extender user-defined types (UDTs).
- The XML Extender user-defined functions (UDFs).
- The XML Extender DTD repository table, DTD_REF, which stores DTDs and information about each DTD.
- The XML Extender usage table, XML_USAGE, which stores common information for each column and collection that is enabled for XML Extender.

**Syntax:**

```
DB2XML.dxxEnableDB(char(dbName) dbName,       /* input */
           long           returnCode,   /* output */
           varchar(1024)  returnMsg)    /* output */
```

**Parameters:**

*Table 47. dxxEnableDB() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *dbName* | The database name. | IN |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Related concepts:**

- "XML Extender administration stored procedures - Overview" on page 181
- Chapter 13, "XML Extender administration support tables," on page 251

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179
- "Enabling databases for XML" on page 50

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxDisableDB() stored procedure

**Purpose:**

Disables the database. When XML Extender disables the database, it drops the following objects:

- The XML Extender user-defined types (UDTs).
- The XML Extender user-defined functions (UDFs).

- The XML Extender DTD repository table, DTD_REF, which stores DTDs and information about each DTD.
- The XML Extender usage table, XML_USAGE, which stores common information for each column that is enabled for XML Extender and for each collection.

**Important:** You must disable all XML columns before attempting to disable a database. The XML Extender cannot disable a database that contains columns or collections that are enabled for XML Extender.

**Syntax:**

```
DB2XML.dxxDisableDB(char(dbName)     dbName,       /* input */
                    long          returnCode,   /* output */
                    varchar(1024) returnMsg)    /* output */
```

**Parameters:**

*Table 48. dxxDisableDB() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *dbName* | The database name. | IN |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Related concepts:**

- "XML Extender administration stored procedures - Overview" on page 181
- Chapter 13, "XML Extender administration support tables," on page 251

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxEnableColumn() stored procedure

**Purpose:**

Enables an XML column. When enabling a column, the XML Extender completes the following tasks:

- Determines whether the XML table has a primary key; if not, the XML Extender alters the XML table and adds a column called DXXROOT_ID.
- Creates side tables that are specified in the DAD file with a column containing a unique identifier for each row in the XML table. This column is either the root_id that is specified by the user, or it is the DXXROOT_ID that was named by the XML Extender.
- Creates a default view for the XML table and its side tables, optionally using a name you specify.

**Syntax:**

```
DB2XML.dxxEnableColumn(char(dbName) dbName,      /* input */
                       char(tbName) tbName,      /* input */
                       char(colName) colName,    /* input */
```

```
CLOB(100K)     DAD,         /* input */
char(tablespace) tablespace,  /* input */
char(defaultView) defaultView, /* input */
char(rootID) rootID,      /* input */
long           returnCode,  /* output */
varchar(1024)  returnMsg)   /* output */
```

**Parameters:**

*Table 49. dxxEnableColumn() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *dbName* | The database name. | IN |
| *tbName* | The name of the table containing the XML column. | IN |
| *colName* | The name of the XML column. | IN |
| *DAD* | A CLOB containing the DAD file. | IN |
| *tablespace* | The table space that contains the side tables other than the default table space. If not specified, the default table space is used. | IN |
| *defaultView* | The name of the default view joining the application table and side tables. | IN |
| *rootID* | The name of the single primary key in the application table that is to be used as the root ID for the side table. | IN |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Related concepts:**

- "XML columns as a storage and access method" on page 71
- "XML Extender administration stored procedures - Overview" on page 181

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxDisableColumn() stored procedure

**Purpose:**

Disables the XML-enabled column.

**Syntax:**

```
DB2XML.dxxDisableColumn(char(dbName) dbName,      /* input */
                char(tbName) tbName,      /* input */
                char(colName) colName,     /* input */
                long          returnCode,  /* output */
                varchar(1024)  returnMsg)   /* output */
```

**Parameters:**

*Table 50. dxxDisableColumn() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *dbName* | The database name. | IN |
| *tbName* | The name of the table containing the XML column. | IN |
| *colName* | The name of the XML column. | IN |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxEnableCollection() stored procedure

**Purpose:**

Enables an XML collection that is associated with an application table.

**Syntax:**

```
dxxEnableCollection(char(dbName) dbName,      /* input */
                   char(colName) colName,     /* input */
                   CLOB(100K)     DAD,         /* input */
                   char(tablespace) tablespace,  /* input */
  long           returnCode,  /* output */
                   varchar(1024)  returnMsg)   /* output */
```

**Parameters:**

*Table 51. dxxEnableCollection() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *dbName* | The database name. | IN |
| *colName* | The name of the XML collection. | IN |
| *DAD* | A CLOB containing the DAD file. | IN |
| *tablespace* | The table space that contains the side tables other than the default table space. If not specified, the default table space is used. | IN |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Related concepts:**

- "XML collections as a storage and access method" on page 87
- "XML Extender administration stored procedures - Overview" on page 181

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxDisableCollection() stored procedure

**Purpose:**

Disables an XML-enabled collection, removing markers that identify tables and columns as part of a collection.

**Syntax:**

```
dxxDisableCollection(char(dbName) dbName,      /* input */
                     char(colName) colName,     /* input */
                     long           returnCode,  /* output */
                     varchar(1024)  returnMsg)   /* output */
```

**Parameters:**

*Table 52. dxxDisableCollection() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *dbName* | The database name. | IN |
| *colName* | The name of the XML collection. | IN |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

# XML Extender composition stored procedures

## XML Extender composition stored procedures - Overview

The composition stored procedures dxxGenXML(), dxxRetrieveXML(), dxxGenXMLCLOB(), and dxxRetrieveXMLCLOB() are used to generate XML documents using data in existing database tables. The dxxGenXML() stored procedure takes a DAD file as input; it does not require an enabled XML collection. The dxxRetrieveXML() stored procedure takes an enabled XML collection name as input.

The following performance enhancements have been made for composition stored procedures, dxxGenXMLClob and DXXRetrieveXMLClob.

- On UNIX® and Windows® operating systems, the length of the override parameter has been has been increased from 1 KB to 32 KB.

  The 1 KB override imposed a restriction on the length of the SQL statement for SQL composition. The restriction encouraged the use of database views to reduce the length of the required SQL statement. However, database views can sometimes incur additional pathlength because of view materialization. With a long override, the strong need for views is reduced.
- The requirement for an intermediate result table has been removed.
- By using these stored procedures:
  - You reduce the instruction path length because there is no need to create result tables.
  - You simplify your programming.
- Use the stored procedures that require an intermediate result table if you want to produce more than one document.

## dxxGenXML() stored procedure

**Purpose:**

Constructs XML documents using data that is stored in the XML collection tables that are specified by the <Xcollection> in the DAD file and inserts each XML document as a row into the result table. You can also open a cursor on the result table and fetch the result set.

To provide flexibility, dxxGenXML() lets the user specify the maximum number of rows to be generated in the result table. This decreases the amount of time the application must wait for the results during any trial process. The stored procedure returns the number of actual rows in the table and any error information, including error codes and error messages.

To support dynamic query, dxxGenXML() takes an input parameter, *override*. Based on the input *overrideType*, the application can override the SQL_stmt for SQL mapping or the conditions in RDB_node for RDB_node mapping in the DAD file. The input parameter *overrideType* is used to clarify the type of the *override*.

**Syntax:**

```
dxxGenXML(CLOB(100k) DAD, /* input */
varchar(resultTabName) resultTabName,/* input */
varchar(resultColumn) result_column,/* input */
varchar(validColumn) valid_column, /* input */
integer overrideType, /* input */
varchar(varchar_value) override, /* input */
integer maxRows, /* input */
integer numRows, /*output*/
integer returnCode, /*output*/
varchar(1024) returnMsg) /*output*/
```

Where the varchar_value is 32672 for Windows and UNIX, and 16366 for i5/OS and z/OS.

**Parameters:**

*Table 53. dxxGenXML() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *DAD* | A CLOB containing the DAD file. | IN |
| *resultTabName* | The name of the result table, which should exist before the call. The table contains only one column of either XMLVARCHAR or XMLCLOB type. | IN |
| *result_column* | The name of the column in the result table in which the composed XML documents are stored. | IN |
| *valid_column* | The name of the column that indicates whether the XML document is valid when it is validated against a document type definition (DTD). | IN |
| *overrideType* | A flag to indicate the type of the following *override* parameter:<br>• **NO_OVERRIDE**: No override.<br>• **SQL_OVERRIDE**: Override by an SQL_stmt.<br>• **XML_OVERRIDE**: Override by a Location Path-based condition. | IN |
| *override* | Overrides the condition in the DAD file. The input value is based on the *overrideType*.<br>• **NO_OVERRIDE**: A NULL string.<br>• **SQL_OVERRIDE**: A valid SQL statement. Using this *overrideType* requires that SQL mapping is used in the DAD file. The input SQL statement overrides the SQL_stmt in the DAD file.<br>• **XML_OVERRIDE**: A string that contains one or more expressions that are separated by the word AND. Using this *overrideType* requires that RDB_node mapping is used in the DAD file. | IN |
| *varchar_value* | The size of the VARCHAR. | IN |
| *maxRows* | The maximum number of rows in the result table. | IN |
| *numRows* | The actual number of generated rows in the result table. | OUT |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Examples:**

The following example fragment assumes that a result table is created with the name of XML_ORDER_TAB, and that the table has one column of XMLVARCHAR type. A complete, working sample is located in `samples/extenders/db2xml/c/tests2x.sqx`.

```
#include "dxx.h"
#include "dxxrc.h"

              EXEC SQL INCLUDE SQLCA;
              EXEC SQL BEGIN DECLARE SECTION;
              SQL TYPE is CLOB(100K) dad;            /* DAD */


              SQL TYPE is CLOB_FILE  dadFile;       /* dad file */
              char             result_tabname[160]; /*name of the result table*/
              char             result_colname[129]; /*name of the column name for
                                                        the result document*/
              char             result_valname[129]; /*name of the column name for
                                                        validation result*/
              char             verride[2];    /* override, will set to NULL*/
              short            overrideType;  /* defined in dxx.h */
              short            max_row;       /* maximum number of rows */
              short            num_row;       /* actual number of rows */
              long             returnCode;    /* return error code */
              char             returnMsg[1024]; /* error message text */
              short            dad_ind;
              short            rtab_ind;
              short            rcol_ind;
              short            rval_ind;
              short            ovtype_ind;
              short            ov_inde;
              short            maxrow_ind;
              short            numrow_ind;
              short            returnCode_ind;
              short            returnMsg_ind;

              EXEC SQL END DECLARE SECTION;

              /* create table */
              EXEC SQL CREATE TABLE xml_order_tab (xmlorder XMLVarchar);

              /* read data from a file to a CLOB */
              strcpy(dadfile.name,"dxxinstall/dad/litem3.dad");
              dadfile.name_length = strlen("dxxinstall/dad/litem3.dad");
              dadfile.file_options = SQL_FILE_READ;
              EXEC SQL VALUES (:dadfile) INTO :dad;
              strcpy(result_tabname,"xml_order_tab");
              override[0] = '\0';
              overrideType = NO_OVERRIDE;
              max_row = 500;
              num_row = 0;
              returnCode = 0;
              msg_txt[0] = '\0';
              collection_ind = 0;
              dad_ind = 0;
              rtab_ind = 0;
              rcol_ind = 0;
              rval_ind = 0;
              ov_ind = -1;
              ovtype_ind = 0;
              maxrow_ind = 0;
              numrow_ind = -1;
              returnCode_ind = -1;
              returnMsg_ind = -1;

              /* Call the stored procedure */
              EXEC SQL CALL db2xml.dxxGenXML(:dad:dad_ind;
                     :result_tabname:rtab_ind,
                     :result_colname:rcol_ind,
                     :result_valname:rval_ind,
```

```
:overrideType:ovtype_ind,:override:ov_ind,
:max_row:maxrow_ind,:num_row:numrow_ind,
:returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

**Related concepts:**

- "XML Extender composition stored procedures - Overview" on page 186

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179
- "Composing XML collections by using RDB_node mapping" on page 61
- "Composing XML documents by using SQL mapping" on page 58

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxRetrieveXML() stored procedure

**Purpose:**

The stored procedure dxxRetrieveXML() serves as a means for retrieving decomposed XML documents. As input, dxxRetrieveXML() takes a name of an enabled XML collection, the name of the created result table, and the maximum number of rows to be returned. It returns a result set of the result table, the actual number of rows in the result set, an error code, and message text.

To support dynamic query, dxxRetrieveXML() takes an input parameter, *override*. Based on the input *overrideType*, the application can override the SQL_stmt for SQL mapping or the conditions in RDB_node for RDB_node mapping in the DAD file. The input parameter *overrideType* is used to clarify the type of the *override*.

The requirements of the DAD file for dxxRetrieveXML() are the same as the requirements for dxxGenXML(). The only difference is that the DAD is not an input parameter for dxxRetrieveXML(), but it is the name of an enabled XML collection.

**Syntax:**

```
dxxRetrieveXML(char(collectionName) collectionName,   /* input */
               char(resultTabName) resultTabName,     /* input */
               char(resultColumn) result_column,
         char(validColumn) valid_column,
         integer        overrideType,     /* input */
        varchar(varchar_value) override,
               integer        maxRows,          /* input */
               integer        numRows,          /* output */
               long           returnCode,       /* output */
               varchar(1024)  returnMsg)        /* output */
```

Where *varchar_value* is 32672 for Windows and UNIX and 16366 for iSeries and z/OS.

**Parameters:**

*Table 54. dxxRetrieveXML() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *collectionName* | The name of an enabled XML collection. | IN |

*Table 54. dxxRetrieveXML() parameters (continued)*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *resultTabName* | The name of the result table, which should exist before the call. The table contains only one column of either XMLVARCHAR or XMLCLOB type. | IN |
| *resultColName* | The name of the column in the result table in which the composed XML documents are stored. | IN |
| *resultValidCol* | The name of the column that indicates whether the XML document is valid when it is validated against a document type definition (DTD). | IN |
| *overrideType* | A flag to indicate the type of the following *override* parameter:<br>• **NO_OVERRIDE**: No override.<br>• **SQL_OVERRIDE**: Override by an SQL_stmt.<br>• **XML_OVERRIDE**: Override by a Location Path-based condition. | IN |
| *override* | Overrides the condition in the DAD file. The input value is based on the *overrideType*.<br>• **NO_OVERRIDE**: A NULL string.<br>• **SQL_OVERRIDE**: A valid SQL statement. Using this *overrideType* requires that SQL mapping is used in the DAD file. The input SQL statement overrides the SQL_stmt in the DAD file.<br>• **XML_OVERRIDE**: A string that contains one or more expressions separated by the word AND. Using this *overrideType* requires that RDB_node mapping is used in the DAD file. | IN |
| *maxRows* | The maximum number of rows in the result table. | IN |
| *numRows* | The actual number of generated rows in the result table. | OUT |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Examples:**

The following fragment is an example of a call to dxxRetrieveXML(). In this example, a result table is created with the name of XML_ORDER_TAB, and it has one column of XMLVARCHAR type. A complete, working sample is located in *dxx_install*/samples/extenders/db2xml/c/qcsrc(rtrx).

```
#include "dxx.h"
#include "dxxrc.h"
```

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
  char    collection[32];      /* dad buffer */
  char    result_tabname[160]; /* name of the result table */
  char    result_colname[129]; /*name of the column name for
                                  the result document*/
  char    result_valname[129]; /*name of the column name for
                                     validation result*/

  char    override[2];       /* override, will set to NULL*/
  short   overrideType;      /* defined in dxx.h */
  short   max_row;           /* maximum number of rows */
  short   num_row;           /* actual number of rows */
  long    returnCode;        /* return error code */
  char    returnMsg[1024];   /* error message text */
  short   dadbuf_ind;
  short   rtab_ind;
  short   rcol_ind;
  short   rval_ind;
  short   ovtype_ind;
  short   ov_inde;
  short   maxrow_ind;
  short   numrow_ind;
  short   returnCode_ind;
  short   returnMsg_ind;

  EXEC SQL END DECLARE SECTION;

  /* create table */
  EXEC SQL CREATE TABLE xml_order_tab (xmlorder XMLVarchar);

  /* initialize host variable and indicators */
  strcpy(collection,"sales_ord");
  strcpy(result_tabname,"xml_order_tab");
  override[0] = '\0';
  overrideType = NO_OVERRIDE;
  max_row = 500;
  num_row = 0;
  returnCode = 0;
  msg_txt[0] = '\0';
  collection_ind = 0;
  rtab_ind = 0;
  rcol_ind = 0;
  rval_ind = 0;
  ov_ind = -1;
  ovtype_ind = 0;
  maxrow_ind = 0;
  numrow_ind = -1;
  returnCode_ind = -1;
  returnMsg_ind = -1;

  /* Call the stored procedure */
  EXEC SQL CALL db2xml.dxxRetrieve(:collection:collection_ind;
         :result_tabname:rtab_ind,
         :result_colname:rcol_ind,
         :result_calname:rval_ind,
         :overrideType:ovtype_ind,:override:ov_ind,
         :max_row:maxrow_ind,:num_row:numrow_ind,
         :returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

**Related concepts:**

- "XML Extender composition stored procedures - Overview" on page 186

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179

- "Composing XML collections by using RDB_node mapping" on page 61
- "Composing XML documents by using SQL mapping" on page 58

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxGenXMLClob stored procedure

**Purpose:**

As input, dxxGenXMLClob takes a buffer containing the DAD. It constructs XML documents using data that is stored in the XML collection tables that are specified by the <Xcollection> in the DAD and returns the first and typically the only XML document generated into the *resultDoc* CLOB.

**Syntax:**

```
dxxGenXMLClob(CLOB(100k)                DAD             /*input*/
              integer                   overrideType,    /*input*/
              varchar(varchar_value)    override,       /*input*/
              CLOB(1M)                  resultDoc,       /*output*/
              integer                   valid,           /*output*/
              integer                   numDocs,         /*output*/
              long                      returnCode,      /*output*/
              varchar(1024)             returnMsg)       /*output*/
```

Where *varchar_value* is 32672 for Windows and UNIX and 16366 for i5/OS and z/OS.

**Parameters:**

*Table 55. dxxGenXMLClob parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *DAD* | A CLOB containing the DAD file. | IN |
| *overrideType* | A flag to indicate the type of *override* parameter:<br>**NO_OVERRIDE**<br>No override.<br>**SQL_OVERRIDE**<br>Override by an SQL_stmt<br>**XML_OVERRIDE**<br>Override by a Location Path-based condition. | IN |

*Table 55. dxxGenXMLClob parameters (continued)*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *override* | Overrides the condition in the DAD file. The input value is based on the *overrideType*.<br>**NO_OVERRIDE**<br>A NULL string.<br>**SQL_OVERRIDE**<br>A valid SQL statement. Using this *overrideType* requires that SQL mapping be used in the DAD file. The input SQL statement overrides the SQL_stmt in the DAD file.<br>**XML_OVERRIDE**<br>A string that contains one or more expressions separated by the word AND. Using this *overrideType* requires that RDB_node mapping be used in the DAD file | IN |
| *resultDoc* | A CLOB that contains the composed XML document. | OUT |
| *valid* | valid is set as follows:<br>• If VALIDATION=YES then valid=1 for successful validation or valid=0 for unsuccessful validation.<br>• If VALIDATION=NO then valid=NULL. | OUT |
| *numDocs* | The number of XML documents that would be generated from the input data.<br>**Note:** Currently only the first document is returned. | OUT |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

The CLOB parameter size is 1 MB. If you have CLOB files that are larger than 1 MB, XML Extender provides a command file to redefine the stored procedure parameter. Download the `crtgenxc.zip` file from the DB2 XML Extender Web site. This ZIP file contains the following programs:

**crtgenxc.db2**
For use on XML Extender V7.2 FixPak 5 and later for UNIX and Windows.

**crtgenxc.iseries**
For use with XML Extender for iSeries

**crtgenxc.zos.jcl and crtgenxc.zos.cmd**
For use with XML Extender for OS/390 V7, APAR PQ58249 and later.

**To specify the CLOB length:** Open the file in an editor and modify the *resultDoc* parameter, shown in the following example.

```
out    resultDoc     clob(clob_size),
```

**Size recommendation:** The size limit of the *resultDoc* parameter depends on your system setup, but be aware that the amount specified in this parameter is the amount allocated by JDBC, regardless of the size of the document. The size should accommodate your largest XML files, but should not exceed 1.5 gigabytes.

To run the command file on UNIX or Windows, from the DB2 command line and directory where the file is located, enter:

```
db2   -tf  crtgenxc.db2
```

**Related concepts:**

- "XML Extender composition stored procedures - Overview" on page 186

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179
- "Composing XML collections by using RDB_node mapping" on page 61
- "Composing XML documents by using SQL mapping" on page 58

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxRetrieveXMLClob stored procedure

**Purpose:**

The dxxRetrieveXMLClob stored procedure enables document composition from relational data.

The requirements for using dxxRetrieveXMLClob are the same as the requirements for dxxGenXMLClob. The only difference is that the DAD is not an input parameter for dxxRetrieveXMLClob, but it is the name of an enabled XML collection.

**Syntax:**

```
dxxRetrieveXMLClob(varchar(collectionName)          collectionName  /*input*/
              integer                  overrideType,   /*input*/
              varchar(varchar_value)   override,       /*input*/
              CLOB(1M)                 resultDoc,      /*output*/
              integer                  valid,          /*output*/
              integer                  numDocs,        /*output*/
              integer                     returnCode,     /*output*/
              varchar(1024)            returnMsg)      /*output*/
```

**Parameters:**

*Table 56. dxxRetrieveXMLClob parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *collectionName* | The name of an enabled XML collection. | IN |
| *overrideType* | A flag to indicate the type of *override* parameter:<br>**NO_OVERRIDE**<br>No override.<br>**SQL_OVERRIDE**<br>Override by an SQL_stmt<br>**XML_OVERRIDE**<br>Override by a Location Path-based condition. | IN |
| *override* | Overrides the condition in the DAD file. The input value is based on the *overrideType*.<br>**NO_OVERRIDE**<br>A NULL string.<br>**SQL_OVERRIDE**<br>A valid SQL statement. Using this *overrideType* requires that SQL mapping be used in the DAD file. The input SQL statement overrides the SQL_stmt in the DAD file.<br>**XML_OVERRIDE**<br>A string that contains one or more expressions separated by the word AND. Using this *overrideType* requires that RDB_node mapping be used in the DAD file | IN |
| *resultDoc* | A CLOB that contains the composed XML document. | IN |
| *valid* | valid is set as follows:<br>• If VALIDATION=YES then valid=1 for successful validation or valid=0 for unsuccessful validation.<br>• If VALIDATION=NO then valid=NULL. | OUT |
| *numDocs* | The number of XML documents that would be generated from the input data. NOTE: currently only the first document is returned. | OUT |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

The CLOB parameter size is 1 MB.

**Related concepts:**

- "XML Extender composition stored procedures - Overview" on page 186

**Related tasks:**
- "Stored Procedures that return CLOBs" on page 180
- "Calling XML Extender stored procedures" on page 179
- "Composing XML collections by using RDB_node mapping" on page 61
- "Composing XML documents by using SQL mapping" on page 58

**Related reference:**
- Appendix C, "XML Extender limits," on page 293

# XML Extender decomposition stored procedures

## XML Extender decomposition stored procedures - Overview

The decomposition stored procedures dxxInsertXML() and dxxShredXML() are used to break down or shred incoming XML documents and to store data in new or existing database tables. The dxxInsertXML() stored procedure takes an enabled XML collection name as input. The dxxShredXML() stored procedure takes a DAD file as input; it does not require an enabled XML collection.

## dxxShredXML() stored procedure

**Purpose:**

Decomposes XML documents, based on a DAD file mapping, storing the content of the XML elements and attributes in specified DB2 database tables. In order for dxxShredXML() to work, all tables specified in the DAD file must exist, and all columns and their data types that are specified in the DAD must be consistent with the existing tables. The stored procedure requires that the columns specified in the join condition, in the DAD, correspond to primary- foreign key relationships in the existing tables. The join condition columns that are specified in the RDB_node of the root element_node must exist in the tables.

The stored procedure fragment in this section is a sample for explanation purposes.

**Syntax:**

```
dxxShredXML(CLOB(100K)    DAD,           /* input */
            CLOB(1M)      xmlobj,        /* input */
            long          returnCode,    /* output */
            varchar(1024) returnMsg)     /* output */
```

**Parameters:**

*Table 57. dxxShredXML() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *DAD* | A CLOB containing the DAD file. | IN |
| *xmlobj* | An XML document object in XMLCLOB type. | IN |
| *returnCode* | The return code from the stored procedure. | OUT |

*Table 57. dxxShredXML() parameters (continued)*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Examples:**

The following fragment is an example of a call to dxxShredXML().

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
            EXEC SQL BEGIN DECLARE SECTION;
              SQL TYPE is CLOB dad;           /* DAD*/
              SQL TYPE is CLOB_FILE  dadFile; /* DAD file*/
              SQL TYPE is CLOB xmlDoc;        /* input XML document */
              SQL TYPE is CLOB_FILE xmlFile;  /* input XMLfile */
              long        returnCode;         /* error code */
              char        returnMsg[1024];    /* error message text */
              short       dad_ind;
              short       xmlDoc_ind;
              short       returnCode_ind;
              short       returnMsg_ind;
              EXEC SQL END DECLARE SECTION;

              /* initialize host variable and indicators */
              strcpy(dadFile.name,"dxx_install
              /samples/extenders/db2xml/dad/getstart_xcollection.dad
              ");
              dadFile.name_length=strlen("dxx_install
              /samples/extenders/db2xml/dad/getstart_xcollection.dad
              ");
              dadFile.file_option=SQL_FILE_READ;
              strcpy(xmlFile.name,"dxx_install
                /samples/extenders/db2xml/xml/getstart.xml");
              xmlFile.name_length=strlen("dxx_install
                /samples/extenders/db2xml/xml/getstart.xml")");
              xmlFile.file_option=SQL_FILE_READ;
              SQL EXEC VALUES (:dadFile) INTO :dad;
              SQL EXEC VALUES (:xmlFile) INTO :xmlDoc;
              returnCode = 0;
              returnMsg[0] = '\0';
              dad_ind = 0;
              xmlDoc_ind = 0;
              returnCode_ind = -1;
              returnMsg_ind = -1;

              /* Call the stored procedure */
              EXEC SQL CALL DB2XML.dxxShredXML(:dad:dad_ind;
                             :xmlDoc:xmlDoc_ind,
                             :returnCode:returnCode_ind,
                             :returnMsg:returnMsg_ind);
```

**Related concepts:**

- "XML Extender decomposition stored procedures - Overview" on page 197

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179
- "Decomposing an XML collection by using RDB_node mapping" on page 65

- "Decomposing XML documents into DB2 database data" on page 92

**Related reference:**
- Appendix C, "XML Extender limits," on page 293

## dxxInsertXML() stored procedure

**Purpose:**

Takes two input parameters, the name of an enabled XML collection and the XML document that are to be decomposed, and returns two output parameters, a return code and a return message.

**Syntax:**

```
dxxInsertXML(char(collectionName) collectionName, /*input*/
             CLOB(1M)       xmlobj,          /* input */
             long           returnCode,      /* output */
             varchar(1024)  returnMsg)       /* output */
```

**Parameters:**

*Table 58. dxxInsertXML() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *collectionName* | The name of an enabled XML collection. | IN |
| *xmlobj* | An XML document object in CLOB type. | IN |
| *returnCode* | The return code from the stored procedure. | OUT |
| *returnMsg* | The message text that is returned in case of error. | OUT |

**Examples:**

In the following fragment example, the dxxInsertXML() call decomposes the input XML document *dxx_install*/xml/order1.xml and inserts data into the SALES_ORDER collection tables according to the mapping that is specified in the DAD file with which it was enabled with. The complete working program can be found in *dxx_install*/samples/extenders/db2xml/c/insertx.sqx.

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char        collection[64];  /* name of an XML collection */
SQL TYPE is CLOB_FILE xmlDoc;       /* input XML document */
SQL TYPE is CLOB(1M) *xmlobj=NULL;
long        returnCode;      /* error code */
char        returnMsg[1024]; /* error message text */
short       collection_ind;
short       xmlDoc_ind;
short       returnCode_ind;
short       returnMsg_ind;
EXEC SQL END DECLARE SECTION;

/* initialize host variable and indicators */
strcpy(collection,"sales_ord")
```

```
strcpy(xmldoc.name,"dxx_install/samples/extenders/db2xml/xml/getstart.xml");
xmldoc.name_length=strlen(xmldoc.name);
xmldoc.file_option=SQL_FILE_READ;
returnCode = 0;
returnMsg[0] = '\0';
collection_ind = 0;
xmlobj_ind = 0;
returnCode_ind = -1;
returnMsg_ind = -1;

/* Call the stored procedure */
EXEC SQL VALUES (:xmlfile) INTO :*xmlDoc;
EXEC SQL CALL DB2XML.dxxInsertXML(:collection:collection_ind;
:*xmlDoc:xmlobj_ind,
:returnCode:returnCode_ind,:returnMsg:returnMsg_ind);
```

**Related concepts:**

- "XML Extender decomposition stored procedures - Overview" on page 197

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179
- "Decomposing an XML collection by using RDB_node mapping" on page 65
- "Decomposing XML documents into DB2 database data" on page 92

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

# Chapter 11. XML Extender stored procedures and functions for MQSeries

## XML Extender stored procedures and functions for MQSeries - Overview

XML Extender provides two methods of storing and accessing XML data. Using the XML column method, you can store XML documents in a DB2® table while querying, updating, and retrieving the documents contents. The MQ XML user-defined functions enable you to query XML documents and then publish the results to a message queue. Additionally, you can use the XML collection method to store the untagged contents of an XML document in one or multiple tables or compose XML documents from multiple tables. Using the MQ XML stored procedures, you can retrieve an XML document from a message queue, decompose it into untagged data, and store the data in DB2 tables.You can also compose an XML document from DB2 data and send the document to an MQSeries® message queue.

MQSeries supports three messaging models to distribute XML data and documents:

**datagrams**
: Messages are sent to a single destination with no reply expected.

**publish/subscribe**
: One or more publishers send a message to a publication service which distributes the message to interested subscribers.

**request/reply**
: Messages are sent to a single destination and the sender expects to receive a response.

MQSeries can be used in numerous ways. Simple datagrams are exchanged to coordinate multiple applications, to exchange information, request services, and to provide notification of interesting events. Publish/subscribe is most often used to disseminate real-time information in a timely manner. The request/reply style is generally used as a simple form of pseudo-synchronous remote procedure call. More complex models can also be constructed by combining these basic styles.

The fundamental messaging techniques described here are used in a wide variety of ways. Because MQSeries is available across a very wide range of operating systems it provides an important mechanism to link disparate applications from similar or dissimilar environments.

To use MQXML functions and stored procedures, ensure that you have the following software installed.

- DB2 Database™ Version 9.1 or later
- DB2 MQSeries Functions Version 9.1 (Available as an optional feature of DB2 Database Version 9.1. Information is available in the DB2 Database Version 9.1 *Command Reference*.)

© Copyright IBM Corp. 1999 - 2006

# XML Extender MQSeries functions

## XML Extender MQSeries functions - Overview

DB2® XML Extender includes the following functions for use with MQSeries®:

- MQPublishXML
- MQReadXML
- MQReadAllXML
- MQReadXMLCLOB
- MQReadAllXMLCLOB
- MQReceiveXML
- MQReceiveAllXML
- MQRcvAllXMLCLOB
- MQReceiveXMLCLOB
- MQSENDXML
- MQSENDXMLFILE
- MQSendXMLFILECLOB

**Related concepts:**

- "XML Extender MQSeries stored procedures - Overview" on page 221
- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related reference:**

- "MQPublishXML function"
- "MQRcvAllXMLCLOB function" on page 214
- "MQRcvXMLCLOB function" on page 216
- "MQReadAllXML function" on page 205
- "MQReadAllXMLCLOB function" on page 208
- "MQReadXML function" on page 204
- "MQReadXMLCLOB function" on page 207
- "MQReceiveAllXML function" on page 212
- "MQReceiveXML function" on page 210
- "MQReceiveXMLCLOB function" on page 215
- "MQSendXMLFILECLOB function" on page 220
- "MQSENDXML function" on page 217
- "MQSENDXMLFILE function" on page 219

## MQPublishXML function

**Purpose:**

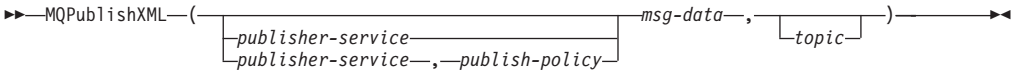
The MQPublishXML function publishes XMLVARCHAR and XMLCLOB data to MQSeries. See the following Web site for more information:

`http://www.software.ibm.com/MQSeries`

The MQPublishXML function publishes the XML data contained in *msg-data* to the MQSeries publisher specified by *publisher-service* using the quality of publish policy

*publish-policy*. The topic of the message is optionally specified by *topic*. An optional user defined message correlation identifier can be specified by *correl-id*. The function returns a 1 if successful.

**Syntax:**




**Parameters:**

*Table 59. MQPublishXML parameters*

| Parameter | Data type | Description |
|---|---|---|
| *publisher-service* | VARCHAR(48) | A string containing the logical MQSeries destination to which the message is to be sent. When specified, the *publisher-service* refers to a publisher Service Point defined in the AMT.XML repository file. If the *publisher-service* is not specified, then the DB2.DEFAULT.PUBLISHER will be used. The maximum size of publisher-service is 48 bytes. |
| *publish-policy* | VARCHAR(48) | A string containing the MQSeries AMI *publish policy* to be used in handling this message. If specified, the *publish-policy* refers to a policy which is defined in the AMT.XML repository file. The publish policy also defines a set of quality of publish options that should be applied to the messaging operation options. These options include message priority and message persistence the *service-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of service-policy is 48 bytes. For more information, see the MQSeries Application Messaging Interface. |
| *msg-data* | XMLVARCHAR or XMLCLOB | An XMLVARCHAR or XMLCLOB expression containing the data to be sent via MQSeries. |

*Table 59. MQPublishXML parameters (continued)*

| Parameter | Data type | Description |
|---|---|---|
| *topic* | VARCHAR(40) | A string containing the topic that the message is to be published under. If no topic is specified, none will be associated with the message. The maximum size of topic is 40 bytes. Multiple topics can be listed within a topic string by separating each topic by ″:″. |

**Return Codes:**
If successful, the MQPublishXML functions return a 1. A value of 0 is returned if the function is unsuccessful.
**Related concepts:**

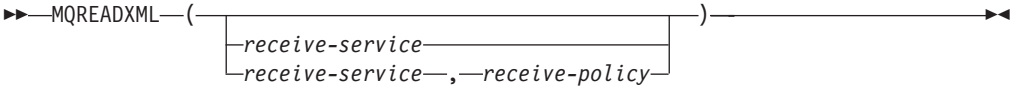
- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

## MQReadXML function

**Purpose:**

The MQREADXML function returns XMLVARCHAR data from the MQSeries location that is specified by the *receive-service*. It uses the quality of *receive-policy*. The MQREADXML function does not remove messages from the queue associated with *receive-service*

**Syntax:**




**Parameters:**

*Table 60. MQReadXML parameters*

| Parameter | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination from which the message is to be received. If the *receive-service* is specified, it refers to a service point defined in the AMT.XML repository file. If *receive-service* is not specified, then the DB2.DEFAULT.SERVICE is used. The maximum size of *receive-service* is 48 bytes |

*Table 60. MQReadXML parameters (continued)*

| Parameter | Data type | Description |
|---|---|---|
| *receive-policy* | VARCHAR(48) | A string containing the MQSeries AMI service policy used in the handling of a message. When the *receive-policy* is specified, it refers to a policy defined in the AMT.XML repository file. A receive policy defines a set of quality of receive options that are applied to the messaging operation. These options include message priority and message persistence. If the *receive-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *receive-policy* is 48 bytes. |

**Results:**
When a message in the queue has been read successfully, MQREADXML returns a db2xml.xmlvarchar. A NULL is returned if no messages are available.
**Examples:**
Example 1: This example reads the message at the head of the queue that is specified by the default service *DB2.DEFAULT.SERVICE*. It uses the default policy *DB2.DEFAULT.POLICY* to read the message.

```
values DB2XML.MQREADXML()
```

This example returns the contents of the message as an XMLVARCHAR. If no messages are available a NULL is returned.

Example 2: This example reads the message at the head of the queue specified by the service *MYSERVICE* using the default policy *DB2.DEFAULT.POLICY*.

```
values DB2XML.MQREADXML('MYSERVICE')
```

In the example, the contents of the message are returned as XMLVARCHAR. If no messages are available the a NULL is returned.

Example 3: This example reads the message at the head of the queue specified by the service *MYSERVICE* using the policy *MYPOLICY*.

```
values DB2XML.MQREADXML('MYSERVICE','MYPOLICY')
```

The contents of the message are returned as XMLVARCHAR if successful. If no messages are available a NULL is returned.

**Related concepts:**

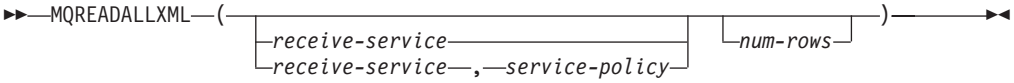
- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

## MQReadAllXML function

**Purpose:**

The MQReadAllXML function returns a table containing the messages and message metadata from the MQSeries location specified by *receive-service* using the quality of *service-policy*. Performing this operation does not remove the messages from the queue associated with *receive-service*. If *num-rows* is specified, then a maximum of num-rows messages will be returned. If num-rows is not specified then all the available messages are returned.

**Syntax:**




**Parameters:**

*Table 61. MQReadAllXML parameters*

| Parameter | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination from which the message is to be read. If specified, the *receive-service* must refer to a service point defined in the AMT.XML repository file. However, if *receive-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of *receive-service* is 48 bytes. |
| *service-policy* | VARCHAR(48) | A string containing the MQSeries AMI Service Policy used in the handling of this message. When the *service-policy* is specified, it refers to a Policy defined in the AMT.XML repository file. The maximum size of receive-service is 48 bytes. For additional information, refer to the MQSeries Application Messaging Interface manual. |
| *num-rows* | INTEGER | A positive integer containing the maximum number of messages to be returned by the function. |

**Results:**
The MQReadAllXML function returns a table containing messages and message metadata as described below.

*Table 62. Result set table*

| Column Name | Data Type | Description |
|---|---|---|
| MSG | XMLVARCHAR | The contents of the MQSeries message. The maximum length is 4K bytes. |

*Table 62. Result set table (continued)*

| Column Name | Data Type | Description |
|---|---|---|
| CORRELID | VARCHAR(24) | A correlation ID that can be used to relate to messages. |
| TOPIC | VARCHAR(40) | The topic the message was published with, if available. |
| QNAME | VARCHAR(48) | The queue name the message was received at |
| MSGID | VARCHAR(24) | The MQSeries assigned unique identifier for a message. |
| MSGFORMAT | VARCHAR(8) | The format of the message as defined by MQSeries. Typical strings have a format of MQSTR. |

**Examples:**
Example 1: All the messages from the queue that are specified by the default service DB2.DEFAULT.SERVICE are read using the default policy DB2.DEFAULT.POLICY. The messages and all the metadata are returned in a table format.

```
select * from table (DB2XML.MQREADALLXML()) t
```

Example 2: All messages that are specified by the service MYSERVICE by using the default policy DB2.DEFAULT.POLICY. Only the *msg* and *correlid* columns are returned. The message queue is in a table format, wherein you can select the fields that you want.

```
select t.MSG, t.CORRELID from table (DB2XML.MQREADALLXML('MYSERVICE')) t
```

Example 3: The queue that is specified by the default service DB2.DEFAULT.SERVICE is read using the default policy DB2.DEFAULT.POLICY.. Only messages with a *CORRELID* of '1234' are returned. Up to 10 messages are read and returned. All columns are returned.

```
select * from table (DB2XML.MQREADALLXML()) t where t.CORRELID = '1234'
```

Example 4: The messages that are specified by the default service DB2.DEFAULT.SERVICE are read using the default policy DB2.DEFAULT.POLICY . All columns are returned.

```
select * from table (DB2XML.MQREADALLXML(10)) t
```

**Related concepts:**

- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

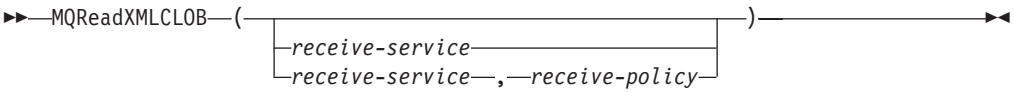
## MQReadXMLCLOB function

**Purpose:**

The MQREADXMLCLOB function returns XMLCLOB data from the MQSeries location specified by *receive-service* using the quality of service policy *receive-policy*. Performing this operation does not remove the message from the queue associated

with the *receive-service*. The message at the head of the queue will be returned. The return value is an XMLCLOB containing the messages. If no messages are available to be returned a NULL will be returned.

**Syntax:**




**Parameters:**

*Table 63. MQReadXMLCLOB parameters*

| Parameter | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination from which the message is to be received. If specified, the *receive-service* refers to a Service Point defined in the AMT.XML repository file. If *receive-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of *receive-service* is 48 bytes |
| *receive-policy* | VARCHAR(48) | A string containing the MQSeries AMI Service Policy used in the handling of this message. When the *receive-policy* is specified, it refers to a Policy defined in the AMT.XML repository file. A Service Policy defines a set of quality of service options that are applied to the messaging operation. These options include message priority and message persistence. If the *receive-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of receive-service is 48 bytes. |

**Results:**
When a message in the queue has been read successfully, MQREADXMLCLOB returns a db2xml.xmlclob. A NULL is returned if no messages are available.

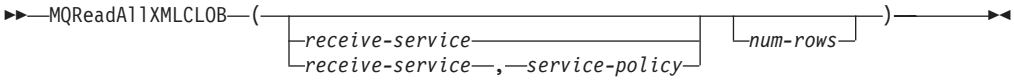
## MQReadAllXMLCLOB function

**Purpose:**

The MQReadAllXMLCLOB function returns a table containing the messages and message metadata from the MQSeries location specified by *receive-service* using the quality of service policy receive-service. Performing this operation does not remove the messages from the queue associated with receive-service. If *num-rows* is

specified, then a maximum of num-rows messages will be returned. If num-rows is not specified then all available messages will be returned.

**Syntax:**




**Parameters:**

*Table 64. MQReadAllXMLCLOB parameters*

| Parameter | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination from which the message is to be read. If specified, the *receive-service* must refer to a Service Point defined in the AMT.XML repository file. However, if *receive-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of *receive-service* is 48 bytes. |
| *service-policy* | VARCHAR(48) | A string containing the MQSeries AMI service policy used in the handling of this message. When the *service-policy* is specified, it refers to a policy defined in the AMT.XML repository file. The maximum size of *service-policy* is 48 bytes. |
| *num-rows* | INTEGER | A positive integer containing the maximum number of messages to be returned by the function. |

**Results:**

The MQReadAllXMLCLOB function returns a table containing messages and message metadata as described below.

*Table 65. MQReadAllXMLCLOB Result set table*

| Column Name | Data Type | Description |
|---|---|---|
| MSG | XMLCLOB | The contents of the MQSeries message, up to 1MB in length. |
| CORRELID | VARCHAR(24) | A correlation ID that can be used to relate messages. |
| TOPIC | VARCHAR(40) | The topic the message was published with, if available. |
| QNAME | VARCHAR(48) | The queue name the message was received at |

*Table 65. MQReadAllXMLCLOB Result set table (continued)*

| Column Name | Data Type | Description |
|---|---|---|
| MSGID | VARCHAR(24) | The MQSeries assigned unique identifier for this message |
| MSGFORMAT | VARCHAR(8) | The format of the message as defined by MQSeries. Typical strings have a format of MQSTR. |

Example 1: All the messages from the queue that are specified by the default service DB2.DEFAULT.SERVICE are read using the default policy DB2.DEFAULT.POLICY. The messages and all the metadata are returned in a table format.

```
select * from table (DB2XML.MQREADALLXMLCLOB()) t
```

Example 2: Messages from the head of the queue are specified by the service MYSERVICE by using the default policy DB2.DEFAULT.POLICY. Only the *msg* and *correlid* columns are returned.

```
select t.MSG, t.CORRELID
from table (DB2XML.MQREADALLXMLCLOB('MYSERVICE')) t
```

Example 3: The head of the queue that is specified by the default service DB2.DEFAULT.SERVICE is read using the default policy DB2.DEFAULT.POLICY . Only messages with a *CORRELID* of '1234' are returned. All columns are returned.

```
select *
from table (DB2XML.MQREADALLXMLCLOB()) t where t.CORRELID = '1234'
```

Example 4: The first 10 messages from the head of the queue that are specified by the default service DB2.DEFAULT.SERVICE are read using the default policy DB2.DEFAULT.POLICY. All columns are returned.

```
select * from table (DB2XML.MQREADALLXMLCLOB(10)) t
```

**Related concepts:**

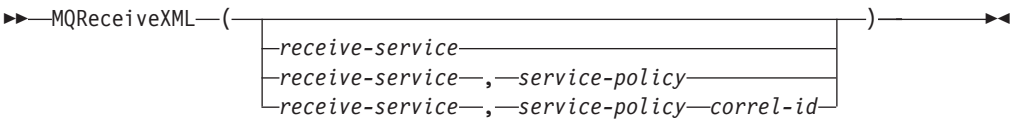
- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

## MQReceiveXML function

**Purpose:**

The MQReceiveXML removes one message associated with *receive-service* from the queue. The function returns XMLVARCHAR data from the MQSeries location specified by the *receive-service* function which uses the quality of *receive-service*.

**Syntax:**




**Parameters:**

*Table 66. MQReceiveXML parameters*

| Parameter | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination from which the message is to be received. If specified, *receive-service* refers to a service point defined in the AMT.XML repository file. If *receive-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of *receive-service* is 48 bytes. |
| *service-policy* | VARCHAR(48) | A string containing the MQSeries AMI service policy to be used in the handling of this message. If specified, the *service-policy* must refer to a policy defined in the AMT.XML repository file. If the *service-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *service-policy* is 48 bytes. |
| *correl-id* | VARCHAR(24) | A string containing an optional correlation identifier to be associated with this message. The *correl-id* is often specified in request/reply to scenarios to associate requests with replies. If it is not outlined, no correlation ID will be specified. The maximum size of *correl-id* is 24 bytes. |

**Results:**
MQReceiveXML functions return a db2xml.XMLVARCHAR if the messages are received from the queue successfully. The maximum message size is 32000 bytes. A NULL is returned if no messages are available. If the *correl-id* is specified then the first message with a matching correlation identifier will be returned. If *correl-id* is not specified then the message at the head of the queue will be returned. The message is removed from the queue.
**Examples:**
Example 1: This example receives the message that is at the head of the queue and is specified by the default service DB2.DEFAULT.SERVICE using the default policy DB2.DEFAULT.POLICY.

```
values db2xml.MQRECEIVEXML()
```

If successful this example returns the contents of a message as an XMLVARCHAR. If no messages are available a NULL is returned.

**Related concepts:**

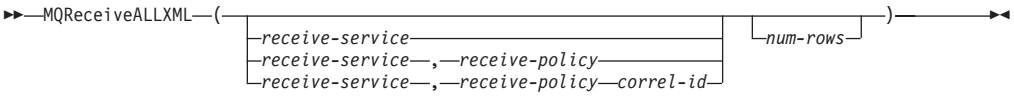
- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

## MQReceiveAllXML function

**Purpose:**

The MQReceiveAllXML removes messages associated with *receive-service* from the queue. If the correl-id is specified then only those messages with a matching correlation identifier will be returned. If *correl-id* is not specified then the message at the head of the queue will be returned. If *num-rows* are specified, then a maximum of *num-rows* messages will be returned. If it is not specified then all available messages will be returned.

**Syntax:**




**Parameters:**

*Table 67. MQReceiveAllXML parameters*

| Parameter | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination to which the message is to be sent. When specified, the *send-service* refers to a Service Point defined in the ATM.XML repository file. If *send-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of *send-service* is 48 bytes. |
| *receive-policy* | VARCHAR(48) | A string containing the MQSeries AMI service policy to be used in the handling of this message. If specified, the *receive-policy* must refer to a policy defined in the AMT.XML repository file. If the *receive-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *receive-policy* is 48 bytes. |
| *correl-id* | VARCHAR(24) | A string containing an optional correlation identifier to be associated with this message. The *correl-id* is often specified in request/reply scenarios to associate requests with replies. If it is not outlined no correlation id will be specified. The maximum size of *correl-id* is 24 bytes. |

*Table 67. MQReceiveAllXML parameters (continued)*

| Parameter | Data type | Description |
|---|---|---|
| *num-rows* | INTEGER | A positive integer that contains the maximum number of messages returned by the function. |

**Results:**
When a table of messages is successfully received from the queue, MQRECEIVEXML returns a db2xml.xmlvarchar. A NULL is returned when no messages are available. The messages are returned as a table of messages with meta-data.

| Column Name | Data type | Description |
|---|---|---|
| MSG | XMLVARCHAR | The contents of the MQSeries message. |
| CORRELID | VARCHAR(24) | A correlation ID that can be used to relate messages. |
| TOPIC | VARCHAR(40) | The topic the message was published with, if available. |
| QNAME | VARCHAR(48) | The queue name the message was received at. |
| MSGID | CHAR(24) | The MQSeries assigned unique identifier for this message |
| MSGFORMAT | VARCHAR(8) | The format of the message as defined by MQSeries. Typical strings have a format of MQSTR. |

**Examples:**
Example 1: All messages received from the queue are specified by the default service (DB2.DEFAULT.SERVICE) using the default policy (DB2.DEFAULT.POLICY). The messages and all the metadata are returned as a table.

```
select * from table (MQRECEIVEALLXML()) t
```

Example 2: All the messages are received from the head of the queue and are specified by the service MYSERVICE using the default policy (DB2.DEFAULT.POLICY). Only the MSG and CORRELID columns are returned. The messages are in table format, wherein you can select the fields that you want.

```
select t.MSG, t.CORRELID from table (MQRECEIVEALLXML('MYSERVICE')) t
```

Example 3: All the messages received from the head of the queue are specified by the service MYSERVICE using the policy MYPOLICY that match the id '1234'. Only the MSG and CORRELID columns are returned.

```
select t.MSG, t.CORRELID from table
(MQRECEIVEALLXML('MYSERVICE','MYPOLICY','1234')) t
```

Example 4: The first 10 messages are received from the head of the queue and specified by the default service (DB2.DEFAULT.SERVICE) using the default policy (DB2.DEFAULT.POLICY) . All columns are returned.

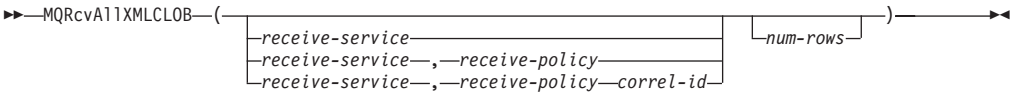
```
select * from table (MQRECEIVEALLXML(10)) t
```

## MQRcvAllXMLCLOB function

**Purpose:**

The MQRcvAllXMLCLOB removes the messages from the queue associated with *receive-service*. If the correl-id is specified then only those messages with a matching correlation identifier will be returned. If *correl-id* is not specified then all messages will be returned. If *num-rows* is specified, then a maximum of *num-rows* messages will be returned as XMLCLOB. If it is not specified then all available messages will be returned.

**Syntax:**




**Parameters:**

*Table 68. MQRcvAllXMLCLOB parameters*

| Parameter | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination from which the message is to be received. If specified, the *receive-service* refers to a Service Point defined in the AMT.XML repository file. If *receive-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of receive-service is 48 bytes. |
| *receive-policy* | VARCHAR(48) | A string containing the MQSeries AMI service policy to be used in the handling of this message. If specified, the *receive-policy* must refer to a policy defined in the AMT.XML repository file. If the *receive-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *receive-policy* is 48 bytes. |
| *correl-id* | VARCHAR(24) | A string containing an optional correlation identifier to be associated with this message. The *correl-id* is often specified in request/reply scenarios to associate requests with replies. If it is not outlined no correlation id will be specified. The maximum size of *correl-id* is 24 bytes. |

*Table 68. MQRcvAllXMLCLOB parameters (continued)*

| Parameter | Data type | Description |
|---|---|---|
| *num-rows* | INTEGER | A positive integer that contains the maximum number of messages returned by the function. |

**Results:**
When a message is successfully received from the queue, MQRcvAllXMLCLOB returns an XMLCLOB. A NULL is returned when no messages are available. The messages are returned in a table as described below

*Table 69. MQRcvAllXML result set table*

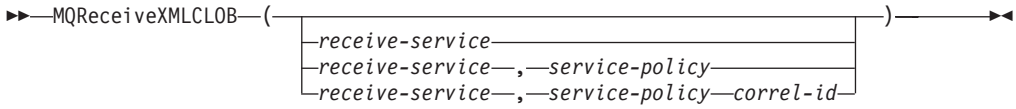
| Column Name | Data type | Description |
|---|---|---|
| MSG | XMLCLOB | The contents of the MQSeries message. |
| CORRELID | VARCHAR(24) | A correlation ID that can be used to relate messages. |
| TOPIC | VARCHAR(40) | If the topic the message was published with, if available. |
| QNAME | VARCHAR(48) | The queue name the message was received at. |
| MSGID | CHAR(24) | The MQSeries assigned unique identifier for this message |
| MSGFORMAT | VARCHAR(8) | The format of the message as defined by MQSeries. Typical strings have a format of MQSTR. |

## MQReceiveXMLCLOB function

**Purpose:**

The MQReceiveXMLCLOB removes messages associated with *receive-service* from the queue. The function returns XMLVARCHAR data from the MQSeries location specified by the *service-policy* function which uses the quality of *receive-service*.

**Syntax:**




**Parameters:**

*Table 70. MQReceiveXMLCLOB parameters*

| Parameter | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination from which the message is to be received. When the *receive-service* is specified, it refers to a Service Point defined in the AMT.XML repository file. However, if *receive-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of *receive-service* is 48 bytes. |
| *service-policy* | VARCHAR(48) | A string containing the MQSeries AMI Service Policy to be used in handling of this message. If specified, the receive-service must refer to a Policy defined in the AMT.XML repository file. If *service-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *service-policy* is 48 bytes. |
| *correl-id* | VARCHAR(24) | A string containing an optional correlation identifier to be associated with this message. The *correl-id* is often specified in request/reply to scenarios to associate requests with replies. If it is not outlined, no correlation ID will be specified. The maximum size of *correl-id* is 24 bytes. |

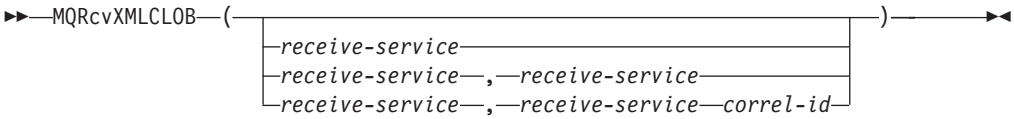
**Results:**
MQReceiveXMLCLOB functions return a db2xml.XMLCLOB if messages are received from the queue successfully. A NULL is returned if no messages are available. If the *correl-id* is specified then the first message with a matching correlation identifier will be returned. However, if the *correl-id* is not specified then the message at the head of the queue will be returned.

## MQRcvXMLCLOB function

**Purpose:**

The MQRcvXMLCLOB removes messages associated with *receive-service* from the queue. The function returns XMLVARCHAR data from the MQSeries location specified by the *receive-service* function which uses the quality of *receive-service*.

**Syntax:**




**Parameters:**

*Table 71. MQRcvXMLCLOB parameters*

| Data type | Data type | Description |
|---|---|---|
| *receive-service* | VARCHAR(48) | A string containing the logical MQSeries destination from which the message is to be received. When the *receive-service* is specified, it refers to a Service Point defined in the AMT.XML repository file. However, if *receive-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of *receive-service* is 48 bytes. |
| *receive-service* | VARCHAR(48) | A string containing the MQSeries AMI Service Policy to be used in handling of this message. If specified, the receive-service must refer to a Policy defined in the AMT.XML repository file. If *receive-service* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *receive-service* is 48 bytes. |
| *correl-id* | VARCHAR(24) | A string containing an optional correlation identifier to be associated with this message. The *correl-id* is often specified in request/reply to scenarios to associate requests with replies. If it is not outlined, no correlation ID will be specified. The maximum size of *correl-id* is 24 bytes. |

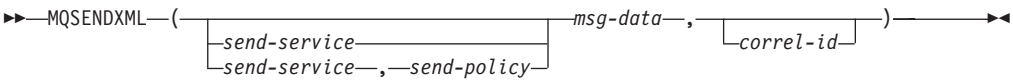
**Results:**
MQRcvMLCLOB functions return a db2xml.XMLCLOB if messages are received from the queue successfully. The maximum message size is 1M. A NULL is returned if no messages are available. If the *correl-id* is specified then the first message with a matching correlation identifier will be returned. However, if the *correl-id* is not specified then the message at the head of the queue will be returned.

## MQSENDXML function

**Purpose:**

The MQSENDXML function sends the data contained in *msg-data* to the MQSeries location specified by send-service using the send-policy. An optional user-defined message correlation identifier can also be specified by correl-id. The function returns a 1 if successful.

**Syntax:**




**Parameters:**

*Table 72. MQSendXML parameters*

| Parameter | Data type | Description |
|---|---|---|
| *msg-data* | XMLVARCHAR or XMLCLOB | An expression containing the data to be sent via MQSeries. |
| *send-service* | VARCHAR(48) | A string containing the logical MQSeries destination to which the message is to be sent. When the *send-service* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *send-service* is not specified. The maximum size of *send-service* is 48 bytes. |
| *send-policy* | VARCHAR(48) | A string containing the MQSeries AMI Service Policy used to handle the message. When specified, the *send-policy* refers to a policy defined in the AMT.XML repository file. If the *send-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of send-policy is 48 bytes. |
| *correl-id* | VARCHAR(24) | A string containing an optional correlation identifier associated with the message. The correl-id is often specified in request/reply scenarios to associate requests with replies. If it is not specified, no correlation id will be shown. The maximum size of correl-id is 24 bytes. |

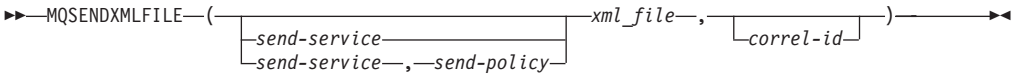
**Results:**
A successful message results in a value of 1. A message containing *msg-data* will be sent to the location specified by send-service using the policy defined by *send-policy*.

# MQSENDXMLFILE function

**Purpose:**

The MQSENDXMLFILE function sends the data contained in *xml_file* to the MQSeries location specified by send-service using the quality of service policy. An optional user defined message correlation identifier can be specified by correl-id. The function returns a '1' if successful.

**Syntax:**




**Parameters:**

*Table 73. MQSENDXMLFILE parameter*

| Parameter | Data type | Description |
|---|---|---|
| *xml_file* | XMLCLOB | An XML file name with a maximum size of 80 bytes. The file contains the data to be sent via MQSeries. |
| *send-service* | VARCHAR(48) | A string containing the logical MQSeries destination to which the message is to be sent. When specified, the *send-service* refers to a Service Point defined in the AMT.XML repository file. If *send-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of *send-service* is 48 bytes. |
| *send-policy* | VARCHAR(48) | A string containing the MQSeries AMI service to be used in handling of this message. If specified, the *send-policy* refers to a Policy defined in the AMT.XML repository file. If *send-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *send-policy* is 48 bytes |
| *correl-id* | VARCHAR(24) | A string containing an optional correlation identifier to be associated with this message. The *correl-id* is often specified in request/reply scenarios to associate requests with replies. If not specified, no correlation id will be listed. The maximum size of *correl-id* is 24 bytes. |

**Results:**

If the function is successful, it results in a '1'. The side effect of successfully executing this function is that a message containing *msg-data* will be sent to the location specified by *send-service* using the policy defined by *send-policy*.

**Examples:**

Example 1: XML documents contained in file "c:\xml\test1.xml" are sent to the default service (DB2.DEFAULT.SERVICE) using the default policy (DB2.DEFAULT.POLICY) with no correlation identifier.

```
Values MQSENDXMLFILE('c:\xml\test1.xml');
```

This example returns the value '1' if successful

Example 2: XML documents contained in file `c:\xml\test2.xml` are sent to the service MYSERVICE using policy MYPOLICY with no correlation identifier.

```
Values MQSENDXMLFILE('MYSERVICE', 'MYPOLICY', 'c:\xml\test2.xml');
```

This example returns the value '1' if successful

Example 3: XML documents contained in file "c:\xml\test3.xml"are sent to the service MYSERVICE using policy MYPOLICY with correlation identifier "Test3".

```
Values MQSENDXML('MYSERVICE','MYPOLICY', 'c:\xml\test3.xml', 'Test3');
```

This example returns the value '1' if successful.

Example 4: XML documents contained in file "c:\xml\test4.xml" are sent to the service MYSERVICE using the default policy (DB2.DEFAULT.POLICY) and no correlation identifier.

```
Values MQSENDXMLFILE('MYSERVICE', 'c:\xml\test4.xml');
```

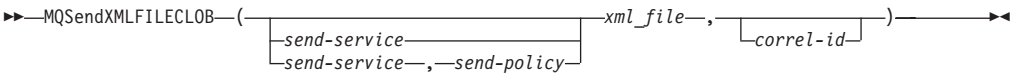
This example returns the value '1' if successful.

## MQSendXMLFILECLOB function

**Purpose:**

The MQSendXMLFILECLOB function sends the data contained in *xml_file* to the MQSeries location specified by send-service using the quality of *send-policy*. The data type that is sent is XMLCLOB. An optional user defined message correlation identifier can be specified by correl-id. The function returns a 1 if successful.

**Syntax:**




**Parameters:**

*Table 74. MQSENDXMLFILE parameter*

| Parameter | Data type | Description |
|---|---|---|
| *xml_file* | XMLCLOB | An XML file name with a maximum size of 80 bytes. The file contains the data to be sent via MQSeries. |

Table 74. MQSENDXMLFILE parameter (continued)

| Parameter | Data type | Description |
|---|---|---|
| *send-service* | VARCHAR(48) | A string containing the logical MQSeries destination to which the message is to be sent. When specified, the send-service refers to a Service Point defined in the AMT.XML repository file. If *send-service* is not specified, then the DB2.DEFAULT.SERVICE will be used. The maximum size of send-service is 48 bytes |
| *send-policy* | VARCHAR(48) | A string containing the MQSeries AMI service to be used in handling of this message. If specified, the *send-policy* refers to a Policy defined in the AMT.XML repository file. If *send-policy* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *send-policy* is 48 bytes |
| *correl-id* | VARCHAR(24) | A string containing an optional correlation identifier to be associated with this message. The *correl-id* is often specified in request/reply scenarios to associate requests with replies. If not specified, no correlation id will be listed. The maximum size of *correl-id* is 24 bytes. |

**Results:**
If the function is successful, it results in a '1'. The side effect of successfully executing this function is that a message containing *msg-data* will be sent to the location specified by *send-service* using the policy defined by *send-policy*.

# XML Extender MQSeries stored procedures

## XML Extender MQSeries stored procedures - Overview

### Composition stored procedures

Use the composition stored procedures, dxxmqGen(), dxxmqGenCLOB(), dxxmqRetrieve(), and dxxmqRetrieveCLOB() to generate XML documents using data in existing database tables, and to send the generated XML documents to a message queue. The dxxmqGen() and dxxmqGenCLOB() stored procedures use a DAD file as input. They do not require enabled XML collections. The dxxmqRetrieve and dxxmqRetrieveCLOB stored procedures use collection names as input.

### Decomposition stored procedures

The decomposition stored procedures dxxmqInsert(), dxxmqInsertAll(), dxxInsertCLOB(), dxxmqShred(), dxxmqShredCLOB, and dxxmqShredAll()

are used to break down or shred incoming XML documents from a message queue, and to store the data in new or existing database tables.

The dxxmqInsert(), dxxmqInsertAll(), dxxmqInsertAllCLOB(), and dxxInsertCLOB() stored procedures use an enabled XML collection name as input.

The dxxmqShred(), dxxmqShredAll(), dxxmqShredCLOB, and dxxmqShredAllCLOB stored procedures use a DAD file as input. They do not require an enabled XML collection.

The table below summarizes the different stored procedures and explains their functions.

*Table 75. The MQSeries® XML stored procedures*

| Function | Purpose |
|---|---|
| dxxmqGen | Invoke the dxxmqGen stored procedure to compose XML documents, using a DAD file as a input parameter. The resulting document type is XMLVARCHAR(32000). |
| dxxmqGenCLOB | Constructs an XML document from data that is stored in the XML collection tables specified in the DAD file, and sends the XML document to an MQ message queue. The resulting document type is XMLCLOB(1M). |
| dxxmqRetrieve | Invoke the dxxmqRetrieve stored procedure to compose XML documents, using a collection name as a input parameter. The resulting document type is XMLVARCHAR(32000). |
| dxxmqRetrieveCLOB | Invoke the dxxmqRetrieve stored procedure to compose XML documents, using a collection name as a input parameter. The resulting document type is XMLCLOB(1M). |
| dxxmqShred | Invoke the dxxmqShred stored procedure to decompose an XML document using a DAD file as an input parameter. The resulting document type is XMLVARCHAR(32000). |
| dxxmqShredAll | Invoke the dxxmqShredAll stored procedure to decompose multiple XML documents using a DAD file as an input parameter. The resulting document type is XMLVARCHAR(32000). |
| dxxmqShredCLOB | Decomposes an incoming XML document from a message queue, based on a DAD file mapping, and stores the content of the XML elements and attributes in specified DB2® tables. The resulting document type is XMLCLOB(1M). |
| dxxmqShredAllCLOB | Decomposes an incoming XML document from a message queue, based on a DAD file mapping, and stores the content of the XML elements and attributes in specified DB2 tables. The resulting document type is XMLCLOB(1M). |

*Table 75. The MQSeries® XML stored procedures (continued)*

| Function | Purpose |
|---|---|
| dxxmqInsert | Invoke the dxxmqInsert stored procedure to decompose an XML document using a collection name as an input parameter. The resulting document type is XMLVARCHAR(32000). |
| dxxmqInsertAll | Invoke the dxxmqInsertAll stored procedure to decompose multiple XML documents using a collection name as an input parameter. The resulting document type is XMLVARCHAR(32000). |
| dxxmqInsertCLOB | Breaks down or shreds an incoming XML document from a message queue, and stores the data in new or existing database tables. The resulting document type is XMLCLOB(1M). |
| dxxmqInsertAllCLOB | Breaks down or shreds all incoming XML documents from a message queue, and stores the data in new or existing database tables. The dxxmqInsertAllCLOB stored procedure uses a collection name, rather than a DAD file name, to determine how to store the data. The resulting document type is XMLCLOB(1M). |

**Related reference:**

- "dxxmqGen() stored procedure"
- "dxxmqGenCLOB stored procedure" on page 226
- "dxxmqInsert stored procedure" on page 237
- "dxxmqInsertAll stored procedure" on page 241
- "dxxmqInsertAllCLOB stored procedure" on page 242
- "dxxmqInsertCLOB stored procedure" on page 239
- "dxxmqRetrieve stored procedure" on page 228
- "dxxmqRetrieveCLOB stored procedure" on page 230
- "dxxmqShred stored procedure" on page 232
- "dxxmqShredAll stored procedure" on page 234
- "dxxmqShredAllCLOB stored procedure" on page 236
- "dxxmqShredCLOB stored procedure" on page 235

## dxxmqGen() stored procedure

**Purpose:**

Constructs an XML document from data that is stored in the XML collection tables specified in the DAD file, and sends the XML document to a MQ message queue. The stored procedure returns a string to indicate the status of the stored procedure.

To support dynamic query, dxxmqGen() takes an input parameter, *override*. Based on the input *overrideType*, the application can override the SQL_stmt for SQL mapping or the conditions in RDB_node for RDB_node mapping in the DAD file. The input parameter *overrideType* is used to clarify the type of the *override*.

**Syntax:**

```
dxxmqGen(varchar(48)    serviceName,     /*input*/
         varchar(48)    policyName,      /*input*/
         varchar(80)    dadFileName,     /*input*/
         integer        overrideType,    /*input*/
         varchar(varchar_value)  override,        /*input*/
         integer        maxRows,         /*input*/
         integer        numRows,         /*output*/
         char(20)       status)          /*output*/
```

**Parameters:**

*Table 76. dxxmqGen() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a service point defined in the AMT.XML repository file. The DB2.DEFAULT.SERIVCE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *dadFileName* | The name of the DAD file. | IN |
| *overrideType* | A flag to indicate the type of the following *override* parameter:<br>• **NO_OVERRIDE**: No override.<br>• **SQL_OVERRIDE**: Override by an SQL_stmt.<br>• **XML_OVERRIDE**: Override by a Location Path-based condition. | IN |
| *override* | Overrides the condition in the DAD file. The input value is based on the *overrideType*.<br>• **NO_OVERRIDE**: A NULL string.<br>• **SQL_OVERRIDE**: A valid SQL statement. Using this *overrideType* requires that SQL mapping is used in the DAD file. The input SQL statement overrides the SQL_stmt in the DAD file.<br>• **XML_OVERRIDE**: A string that contains one or more expressions in double quotation marks separated by "AND". Using this *overrideType* requires that RDB_node mapping is used in the DAD file. | IN |
| *maxRows* | The maximum number of messages generated in the message queue. | IN |

*Table 76. dxxmqGen() parameters (continued)*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *numRows* | The actual number of generated rows in the message queue. | OUT |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Examples:**

The following example fragment generates an XML document and sends it to the queue. It assumes that a MQ/AMI service, *myService*, and a policy, *myPolicy*, have been defined in the repository file. This file stores repository definitions in XML format.

```
#include "dxx.h"
#include "dxxrc.h"
              EXEC SQL INCLUDE SQLCA;
              EXEC SQL BEGIN DECLARE SECTION;
              char           serviceName[48];  /* name of the MQ/AMI service*/
              char           policyName[48];   /* name of the MQ/AMI policy*/
              char           dadFileName[80];  /* name of the DAD file */
              char           override[2];      /* override, will set to NULL*/
              short          overrideType;     /* defined in dxx.h */
              short          max_row;          /* maximum number of rows */
              short          num_row;          /* actual number of rows */
              char           status[20]        /* status code or message */
              short          ovtype_ind;
              short          ov_ind;
              short          maxrow_ind;
              short          numrow_ind;
              short          dadFileName_ind;
              short          serviceName_ind;
              short          policyName_ind;
              short          status_ind;

              EXEC SQL END DECLARE SECTION;
              strcpy(dadFileName,"c:\dxx\dad\litem3.dad");
              strcpy(serviceName,"myService");
              strcpy(policyName,"myPolicy");
              override[0] = '\0';
              overrideType = NO_OVERRIDE;
              max_row = 500;
              num_row = 0;
              status[0] = '\0';
              dadFileName_ind = 0;
              serviceName_ind = 0;
              policyName_ind = 0;
              maxrow_ind = 0;
              numrow_ind = -1;
              ovtype_ind=0;
              ov_ind=-1;
              status_ind = -1;

             /* Call the stored procedure */
             EXEC SQL CALL db2xml.dxxmqGen(:serviceName:serviceName_ind,
                    :policyName:policyName_ind,
                    :dadFileName:dadFileName_ind,
                    :overrideType:ovtype_ind,
```

```
:override:ov_ind,
:max_row:maxrow_ind,
:num_row:numrow_ind,
:status:status_ind);
```

**Related concepts:**

- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related tasks:**

- "Calling XML Extender stored procedures" on page 179

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqGenCLOB stored procedure

**Purpose:**

Constructs an XML document from data that is stored in the XML collection tables specified in the DAD file, and sends the XML document to a MQ message queue. The document type is XMLCLOB. The stored procedure returns a string to indicate the status of the stored procedure. This stored procedure is not supported for the Enterprise Server Edition (ESE).

To support dynamic query, dxxmqGenCLOB takes an input parameter, *override*. Based on the input *overrideType*, the application can override the SQL_stmt for SQL mapping or the conditions in RDB_node for RDB_node mapping in the DAD file. The input parameter *overrideType* is used to clarify the type of the *override*.

**Syntax:**

```
dxxmqGenCLOB(varchar(48)    serviceName,     /*input*/
             varchar(48)    policyName,      /*input*/
             varchar(80)    dadFileName,     /*input*/
             integer        overrideType,    /*input*/
             varchar(varchar_value)  override,        /*input*/
             integer        maxRows,         /*input*/
             integer        numRows,         /*output*/
              char(20)       status)         /*output*/
```

**Parameters:**

*Table 77. dxxmqGenCLOB parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a service entry in the table DB2MQ.MQService. The DB2.DEFAULT.SERIVCE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |

*Table 77. dxxmqGenCLOB parameters (continued)*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *policyName* | A string containing the MQSeries Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the table DB2MQ.MQPolicy. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *dadFileName* | The name of the DAD file. | IN |
| *overrideType* | A flag to indicate the type of the following *override* parameter:<br>• **NO_OVERRIDE**: No override.<br>• **SQL_OVERRIDE**: Override by an SQL_stmt.<br>• **XML_OVERRIDE**: Override by a Location Path-based condition. | IN |
| *override* | Overrides the condition in the DAD file. The input value is based on the *overrideType*.<br>• **NO_OVERRIDE**: A NULL string.<br>• **SQL_OVERRIDE**: A valid SQL statement. Using this *overrideType* requires that SQL mapping is used in the DAD file. The input SQL statement overrides the SQL_stmt in the DAD file.<br>• **XML_OVERRIDE**: A string that contains one or more expressions in double quotation marks separated by "AND". Using this *overrideType* requires that RDB_node mapping is used in the DAD file. | IN |
| *maxRows* | The maximum number of messages generated in the message queue. | IN |
| *numRows* | The actual number of generated rows in the message queue. | OUT |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Related concepts:**

• "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related reference:**

• Appendix C, "XML Extender limits," on page 293

## dxxmqRetrieve stored procedure

**Purpose:**

The stored procedure dxxmqRetrieve() serves as a means for retrieving decomposed XML documents. As input, dxxmqRetrieve() takes a buffer containing the enabled XML collection name, the MQ service and policy names. It sends the composed XML document to a MQ Queue; it returns the number of rows sent to the queue and a status message. The dxxmqRetrieve stored procedure enables the same DAD file to be used for both composition and decomposition.

To support dynamic query, dxxmqRetrieve() takes an input parameter, *override*. Based on the input *overrideType*, the application can override the SQL_stmt for SQL mapping or the conditions in RDB_node for RDB_node mapping in the DAD file. The input parameter *overrideType* is used to clarify the type of the *override*.

The requirements of the DAD file for dxxmqRetrieve() are the same as the requirements for dxxmqGen(). The only difference is that the DAD is not an input parameter for dxxmqRetrieve(); the required parameter is instead the name of an enabled XML collection.

**Syntax:**

```
dxxmqRetrieve(varchar(48)    serviceName,        /*input*/
              varchar(48)    policyName,         /*input*/
              varchar(80)    collectionName,     /*input*/
              integer        overrideType,       /*input*/
              varchar(varchar_value)  override,           /*input*/
              integer        maxrows,            /*input*/
              integer        numrows,            /*output*/
              char(20)       status)             /*output*/
```

**Parameters:**

*Table 78. dxxmqRetrieve() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *collectionName* | The name of an enabled collection. | IN |

Table 78. dxxmqRetrieve() parameters (continued)

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *overrideType* | A flag to indicate the type of the following *override* parameter:<br>• **NO_OVERRIDE**: No override.<br>• **SQL_OVERRIDE**: Override by an SQL_stmt.<br>• **XML_OVERRIDE**: Override by a Location Path-based condition. | IN |
| *override* | Overrides the condition in the DAD file. The input value is based on the *overrideType*.<br>• **NO_OVERRIDE**: A NULL string.<br>• **SQL_OVERRIDE**: A valid SQL statement. Using this *overrideType* requires that SQL mapping is used in the DAD file. The input SQL statement overrides the SQL_stmt in the DAD file.<br>• **XML_OVERRIDE**: A string that contains one or more expressions in double quotation marks separated by "AND". The maximum length is 1024 bytes. The *overrideType* string requires that RDB_node mapping is used in the DAD file. | IN |
| *maxRows* | The maximum number of rows in the result table. | IN |
| *numRows* | The actual number generated rows in the result table. | OUT |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Examples:**

The following fragment is an example of a call to dxxmqRetrieve().

```
#include "dxx.h"
#include "dxxrc.h"
               EXEC SQL INCLUDE SQLCA;
               EXEC SQL BEGIN DECLARE SECTION;
               char      serviceName[48];   /* name of the MQ/AMI service*/
               char      policyName[48];    /* name of the MQ/AMI policy*/
               char      collection[32];    /* name of the XML collection */
               char      override[2];       /* override, will set to NULL*/
               short     overrideType;      /* defined in dxx.h */
               short     max_row;           /* maximum number of rows */
               short     num_row;           /* actual number of rows */
               char      status[20];        /* status code or message */
               short     ovtype_ind;
               short     ov_ind;
               short     maxrow_ind;
               short     numrow_ind;
               short     collection_ind;
```

```
short     serviceName_ind;
short     policyName_ind;
short     status_ind;

EXEC SQL END DECLARE SECTION;

/* initialize host variable and indicators */
strcpy(collection,"sales_ord");
strcpy(serviceName,"myService");
strcpy(policyName,"myPolicy");
override[0] = '\0';
overrideType = NO_OVERRIDE;
max_row = 500;
num_row = 0;
status[0] = '\0';
serviceName_ind = 0;
policyName_ind = 0;
collection_ind = 0;
maxrow_ind = 0;
numrow_ind = -1;
ovtype_ind=0;
ov_ind=-1;
status_ind = -1;

/* Call the stored procedure */
EXEC SQL CALL db2xml.dxxmqRetrieve(:serviceName:serviceName_ind,
       :policyName:policyName_ind,
       :collection:collection_ind,
       :overrideType:ovtype_ind,
       :override:ov_ind,
       :max_row:maxrow_ind,
       :num_row:numrow_ind,
       :status:status_ind);
```

**Related concepts:**

- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqRetrieveCLOB stored procedure

**Purpose:**

The stored procedure dxxmqRetrieveCLOB serves as a means for retrieving decomposed XML documents. As input, dxxmqRetrieveCLOB takes a buffer containing the enabled XML collection name, the MQ/AMI service and policy names. It sends the composed XML document to a MQ Queue; and it returns the number of rows sent to the queue and a status message.The dxxmqRetrieveCLOB stored procedure enables the same DAD file to be used for both composition and decomposition. This stored procedure is not supported for Enterprise Server Edition (ESE).

To support dynamic query, dxxmqRetrieveCLOB takes an input parameter, *override*. Based on the input *overrideType*, the application can override the SQL_stmt for SQL mapping or the conditions in RDB_node for RDB_node mapping in the DAD file. The input parameter *overrideType* is used to clarify the type of the *override*.

The requirements of the DAD file for dxxmqRetrieveCLOB are the same as the requirements for dxxmqGenCLOB. The only difference is that the DAD is not an input parameter for dxxmqRetrieveCLOB; the required parameter is instead the name of an enabled XML collection.

**Syntax:**

```
dxxmqRetrieveCLOB(varchar(48)    serviceName,        /*input*/
                  varchar(48)    policyName,         /*input*/
                  varchar(80)    collectionName,     /*input*/
                  integer        overrideType,       /*input*/
                  varchar(varchar_value)  override,          /*input*/
                  integer        maxrows,            /*input*/
                  integer        numrows,            /*output*/
                  char(20)       status)             /*output*/
```

**Parameters:**

*Table 79. dxxmqRetrieveCLOB parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *collectionName* | The name of an enabled collection. | IN |
| *overrideType* | A flag to indicate the type of the following *override* parameter:<br>• **NO_OVERRIDE**: No override.<br>• **SQL_OVERRIDE**: Override by an SQL_stmt.<br>• **XML_OVERRIDE**: Override by a Location Path-based condition. | IN |

*Table 79. dxxmqRetrieveCLOB parameters (continued)*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *override* | Overrides the condition in the DAD file. The input value is based on the *overrideType*.<br>• **NO_OVERRIDE**: A NULL string.<br>• **SQL_OVERRIDE**: A valid SQL statement. Using this *overrideType* requires that SQL mapping is used in the DAD file. The input SQL statement overrides the SQL_stmt in the DAD file.<br>• **XML_OVERRIDE**: A string that contains one or more expressions in double quotation marks separated by "AND". The maximum size is 1024 bytes. The *overrideType* string requires that RDB_node mapping is used in the DAD file. | IN |
| *maxRows* | The maximum number of rows in the result table. | IN |
| *numRows* | The actual number generated rows in the result table. | OUT |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Related concepts:**

- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqShred stored procedure

**Purpose:**

Decomposes an incoming XML document from a message queue, based on a DAD file mapping, and stores the content of the XML elements and attributes in specified DB2 tables.

In order for dxxmqShred() to work, all tables specified in the DAD file must exist, and all columns and their data types that are specified in the DAD must be consistent with the existing tables. The stored procedure requires that the columns specified in the join condition, in the DAD, correspond to primary- foreign key relationships in the existing tables. The join condition columns that are specified in the RDB_node of the root element_node must exist in the tables.

**Syntax:**

```
dxxmqShred(varchar(48)   serviceName,     /* input */
           varchar(48)   policyName,     /* input */
           varchar(80)   dadFileName,    /* input */
           varchar(10)   status)         /* output */
```

**Parameters:**

*Table 80. dxxmqShred() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *dadFileName* | The name of the DAD file. The maximum size is 80 bytes. | IN |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Examples:**

The following fragment is an example of a call to dxxmqShred().

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
        EXEC SQL BEGIN DECLARE SECTION;
         char        serviceName[48];    /* name of the MQ/AMI service */
         char        policyName[48];     /* name of the MQ/AMI policy */
         char        dadFileName[80];    /* name of the DAD file */
         char        status[20];         /* status code or message */
         short       serviceName_ind;
         short       policyName_ind;
         short       dadFileName_ind;
         short       status_ind;
         EXEC SQL END DECLARE SECTION;

              /* initialize host variable and indicators */
         strcpy(dadFileName,"e:/dxx/samples/dad/getstart_xcollection.dad");
         strcpy(serviceName, "myService");
         strcpy(policyName, "myPolicy");
         status[0]='\0';
         serviceName_ind=0;
```

```
policyName_ind=0;
dadFileName_ind=0;
status_ind=-1;

     /* Call the stored procedure */
EXEC SQL CALL db2xml.dxxmqShred(:serviceName:serviceName_ind,
                  :policyName:policyName_ind,
                  :dadFileName:dadFileName_ind,
                  :status:status_ind);
```

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqShredAll stored procedure

**Purpose:**

Decomposes all incoming XML documents from a message queue, based on a DAD file mapping. The contents of the XML elements and attributes are stored in specified DB2 tables.

In order for dxxmqShredAll() to work, all tables specified in the DAD file must exist, and all columns and their data types that are specified in the DAD must be consistent with the existing tables. The stored procedure requires that the columns specified in the join condition, in the DAD, correspond to primary-foreign key relationships in the existing tables. The join condition columns that are specified in the RDB_node of the root element_node must exist in the tables.

**Syntax:**

```
dxxmqShredAll(varchar(48)   serviceName,     /* input */
              varchar(48)   policyName,     /* input */
              varchar(80)   dadFileName,    /* input */
              varchar(20)   status)         /* output */
```

**Parameters:**

*Table 81. dxxmqShredAll() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *dadFileName* | The name of the DAD file. The maximum size is 80 bytes. | IN |

*Table 81. dxxmqShredAll() parameters (continued)*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Examples:**

The following fragment is an example of a call to dxxmqShredAll().

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
      EXEC SQL BEGIN DECLARE SECTION;
        char        serviceName[48];   /* name of the MQ/AMI service */
        char        policyName[48];    /* name of the MQ/AMI policy */
        char        dadFileName[80];   /* name of the DAD file */
        char        status[20];        /* status code or message */
        short       serviceName_ind;
        short       policyName_ind;
        short       dadFileName_ind;
        short       status_ind;
        EXEC SQL END DECLARE SECTION;

              /* initialize host variable and indicators */
        strcpy(dadFileName,"e:/dxx/samples/dad/getstart_xcollection.dad");
        strcpy(serviceName, "myService");
        strcpy(policyName, "myPolicy");
        status[0]=\0;
        serviceName_ind=0;
        policyName_ind=0;
        dadFileName_ind=0;
        status_ind=-1;

          /* Call the stored procedure */
        EXEC SQL CALL db2xml.dxxmqShredAll(:serviceName:serviceName_ind,
                          :policyName:policyName_ind,
                          :dadFileName:dadFileName_ind,
                          :status:status_ind);
```

**Related concepts:**

- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqShredCLOB stored procedure

**Purpose:**

Decomposes an incoming XML document from a message queue, based on a DAD file mapping, and stores the content of the XML elements and attributes in specified DB2 tables. The incoming document type is XMLCLOB.

For dxxmqShredCLOB, all tables specified in the DAD file must exist, and all columns and data types that are specified in the DAD must be consistent with the existing tables. This stored procedure requires that the columns specified in the join condition of the DAD, correspond to primary-foreign key relationships in the existing tables. The joint condition columns that are specified in the RDB_node of the root element_node must exist in the tables.

**Syntax:**

```
dxxmqShredCLOB(varchar(48)   serviceName,    /* input */
               varchar(48)   policyName,     /* input */
               varchar(80)   dadFileName,    /* input */
               varchar(10)   status)         /* output */
```

**Parameters:**

*Table 82. dxxmqShredCLOB parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *dadFileName* | The name of the DAD file. The maximum size in 80 bytes. | IN |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Related concepts:**

- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqShredAllCLOB stored procedure

**Purpose:**

Decomposes an incoming XML document from a message queue, based on a DAD file mapping, and stores the content of the XML elements and attributes in specified DB2 database tables.

For dxxmqShredAllCLOB, all tables specified in the DAD file must exist, and all columns and data types that are specified in the DAD must be consistent with the existing tables. This stored procedure requires that the columns specified in the join condition of the DAD, correspond to primary-foreign key relationships in the existing tables. The joint condition columns that are specified in the RDB_node of the root element_node must exist in the tables.

**Syntax:**

```
dxxmqShredCLOB(varchar(48)   serviceName,    /* input */
               varchar(48)   policyName,     /* input */
               varchar(80)   dadFileName,    /* input */
               varchar(10)   status)         /* output */
```

**Parameters:**

*Table 83. dxxmqShredAllCLOB parameters*

| Parameter | Description | IN/OUT Parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *dadFileName* | The name of the DAD file. The maximum size is 80 bytes. | IN |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Related reference:**

• Appendix C, "XML Extender limits," on page 293

## dxxmqInsert stored procedure

**Purpose:**

Breaks down or shreds an incoming XML document from a message queue, and stores the data in new or existing database tables. dxxmqInsert uses a collection name, rather than a DAD file name, to determine how to store the data.

**Syntax:**

```
dxxmqInsert(varchar(48)  serviceName,      /* input */
            varchar(48)   policyName,      /* input */
            varchar(80)   collectionName,  /* input */
            varchar(20)   status)          /* output */
```

**Parameters:**

*Table 84. dxxmqInsert() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *collectionName* | The name of an enabled XML collection. The maximum size is 80 bytes. | IN |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Examples:**

In the following fragment example, the dxxmqInsert() call retrieves the input XML document `order1.xml` from a message queue defined by *serviceName*, decomposes the document, and inserts data into the SALES_ORDER collection tables according to the mapping that is specified in the DAD file with which it was enabled.

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
            EXEC SQL BEGIN DECLARE SECTION;
              char          serviceName[48];
```

```
char          policyName[48];
char          collection[80];     /* name of an XML collection */
char          status[10];

short                 serviceName_ind;
short                 policyName_ind;
short                 collection_ind;
short                 status_ind;
EXEC SQL END DECLARE SECTION;

/* initialize host variable and indicators */
strcpy(serviceName, "myService");
strcpy(policyName, "myPolicy");
strcpy(collection,"sales_ord")
status[0]=\0;
serviceName_ind = 0;
policyName_ind = 0;
collection_ind = 0;
status_ind = -1;

/* Call the stored procedure */
EXEC SQL CALL db2xml.dxxmqInsert(:serviceName:serviceName_ind,
                :policyName:policyName_ind,
                :collection:collection_ind,
                :status:status_ind);
```

**Related concepts:**

- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqInsertCLOB stored procedure

**Purpose:**

Breaks down or shreds an incoming XML document from a message queue, and stores the data in new or existing database tables. dxxmqInsertCLOB uses a collection name, rather than a DAD file name, to determine how to store the data. The incoming document type is XMLCLOB

**Syntax:**

```
dxxmqInsertCLOB(varchar(48)  serviceName,  /* input */
            varchar(48)   policyName,      /* input */
            varchar(80)   collectionName,  /* input */
            varchar(20)   status)          /* output */
```

**Parameters:**

*Table 85. dxxmqInsertCLOB() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *collectionName* | The name of an enabled XML collection. | IN |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Examples:**

In the following fragment example, the dxxmqInsertCLOB() call retrieves the input XML document `order1.xml` from a message queue defined by *serviceName*, decomposes the document, and inserts data into the SALES_ORDER collection tables according to the mapping that is specified in the DAD file with which it was enabled.

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
            EXEC SQL BEGIN DECLARE SECTION;
              char          serviceName[48];
              char          policyName[48];
              char          collection[48];     /* name of an XML collection */
              char          status[10];

              short                 serviceName_ind;
              short                 policyName_ind;
              short                 collection_ind;
              short                 status_ind;
              EXEC SQL END DECLARE SECTION;

              /* initialize host variable and indicators */
              strcpy(serviceName, "myService");
```

```
strcpy(policyName, "myPolicy");
strcpy(collection,"sales_ord")
status[0] = \0;
serviceName_ind = 0;
policyName_ind = 0;
collection_ind = 0;
status_ind = -1;

/* Call the stored procedure */
EXEC SQL CALL db2xml.dxxmqInsertCLOB(:serviceName:serviceName_ind;
                :policyName:policyName_ind,
                :collection:collection_ind,
                :status:status_ind);
```

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqInsertAll stored procedure

**Purpose:**

Breaks down or shreds all incoming XML documents from a message queue, and stores the data in new or existing database tables. dxxmqInsertAll uses a collection name, rather than a DAD file name, to determine how to store the data.

**Syntax:**

```
dxxmqInsertAll(varchar(48)  serviceName,      /* input */
               varchar(48)   policyName,      /* input */
               varchar(48)   collectionName,  /* input */
               varchar(20)   status)          /* output */
```

**Parameters:**

*Table 86. dxxmqInsertAll() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *collectionName* | The name of an enabled XML collection. The maximum size is 80 bytes. | IN |

*Table 86. dxxmqInsertAll() parameters (continued)*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Examples:**

In the following fragment example, the dxxmqInsertAll call retrieves all input XML documents from a message queue defined by *serviceName*, decomposes the documents, and inserts data into the SALES_ORDER collection tables according to the mapping that is specified in the DAD file with which it was enabled.

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
            EXEC SQL BEGIN DECLARE SECTION;
              char          serviceName[48];
              char          policyName[48];
              char          collection[80];     /* name of an XML collection */
              char          status[10];

              short                 serviceName_ind;
              short                 policyName_ind;
              short                 collection_ind;
              short                 status_ind;
              EXEC SQL END DECLARE SECTION;

              /* initialize host variable and indicators */
              strcpy(serviceName, "myService");
              strcpy(policyName, "myPolicy");
              strcpy(collection,"sales_ord");
              status[0]='\0';
              serviceName_ind = 0;
              policyName_ind = 0;
              collection_ind = 0;
              status_ind = -1;

              /* Call the stored procedure */
              EXEC SQL CALL db2xml.dxxmqInsertAll(:serviceName:serviceName_ind,
                              :policyName:policyName_ind,
                              :collection:collection_ind,
                              :status:status_ind);
```

**Related concepts:**

- "XML Extender stored procedures and functions for MQSeries - Overview" on page 201

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

## dxxmqInsertAllCLOB stored procedure

**Purpose:**

Breaks down or shreds all incoming XML documents from a message queue, and stores the data in new or existing database tables. The dxxmqInsertAllCLOB stored procedure uses a collection name, rather than a DAD file name, to determine how to store the data.

**Syntax:**

```
dxxmqInsertAllCLOB(varchar(48)  serviceName,  /* input */
                   varchar(48)   policyName,      /* input */
                   varchar(48)   collectionName,  /* input */
                   varchar(20)   status)          /* output */
```

**Parameters:**

*Table 87. dxxmqInsertAllCLOB() parameters*

| Parameter | Description | IN/OUT parameter |
|---|---|---|
| *serviceName* | A string containing the logical MQSeries destination to which the message is to be sent. When the *serviceName* is listed, it refers to a Service Point defined in the AMT.XML repository file. The DB2.DEFAULT.SERVICE is used when the *serviceName* is not specified. The maximum size of *serviceName* is 48 bytes. | IN |
| *policyName* | A string containing the MQSeries AMI Service Policy used to handle messages. When specified, the *policyName* refers to a policy defined in the AMT.XML repository file. If the *policyName* is not specified, then the default DB2.DEFAULT.POLICY will be used. The maximum size of *policyName* is 48 bytes. | IN |
| *collectionName* | The name of an enabled XML collection. | IN |
| *status* | The text and codes returned that specify whether or not the stored procedure ran successfully, any error codes that are generated, and the number of XML documents which are received or sent to the message queue. | OUT |

**Examples:**

In the following fragment example, the dxxmqInsertAllCLOB call retrieves all input XML documents from a message queue defined by *serviceName*, decomposes the documents, and inserts data into the SALES_ORDER collection tables according to the mapping that is specified in the DAD file with which it was enabled.

```
#include "dxx.h"
#include "dxxrc.h"

EXEC SQL INCLUDE SQLCA;
            EXEC SQL BEGIN DECLARE SECTION;
              char          serviceName[48];
```

```
char          policyName[48];
char          collection[48];     /* name of an XML collection */
char          status[10];

short                 serviceName_ind;
short                 policyName_ind;
short                 collection_ind;
short                 status_ind;
EXEC SQL END DECLARE SECTION;

/* initialize host variable and indicators */
strcpy(serviceName, "myService");
strcpy(policyName, "myPolicy");
strcpy(collection,"sales_ord")
status[0] = '\0';
serviceName_ind = 0;
policyName_ind = 0;
collection_ind = 0;
status_ind = -1;

/* Call the stored procedure */
EXEC SQL CALL db2xml.dxxmqInsertAllCLOB(:serviceName:serviceName_ind;
               :policyName:policyName_ind,
               :collection:collection_ind,
               :status:status_ind);
```

**Related reference:**

- Appendix C, "XML Extender limits," on page 293

# Chapter 12. Extensible stylesheet language transformation (XSLT)

## Creating an HTML document using an XSLT stylesheet

The Extensible stylesheet language transformation (XSLT) consists of a series of markups that can be used to apply formatting rules to each of the elements inside an XML document. XSL works by applying various style rules to the contents of an XML document based on the elements that it encounters. By design, XSLT stylesheets are regular XML documents.

Originally created for page layout, XSLT is now used in a variety of ways. For example, it can be used as a general-purpose translation tool, a system for reorganizing document content, or a way to generate multiple results such as HTML, WAP, and SVG from a single source.

XSLT is a critical bridge between XML processing and more familiar languages such as HTML. XSLT allows you to perform the following actions:

- Transform an XML structure into other data types by removing or replacing the XML tags
- Change the order of the information
- Extract special information
- Sort information

**Prerequisites:**

Before you transform an XML document to HTML, complete the following tasks:

1. Insert an XML document in the doc column of the result_tab table.
2. Create a stylesheet.

You can create your HTML file by using the XSLTransformToFile or the XSLTransformToClob UDFs. The first returns the HTML document in a file while the second returns it as a CLOB

**Procedure:**

To create your HTML document as a file, use the following syntax:

```
SELECT XSLTransformToFile( CAST(doc AS CLOB(4k)),
    'dxx_install\samples\extenders\db2xml\xslt\getstart.xsl',
         0,
   'html\getstart.html')
 FROM RESULT_TAB
```

where *dxx_install* is the directory where you installed DB2 XML Extender.

For your convenience, the samples command file, *dxx_install*\samples\extenders\db2xml\getstart_xslt.cmd, does the setup and invokes XSLTransformToFile.

**Related reference:**

- "XSLTransformToFile() user-defined function" on page 247
- "XSLTransformToClob() user-defined function" on page 246

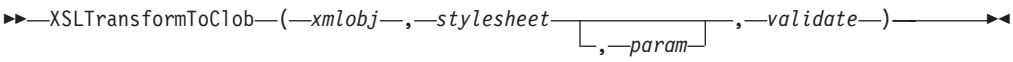
© Copyright IBM Corp. 1999 - 2006

## XSLTransformToClob() user-defined function

**Purpose:**

XSLTransformToClob() reads an XML document as CLOB locator and a stylesheet as CLOB or from a file, and returns the document as CLOB.

**Syntax:**




**Parameters:**

| Parameter | Data type | Description |
|---|---|---|
| xmlobj | CLOB | The XML document |
| stylesheet | CLOB, | The style sheet |
| | VARCHAR | The location and name of the stylesheet input file |
| param | CLOB | The XSLT parameter document. |
| | VARCHAR | The location and name of the XSLT parameter file. |
| validate | INTEGER | Enable (1) or disable (0) validation of the xmlobj |

**Results:**
The XSLTransformToClob() returns a data of CLOB type if successful.
**Examples:**
The following example creates a table, xslt_tab, where the contents of the files getstart.xml (the XML document), getstart.xsl (the stylesheet), and xslt_param.xml (the XSLT parameter document) are inserted.

```
CREATE TABLE xslt_tab(xmlobj CLOB(4k), stylesheet CLOB(4k), parameters CLOB(4k))
INSERT INTO xslt_tab(xmlobj, stylesheet, parameters)
  VALUES( DB2XML.XMLCLOBFromFile(
     dxx_install\samples\extenders\db2xml\xml\getstart.xml ),
    DB2XML.XMLCLOBFromFile(dxx_install\samples\extenders\db2xml\xslt\getstart.xsl ),
    DB2XML.XMLCLOBFromFile(xslt_param.xml))
```

Create a file named xslt_param.xml containing the following lines:

```
<?xml version="1.0"?>
<params xmlns="http://www.ibm.com.XSLtransformParameters">
<param name="noShipments" value="true"/>
<param name="headline">The customers...</param>
</params>
.
.

.
```

**Example 1:** The following example transforms an XML document into a HTML document using the stylesheet that has been stored in the table xslt_tab:

```
SELECT XSLTransformToClob(xmlobj, stylesheet, 0)
FROM xslt_tab
```

**Example 2:** This example transforms an XML document into an HTML document using a stylesheet file

```
SELECT XSLTransformToClob( xmlobj,
  c:\dxx_install\samples\extenders\db2xml\xslt\getstart.xsl', 1)
FROM xslt_tab
```

**Example 3:** In this example the output is changed by additional parameters stored in the table xslt_tab. The XSLT parameter document defines the namespace. The parameters must be wrapped in the <*param*> element. The corresponding value also can be specified in a *value* attribute, or in the content of the <*param*> element.

```
SELECT XSLTransformToClob(
   xmlobj,
   'dxx_install\samples\extenders\db2xml\xslt\getstart.xsl',
   parameters,
   1)
FROM xslt_tab
```

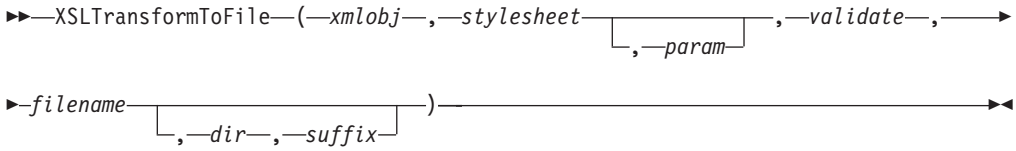
Where *dxx_install* is the directory where you installed DB2 XML Extender.

## XSLTransformToFile() user-defined function

**Purpose:**

Transforms the incoming XML document with the stylesheet and the XSLT parameter document. The transformed XML document is written into a file. When a directory and a suffix are passed as parameters, this UDF creates a file with the specified suffix in the specified directory.

**Syntax:**




**Parameters:**

*Table 88. XSLTransformDir() parameter descriptions*

| Parameter | Data type | Description |
|---|---|---|
| *xmlobj* | CLOB | The XML document |
| *stylesheet* | CLOB | The style sheet |
| | VARCHAR | The location and name of the stylesheet input file |
| *param* | CLOB | The XSLT parameter document |
| | VARCHAR | The location and name of the XSLT parameter file |
| *validate* | INTEGER | Enable (1) or disable (0) validation of the xmlobj |
| *filename* | VARCHAR | The name of the output file |
| *dir* | VARCHAR | The directory of the output file |

*Table 88. XSLTransformDir() parameter descriptions (continued)*

| Parameter | Data type | Description |
|---|---|---|
| *suffix* | VARCHAR | The suffix of the output file |

**Results:**
The XSLTransformToFile() returns a VARCHAR for the written file name.
**Examples:**
The following example creates a table, xslt_tab, where the contents of the files getstart.xml (the XML document), getstart.xsl (the stylesheet), and xslt_param.xml (the XSLT parameter document) are inserted.

```
CREATE TABLE xslt_tab(xmlobj CLOB(4k), stylesheet CLOB(4k), parameters CLOB(4k))
INSERT INTO xslt_tab(xmlobj, stylesheet, parameters)
VALUES( DB2XML.XMLCLOBFromFile(
        dxx_install\samples\extenders\db2xml\xml\getstart.xml ),
        DB2XML.XMLCLOBFromFile(dxx_install\samples\
             extenders\db2xml\xslt\getstart.xsl ),
        DB2XML.XMLCLOBFromFile(xslt_param.xml))
```

where *dxx_install* is the directory where you installed DB2 XML Extender.

Create a file named xslt_param.xml containing the following lines:

```
<?xml version="1.0"?>
<params xmlns="http://www.ibm.com.XSLtransformParameters">
<param name="noShipments" value="true"/>
<param name="headline">The customers...</param>
</params>
.
.
.
```

**Example 1:** This example transforms the XML document into an HTML document and writes the created document to the specified file:

```
SELECT XSLTransformFile( xmlobj, stylesheet, 0
   'dxx_install\samples\extenders\db2xml\html\getstart.html')

  FROM xslt_tab
```

where *dxx_install* is the directory where you installed DB2 XML Extender.

**Example 2:** This example writes an HTML document to a file using a stylesheet file. Validation is enabled but the result is the same. This feature is necessary to include default values from an XML schema in the transformation process.

```
SELECT XSLTransformToFile(xmlobj,
       'dxx_install\samples\extenders\db2xml\xslt\getstart.xsl', 1,
       'dxx_install\samples\extenders\db2xml\html\getstart.html')
       FROM xslt_tab
```

Where *dxx_install* is the directory where you installed DB2 XML Extender.

**Example 3:** In this example the output is changed by additional parameters stored in the table xslt_tab. The XSLT parameter document defines the namespace. The parameters must be wrapped in the <*param*> element. The corresponding value also can be specified in a *value* attribute, or in the content of the <*param*> element.

```
SELECT XSLTransformToFile(
     xmlobj,
     'dxx_install\samples\extenders\db2xml\getstart.xsl',
     parameters, 1, 'dxx_install\samples\extenders\db2xml\getstart.html')
FROM xslt_tab
```

Where *dxx_install* is the directory where you installed DB2 XML Extender.

**Example 4:** This example writes the transformed document into a file with an .html suffix into the *dxx_install*\samples\extenders\db2xml\html directory using the stylesheet in the stylesheet column. The transformed document file name is stored in an additional column in the table.

```
UPDATE TABLE xslt_tab ADD COLUMN filename VARCHAR(512)
UPDATE TABLE xslt_tab SET filename =
  XSLTransformToFile(xmlobj,stylesheet, parameters, 1,
    'dxx_install\samples\extenders\db2xml\html'
          , 'html')
  FROM xslt_tab
```

Where *dxx_install* is the directory where you installed DB2 XML Extender.

# Chapter 13. XML Extender administration support tables

When a database is enabled, a DTD repository table (DTD_REF) and an XML_USAGE table are created. The DTD_REF table contains information about all of the DTDs. The XML_USAGE table stores common information for each XML-enabled column.

## DTD reference table

The XML Extender also serves as an XML DTD repository. When a database is XML-enabled, a DTD repository table, DTD_REF, is created. Each row of this table represents a DTD with additional metadata information. You can access this table, and insert your own DTDs. The DTDs in the DTD_REF table are used to validate XML documents and to help applications to define a DAD file. It has the schema name of DB2XML. A DTD_REF table can have the columns shown in Table 89.

*Table 89. DTD_REF table*

| Column name | Data type | Description |
|---|---|---|
| DTDID | VARCHAR(128) | The primary key (unique and not NULL). It is used to identify the DTD. When the DTD is specified in the DAD file, the DAD file must adhere to the schema that is defined by the DTD. |
| CONTENT | XMLCLOB | The content of the DTD. |
| USAGE_COUNT | INTEGER | The number of XML columns and XML collections in the database that use the DTD to define their DAD files. |
| AUTHOR | VARCHAR(128) | The author of the DTD. This information is optional. |
| CREATOR | VARCHAR(128) | The user ID that does the first insertion. This column is optional. |
| UPDATOR | VARCHAR(128) | The user ID that does the last update. This column is optional. |

A DTD can be modified by the application only when the USAGE_COUNT is zero.

## XML usage table (XML_USAGE)

The XML_USAGE table stores common information for each XML-enabled column. The XML_USAGE table's schema name is DB2XML, and its primary key is (*table_name*, *col_name*). An XML_USAGE table is created at the time the database is enabled. The columns in the XML_USAGE table are shown in Table 90.

*Table 90. XML_USAGE table*

| Column name | Description |
|---|---|
| table_schema | For an XML column, the schema name of the user table that contains an XML column. For an XML collection, a value of DXX_COLL as the default schema name. |

© Copyright IBM Corp. 1999 - 2006

*Table 90. XML_USAGE table (continued)*

| Column name | Description |
|---|---|
| table_name | For an XML column, the name of the user table that contains an XML column. For an XML collection, a value DXX_COLLECTION, which identifies the entity as a collection. |
| col_name | The name of the XML column or XML collection. It is part of the composite key along with the table_name. |
| DTDID | A string associating a DTD inserted into DTD_REF with a DTD specified in a DAD file; this value must match the value of the DTDID element in the DAD. This column is a foreign key. |
| DAD | The content of the DAD file that is associated with the XML column or XML collection. |
| access_mode | Specifies which access mode is used: 1 for XML collection, 0 for XML column |
| default_view | Stores the default view name if there is one. |
| trigger_suffix | Not NULL. For unique trigger names. |
| validation | Has a value of 1 to validate, and 0 to skip validation. |
| version | The trigger version of the XML column. |
| root_id | The root ID of the XML column. |
| side_tab_qual | The side table qualifier. |

Do not add, modify, or delete entries from the XML_USAGE table; it is for XML Extender internal use only.

# Chapter 14. Troubleshooting

## Troubleshooting XML Extender

All embedded SQL statements in your program and DB2 database command line interface (CLI) calls in your program, including those that invoke the DB2 database XML Extender user-defined functions (UDFs), generate codes that indicate whether the embedded SQL statement or DB2 database CLI call ran successfully.

Your program can retrieve information that supplements these codes including SQLSTATE information and error messages. You can use this diagnostic information to isolate and fix problems in your program.

Occasionally the source of a problem cannot be easily diagnosed. In these cases, you might need to provide information to IBM Software Support to isolate and fix the problem. The XML Extender includes a trace facility that records XML Extender activity. The trace information can be valuable input to IBM Software Support. You should use the trace facility only under instruction from IBM Software Support.

This section describes the trace facility, and error codes and messages.

**Related reference:**

- "SQLSTATE codes and associated message numbers for XML Extender" on page 255
- "Starting the trace for XML Extender"
- "Stopping the trace" on page 254
- "XML Extender messages" on page 260

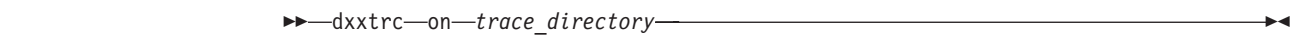
## Starting the trace for XML Extender

**Purpose:**

Records the XML Extender server activity. To start the trace, apply the `on` option to **dxxtrc**, along with the name of an existing directory to contain the trace file. When the trace is turned on, the file, `dxx`*`INSTANCE`*`.trc`, is placed in the specified directory. *INSTANCE* is the value of DB2INSTANCE. Each DB2 instance has its own log file. The trace file is not limited in size.

**Syntax:**

**Starting the trace:**




**Parameters:**

© Copyright IBM Corp. 1999 - 2006

*Table 91. Trace parameters*

| Parameter | Description |
|---|---|
| *trace_directory* | Name of an existing path and directory where the dxx*INSTANCE*.trc is placed. Required, no default. |

**Examples:**
The following example demonstrates starting the trace for an instance db2inst1. The trace file, dxxdb2inst1.trc, is placed in the /home/db2inst1/dxx_install/log directory.

```
dxxtrc on /home/db2inst1/dxx_install/log
```

## Stopping the trace

**Purpose:**

Turns the trace off. Trace information is no longer logged.

**Recommendation:** Because running the trace log file size is not limited and can impact performance, turn trace off in a production environment.

**Syntax:**

**Stopping the trace:**

**Examples:**
This example shows that the trace facility is turned off.

```
dxxtrc off
```

## XML Extender UDF return codes

Embedded SQL statements return codes in the SQLCODE, SQLWARN, and SQLSTATE fields of the SQLCA structure. This structure is defined in an SQLCA INCLUDE file.

DB2 CLI calls return SQLCODE and SQLSTATE values that you can retrieve using the SQLError function.

An SQLCODE value of 0 means that the statement ran successfully (with possible warning conditions). A positive SQLCODE value means that the statement ran successfully but with a warning. (Embedded SQL statements return information about the warning that is associated with 0 or positive SQLCODE values in the SQLWARN field.) A negative SQLCODE value means that an error occurred.

DB2 associates a message with each SQLCODE value. If an XML Extender UDF encounters a warning or error condition, it passes associated information to DB2 database for inclusion in the SQLCODE message.

Embedded SQL statements and DB2 database CLI calls that invoke the DB2 XML Extender UDFs might return SQLCODE messages and SQLSTATE values that are unique to these UDFs, but DB2 database returns these values in the same way that it does for other embedded SQL statements or other DB2 database CLI calls. Thus,

the way that you access these values is the same as for embedded SQL statements or DB2 database CLI calls that do not start the DB2 database XML Extender UDFs.

## XML Extender stored procedure return codes

The XML Extender provides return codes to help resolve problems with stored procedures. When you receive a return code from a stored procedure, check the following file, which matches the return code with an XML Extender error message number and the symbolic constant.

`dxx_install/include/dxxrc.h`

**Related reference:**

- "SQLSTATE codes and associated message numbers for XML Extender"

## SQLSTATE codes and associated message numbers for XML Extender

*Table 92. SQLSTATE codes and associated message numbers*

| SQLSTATE | Message Number | Description |
|---|---|---|
| 00000 | DXX*nnnn*I | No error occurred. |
| 01HX0 | DXXD003W | The element or attribute specified in the path expression is missing from the XML document. |
| 38X00 | DXXC000E | The XML Extender is unable to open the specified file. |
| 38X01 | DXXA072E | XML Extender tried to automatically bind the database before enabling it, but could not find the bind files. |
| | DXXC001E | The XML Extender could not find the file specified. |
| 38X02 | DXXC002E | The XML Extender is unable to read data from the specified file. |
| 38X03 | DXXC003E | The XML Extender is unable to write data to the file. |
| | DXXC011E | The XML Extender is unable to write data to the trace control file. |
| 38X04 | DXXC004E | The XML Extender was unable to operate the specified locator. |
| 38X05 | DXXC005E | The file size is greater than the XMLVarchar size, and the XML Extender is unable to import all the data from the file. |
| 38X06 | DXXC006E | The file size is greater than the size of the XMLCLOB, and the XML Extender is unable to import all the data from the file. |
| 38X07 | DXXC007E | The number of bytes in the LOB locator does not equal the file size. |

*Table 92. SQLSTATE codes and associated message numbers (continued)*

| SQLSTATE | Message Number | Description |
|---|---|---|
| 38X08 | DXXD001E | A scalar extraction function used a location path that occurs multiple times. A scalar function can use only a location path that does not have multiple occurrence. |
| 38X09 | DXXD002E | The path expression is syntactically incorrect. |
| 38X10 | DXXG002E | The XML Extender was unable to allocate memory from the operating system. |
| 38X11 | DXXA009E | This stored procedure is for an XML column only. |
| 38X12 | DXXA010E | While attempting to enable the column, the XML Extender could not find the DTD ID, which is the identifier specified for the DTD in the document access definition (DAD) file. |
| | DXXQ060E | The XML Extender could not find the SCHEMA ID while attempting to enable the column. The SCHEMA ID corresponds to the value of the location attribute of the nonamespacelocation tag which is inside the schemabindings tag in the DAD file. |
| 38X13 | DXXQ072E | The XML Extender uses the XML Parser that is delivered with the XML Toolkit for iSeries. |
| 38X14 | DXXD000E | There was an attempt to store an invalid document into a table. Validation failed. |
| 38X15 | DXXA056E | The validation element in the document access definition (DAD) file is wrong or missing. |
| | DXXA057E | The name attribute of a side table in the document access definition (DAD) file is wrong or missing. |
| | DXXA058E | The name attribute of a column in the document access definition (DAD) file is wrong or missing. |
| | DXXA059E | The type attribute of a column in the document access definition (DAD) file is wrong or missing. |
| | DXXA060E | The path attribute of a column in the document access definition (DAD) file is wrong or missing. |
| | DXXA061E | The multi_occurrence attribute of a column in the document access definition (DAD) file is wrong or missing. |

*Table 92. SQLSTATE codes and associated message numbers (continued)*

| SQLSTATE | Message Number | Description |
|---|---|---|
| | DXXQ000E | A mandatory element is missing from the document access definition (DAD) file. |
| | DXXQ056E | The specified element/attribute cannot be mapped to a column that is specified as part of a foreign key. Data values for foreign keys are determined by that of the primary keys; a mapping of the specified element/attribute in the XML document to a table and column is not necessary. |
| | DXXQ057E | The schemabindings and DTD ID tags cannot exist together in the DAD file. |
| | DXXQ058E | The nonamespacelocation tag inside the schemabindings tag is missing in the DAD file. |
| | DXXQ059E | The doctype tag cannot be located inside the XCollection tag in the DAD for schema validation. |
| | DXXQ062E | This error condition is usually caused by a missing multi_occurrence = YES specification on the parent element_node of the given element or attribute. |
| | DXXQ063E | The value of the multi_occurrence attribute on the specified element_node in the document access definition (DAD) file is wrong or missing. The value must be 'yes' or 'no', case insensitive. |
| | DXXQ064E | A key column specified in the join condition was not mapped to any element or attribute node. |
| 38X16 | DXXG004E | A null value for a required parameter was passed to an XML stored procedure. |
| 38X17 | DXXQ001E | The SQL statement in the document access definition (DAD) file or the statement that overrides it is not valid. A SELECT statement is required for generating XML documents. |
| 38X18 | DXXG001E | XML Extender encountered an internal error. |
| | DXXG006E | XML Extender encountered an internal error while using CLI. |

*Table 92. SQLSTATE codes and associated message numbers (continued)*

| SQLSTATE | Message Number | Description |
|---|---|---|
| 38X19 | DXXQ002E | The system is running out of space in memory or disk. There is no space to contain the resulting XML documents. |
| 38X20 | DXXQ003W | The user-defined SQL query generates more XML documents than the specified maximum. Only the specified number of documents are returned. |
| 38X21 | DXXQ004E | The specified column is not one of the columns in the result of the SQL query. |
| 38X22 | DXXQ005E | The mapping of the SQL query to XML is incorrect. |
| 38X23 | DXXQ006E | An attribute_node element in the document access definition (DAD) file does not have a name attribute. |
| 38X24 | DXXQ007E | The attribute_node element in the document access definition (DAD) does not have a column element or RDB_node. |
| 38X25 | DXXQ008E | A text_node element in the document access definition (DAD) file does not have a column element. |
| 38X26 | DXXQ009E | The specified result table could not be found in the system catalog. |
| 38X27 | DXXQ010E<br>DXXQ040E | The RDB_node of the attribute_node or text_node must have a table. |
| | DXXQ011E | The RDB_node of the attribute_node or text_node must have a column. |
| | DXXQ017E | An XML document generated by the XML Extender is too large to fit into the column of the result table. |
| | DXXQ040E | The specified element name in document access definition (DAD) file is wrong. |
| 38X28 | DXXQ012E | XML Extender could not find the expected element while processing the DAD. |
| | DXXQ016E | All tables must be defined in the RDB_node of the top element in the document access definition (DAD) file. Sub-element tables must match the tables defined in the top element. The table name in this RDB_node is not in the top element. |

*Table 92. SQLSTATE codes and associated message numbers (continued)*

| SQLSTATE | Message Number | Description |
|---|---|---|
| 38X29 | DXXQ013E | The element table or column must have a name in the document access definition (DAD) file. |
| | DXXQ015E | The condition in the condition element in the document access definition (DAD) file has an invalid format. |
| | DXXQ061E | The format of the string representation is invalid. If the string is a date, time, or timestamp value, the syntax does not conform to its data type. |
| 38X30 | DXXQ014E | An element_node element in the document access definition (DAD) file does not have a name attribute. |
| | DXXQ018E | The ORDER BY clause is missing from the SQL statement in a document access definition (DAD) file that maps SQL to XML. |
| 38X31 | DXXQ019E | The objids element does not have a column element in the document access definition (DAD) file that maps SQL to XML. |
| 38X33 | DXXG005E | This parameter is not supported in this release. It will be supported in the future release. |
| 38X34 | DXXG000E | An invalid file name was specified. |
| 38X36 | DXXA073E | The database was not bound when you tried to enable it. |
| 38X37 | DXXG007E | The server operating system locale is inconsistent with the DB2 code page. |
| 38X38 | DXXG008E | The server operating system locale can not be found in the code page table. |
| 38X41 | DXXQ048E | The stylesheet processor returned an internal error. The XML document or the stylesheet might not be vaild. |
| 38X42 | DXXQ049E | The specified output file already exists in this directory. |
| 38X43 | DXXQ050E | The UDF was unable to create a unique file name for the output document in the specified directory because it does not have access. All file names that can be generated are in use or directory might not exist. |
| 38X44 | DXXQ051E | One or more input or output parameters have no valid value. |
| 38X45 | DXXQ055E | ICU error encountered during conversion operation. |

# XML Extender messages

**DXXA000I Enabling column** <*column_name*>**. Please Wait.**

**Explanation:** This is an informational message.

**User response:** No action required.

**DXXA001S An unexpected error occurred in build** <*build_ID*>**, file** <*file_name*>**, and line** <*line_number*>**.**

**Explanation:** An unexpected error occurred.

**User response:** If this error persists, contact your Software Service Provider. When reporting the error, be sure to include all the message text, the trace file, and an explanation of how to reproduce the problem.

**DXXA002I Connecting to database** <*database*>**.**

**Explanation:** This is an informational message.

**User response:** No action required.

**DXXA003E Cannot connect to database** <*database*>**.**

**Explanation:** The database specified might not exist or could be corrupted.

**User response:**

1. Ensure the database is specified correctly.
2. Ensure the database exists and is accessible.
3. Determine if the database is corrupted. If it is, ask your database administrator to recover it from a backup.

**DXXA004E Cannot enable database** <*database*>**.**

**Explanation:** The database might already be enabled or might be corrupted.

**User response:**

1. Determine if the database is enabled.
2. Determine if the database is corrupted. If it is, ask your database administrator to recover it from a backup.

**DXXA005I Enabling database** <*database*>**. Please wait.**

**Explanation:** This is an informational message.

**User response:** No action required.

**DXXA006I The database** <*database*> **was enabled successfully.**

**Explanation:** This is an informational message.

**User response:** No action required.

**DXXA007E Cannot disable database** <*database*>**.**

**Explanation:** The database cannot be disabled by XML Extender if it contains any XML columns or collections.

**User response:** Backup any important data, disable any XML columns or collections, and update or drop any tables until there are no XML data types left in the database.

**DXXA008I Disabling column** <*column_name*>**. Please Wait.**

**Explanation:** This is an information message.

**User response:** No action required.

**DXXA009E Xcolumn tag is not specified in the DAD file.**

**Explanation:** This stored procedure is for XML Column only.

**User response:** Ensure the Xcolumn tag is specified correctly in the DAD file.

**DXXA010E Attempt to find DTD ID** <*dtdid*> **failed.**

**Explanation:** While attempting to enable the column, the XML Extender could not find the DTD ID, which is the identifier specified for the DTD in the document access definition (DAD) file.

**User response:** Ensure the correct value for the DTD ID is specified in the DAD file.

**DXXA011E Inserting a record into DB2XML.XML_USAGE table failed.**

**Explanation:** While attempting to enable the column, the XML Extender could not insert a record into the DB2XML.XML_USAGE table.

**User response:** Ensure the DB2XML.XML_USAGE table exists and that a record by the same name does not already exist in the table.

**DXXA012E Attempt to update DB2XML.DTD_REF table failed.**

**Explanation:** While attempting to enable the column, the XML Extender could not update the DB2XML.DTD_REF table.

**User response:** Ensure the DB2XML.DTD_REF table exists. Determine whether the table is corrupted or if the administration user ID has the correct authority to update the table.

**DXXA013E Attempt to alter table** <*table_name*> **failed.**

**Explanation:** While attempting to enable the column, the XML Extender could not alter the specified table.

**User response:** Check the privileges required to alter the table.

**DXXA014E The specified root ID column:** <*root_id*> **is not a single primary key of table** <*table_name*>**.**

**Explanation:** The root ID specified is either not a key, or it is not a single key of table *table_name*.

**User response:** Ensure the specified root ID is the single primary key of the table.

**DXXA015E The column DXXROOT_ID already exists in table** <*table_name*>**.**

**Explanation:** The column DXXROOT_ID exists, but was not created by XML Extender.

**User response:** Specify a primary column for the root ID option when enabling a column, using a different different column name.

**DXXA016E The input table** <*table_name*> **does not exist.**

**Explanation:** The XML Extender was unable to find the specified table in the system catalog.

**User response:** Ensure that the table exists in the database, and is specified correctly.

**DXXA017E The input column** <*column_name*> **does not exist in the specified table** <*table_name*>**.**

**Explanation:** The XML Extender was unable to find the column in the system catalog.

**User response:** Ensure the column exists in a user table.

**DXXA018E The specified column is not enabled for XML data.**

**Explanation:** While attempting to disable the column, XML Extender could not find the column in the DB2XML.XML_USAGE table, indicating that the column is not enabled. If the column is not XML-enabled, you do not need to disable it.

**User response:** No action required.

**DXXA019E A input parameter required to enable the column is null.**

**Explanation:** A required input parameter for the enable_column() stored procedure is null.

**User response:** Check all the input parameters for the enable_column() stored procedure.

**DXXA020E Columns cannot be found in the table** <*table_name*>**.**

**Explanation:** While attempting to create the default view, the XML Extender could not find columns in the specified table.

**User response:** Ensure the column and table name are specified correctly.

**DXXA021E Cannot create the default view** <*default_view*>**.**

**Explanation:** While attempting to enable a column, the XML Extender could not create the specified view.

**User response:** Ensure that the default view name is unique. If a view with the name already exists, specify a unique name for the default view.

**DXXA022I Column** <*column_name*> **enabled.**

**Explanation:** This is an informational message.

**User response:** No response required.

**DXXA023E Cannot find the DAD file.**

**Explanation:** While attempting to disable a column, the XML Extender was unable to find the document access definition (DAD) file.

**User response:** Ensure you specified the correct database name, table name, or column name.

**DXXA024E The XML Extender encountered an internal error while accessing the system catalog tables.**

**Explanation:** The XML Extender was unable to access system catalog table.

**User response:** Ensure the database is in a stable state.

**DXXA025E Cannot drop the default view** <*default_view*>**.**

**Explanation:** While attempting to disable a column, the XML Extender could not drop the default view.

**User response:** Ensure the administration user ID for XML Extender has the privileges necessary to drop the default view.

**DXXA026E Unable to drop the side table <*side_table*>.**

**Explanation:** While attempting to disable a column, the XML Extender was unable to drop the specified table.

**User response:** Ensure that the administrator user ID for XML Extender has the privileges necessary to drop the table.

**DXXA027E Could not disable the column.**

**Explanation:** XML Extender could not disable a column because an internal trigger failed. Possible causes:

- The system is out of memory.
- A trigger with this name does not exist.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA028E Could not disable the column.**

**Explanation:** XML Extender could not disable a column because an internal trigger failed. Possible causes:

- The system is out of memory.
- A trigger with this name does not exist.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA029E Could not disable the column.**

**Explanation:** XML Extender could not disable a column because an internal trigger failed. Possible causes:

- The system is out of memory.
- A trigger with this name does not exist.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA030E Could not disable the column.**

**Explanation:** XML Extender could not disable a column because an internal trigger failed. Possible causes:

- The system is out of memory.
- A trigger with this name does not exist.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA031E Unable to reset the DXXROOT_ID column value in the application table to NULL.**

**Explanation:** While attempting to disable a column, the XML Extender was unable to set the value of DXXROOT_ID in the application table to NULL.

**User response:** Ensure that the administrator user ID for XML Extender has the privileges necessary to alter the application table.

**DXXA032E Decrement of USAGE_COUNT in DB2XML.XML_USAGE table failed.**

**Explanation:** While attempting to disable the column, the XML Extender was unable to reduce the value of the USAGE_COUNT column by one.

**User response:** Ensure that the DB2XML.XML_USAGE table exists and that the administrator user ID for XML Extender has the necessary privileges to update the table.

**DXXA033E Attempt to delete a row from the DB2XML.XML_USAGE table failed.**

**Explanation:** While attempting to disable a column, the XML Extender was unable to delete its associate row in the DB2XML.XML_USAGE table.

**User response:** Ensure that the DB2XML.XML_USAGE table exists and that the administration user ID for XML Extender has the privileges necessary to update this table.

**DXXA034I XML Extender has successfully disabled column <*column_name*>.**

**Explanation:** This is an informational message

**User response:** No action required.

**DXXA035I XML Extender is disabling database <*database*>. Please wait.**

**Explanation:** This is an informational message.

**User response:** No action is required.

**DXXA036I XML Extender has successfully disabled database <*database*>.**

**Explanation:** This is an informational message.

**User response:** No action is required.

**DXXA037E The specified table space name is longer than 18 characters.**

**Explanation:** The table space name cannot be longer than 18 alphanumeric characters.

**User response:** Specify a name less than 18 characters.

**DXXA038E The specified default view name is longer than 18 characters.**

**Explanation:** The default view name cannot be longer than 18 alphanumeric characters.

**User response:** Specify a name less than 18 characters.

**DXXA039E The specified ROOT_ID name is longer than 18 characters.**

**Explanation:** The ROOT_ID name cannot be longer than 18 alphanumeric characters.

**User response:** Specify a name less than 18 characters.

**DXXA046E Unable to create the side table <*side_table*>.**

**Explanation:** While attempting to enable a column, the XML Extender was unable to create the specified side table.

**User response:** Ensure that the administrator user ID for XML Extender has the privileges necessary to create the side table.

**DXXA047E Could not enable the column.**

**Explanation:** XML Extender could not enable a column because an internal trigger failed. Possible causes:

- The DAD file has incorrect syntax.
- The system is out of memory.
- Another trigger exists with the same name.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA048E Could not enable the column.**

**Explanation:** XML Extender could not enable a column because an internal trigger failed. Possible causes:

- The DAD file has incorrect syntax.
- The system is out of memory.
- Another trigger exists with the same name.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA049E Could not enable the column.**

**Explanation:** XML Extender could not enable a column because an internal trigger failed. Possible causes:

- The DAD file has incorrect syntax.
- The system is out of memory.
- Another trigger exists with the same name.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA050E Could not enable the column.**

**Explanation:** XML Extender could not enable a column because an internal trigger failed. Possible causes:

- The DAD file has incorrect syntax.
- The system is out of memory.
- Another trigger exists with the same name.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA051E Could not disable the column.**

**Explanation:** XML Extender could not disable a column because an internal trigger failed. Possible causes:

- The system is out of memory.
- A trigger with this name does not exist.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA052E Could not disable the column.**

**Explanation:** XML Extender could not disable a column because an internal trigger failed. Possible causes:

- The DAD file has incorrect syntax.
- The system is out of memory.
- Another trigger exists with the same name.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

**DXXA053E Could not enable the column.**

**Explanation:** XML Extender could not enable a column because an internal trigger failed. Possible causes:

- The DAD file has incorrect syntax.
- The system is out of memory.
- Another trigger exists with the same name.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

---

**DXXA054E Could not enable the column.**

**Explanation:** XML Extender could not enable a column because an internal trigger failed. Possible causes:

- The DAD file has incorrect syntax.
- The system is out of memory.
- Another trigger exists with the same name.

**User response:** Use the trace facility to create a trace file and try to correct the problem. If the problem persists, contact your Software Service Provider and provide the trace file.

---

**DXXA056E The validation value** <*validation_value*> **in the DAD file is invalid.**

**Explanation:** The validation element in document access definition (DAD) file is wrong or missing.

**User response:** Ensure that the validation element is specified correctly in the DAD file.

---

**DXXA057E A side table name** <*side_table_name*> **in DAD is invalid.**

**Explanation:** The name attribute of a side table in the document access definition (DAD) file is wrong or missing.

**User response:** Ensure that the name attribute of a side table is specified correctly in the DAD file.

---

**DXXA058E A column name** <*column_name*> **in the DAD file is invalid.**

**Explanation:** The name attribute of a column in the document access definition (DAD) file is wrong or missing.

**User response:** Ensure that the name attribute of a column is specified correctly in the DAD file.

---

**DXXA059E The type** <*column_type*> **of column** <*column_name*> **in the DAD file is invalid.**

**Explanation:** The type attribute of a column in the document access definition (DAD) file is wrong or missing.

**User response:** Ensure that the type attribute of a column is specified correctly in the DAD file.

---

**DXXA060E The path attribute** <*location_path*> **of** <*column_name*> **in the DAD file is invalid.**

**Explanation:** The path attribute of a column in the document access definition (DAD) file is wrong or missing.

**User response:** Ensure that the path attribute of a column is specified correctly in the DAD file.

---

**DXXA061E The multi_occurrence attribute** <*multi_occurrence*> **of** <*column_name*> **in the DAD file is invalid.**

**Explanation:** The multi_occurrence attribute of a column in the document access definition (DAD) file is wrong or missing.

**User response:** Ensure that the multi_occurrence attribute of a column is specified correctly in the DAD file.

---

**DXXA062E Unable to retrieve the column number for** <*column_name*> **in table** <*table_name*>**.**

**Explanation:** XML Extender could not retrieve the column number for *column_name* in table *table_name* from the system catalog.

**User response:** Make sure the application table is well defined.

---

**DXXA063I Enabling collection** <*collection_name*>**. Please Wait.**

**Explanation:** This is an information message.

**User response:** No action required.

---

**DXXA064I Disabling collection** <*collection_name*>**. Please Wait.**

**Explanation:** This is an information message.

**User response:** No action required.

---

**DXXA065E Calling stored procedure <*procedure_name*> failed.**

**Explanation:** Check the shared library db2xml and see if the permission is correct.

**User response:** Make sure the client has permission to run the stored procedure.

**DXXA066I XML Extender has successfully disabled collection <*collection_name*>.**

**Explanation:** This is an informational message.

**User response:** No response required.

**DXXA067I XML Extender has successfully enabled collection <*collection_name*>.**

**Explanation:** This is an informational message.

**User response:** No response required.

**DXXA068I XML Extender has successfully turned the trace on.**

**Explanation:** This is an informational message.

**User response:** No response required.

**DXXA069I XML Extender has successfully turned the trace off.**

**Explanation:** This is an informational message.

**User response:** No response required.

**DXXA070W The database has already been enabled.**

**Explanation:** The enable database command was executed on the enabled database

**User response:** No action is required.

**DXXA071W The database has already been disabled.**

**Explanation:** The disable database command was executed on the disabled database

**User response:** No action is required.

**DXXA072E XML Extender couldn't find the bind files. Bind the database before enabling it.**

**Explanation:** XML Extender tried to automatically bind the database before enabling it, but could not find the bind files

**User response:** Bind the database before enabling it.

**DXXA073E The database is not bound. Please bind the database before enabling it.**

**Explanation:** The database was not bound when user tried to enable it.

**User response:** Bind the database before enabling it.

**DXXA074E Wrong parameter type. The stored procedure expects a STRING parameter.**

**Explanation:** The stored procedure expects a STRING parameter.

**User response:** Declare the input parameter to be STRING type.

**DXXA075E Wrong parameter type. The input parameter should be a LONG type.**

**Explanation:** The stored procedure expects the input parameter to be a LONG type.

**User response:** Declare the input parameter to be a LONG type.

**DXXA076E XML Extender trace instance ID invalid.**

**Explanation:** Cannot start trace with the instance ID provided.

**User response:** Ensure that the instance ID is a valid iSeries user ID.

**DXXA077E The license key is not valid. See the server error log for more detail.**

**Explanation:** The software license has expired or does not exist.

**User response:** Contact your service provider to obtain a new software license.

**DXXC000E Unable to open the specified file.**

**Explanation:** The XML Extender is unable to open the specified file.

**User response:** Ensure that the application user ID has read and write permission for the file.

**DXXC001E The specified file is not found.**

**Explanation:** The XML Extender could not find the file specified.

**User response:** Ensure that the file exists and the path is specified correctly.

**DXXC002E Unable to read file.**

**Explanation:** The XML Extender is unable to read data from the specified file.

**User response:** Ensure that the application user ID has read permission for the file.

**DXXC003E Unable to write to the specified file.**

**Explanation:** The XML Extender is unable to write data to the file.

**User response:** Ensure that the application user ID has write permission for the file or that the file system has sufficient space.

**DXXC004E Unable to operate the LOB Locator: rc=<*locator_rc*>.**

**Explanation:** The XML Extender was unable to operate the specified locator.

**User response:** Ensure the LOB Locator is set correctly.

**DXXC005E Input file size is greater than XMLVarchar size.**

**Explanation:** The file size is greater than the XMLVarchar size and the XML Extender is unable to import all the data from the file.

**User response:** Use the XMLCLOB column type.

**DXXC006E The input file exceeds the DB2 database LOB limit.**

**Explanation:** The file size is greater than the size of the XMLCLOB and the XML Extender is unable to import all the data from the file.

**User response:** Decompose the file into smaller objects or use an XML collection.

**DXXC007E Unable to retrieve data from the file to the LOB Locator.**

**Explanation:** The number of bytes in the LOB Locator does not equal the file size.

**User response:** Ensure the LOB Locator is set correctly.

**DXXC008E Can not remove the file <*file_name*>.**

**Explanation:** The file has a sharing access violation or is still open.

**User response:** Close the file or stop any processes that are holding the file. You might have to stop and restart DB2.

**DXXC009E Unable to create file to <*directory*> directory.**

**Explanation:** The XML Extender is unable to create a file in directory *directory*.

**User response:** Ensure that the directory exists, that the application user ID has write permission for the directory, and that the file system has sufficient space for the file.

**DXXC010E Error while writing to file <*file_name*>.**

**Explanation:** There was an error while writing to the file *file_name*.

**User response:** Ensure that the file system has sufficient space for the file.

**DXXC011E Unable to write to the trace control file.**

**Explanation:** The XML Extender is unable to write data to the trace control file.

**User response:** Ensure that the application user ID has write permission for the file or that the file system has sufficient space.

**DXXC012E Cannot create temporary file.**

**Explanation:** Cannot create file in system temp directory.

**User response:** Ensure that the application user ID has write permission for the file system temp directory or that the file system has sufficient space for the file.

**DXXC013E The results of the extract UDF exceed the size limit for the UDF return type.**

**Explanation:** The data returned by an extract UDF must fit into the size limit of the return type of the UDF, as defined in the IBM DB2 XML Extender Administration and Programming guide. For example, the results of extractVarchar must be no more than 4000 bytes (including the terminating NULL).

**User response:** Use an extract UDF that has a larger size limit for the return type: 254 bytes for extractChar(), 4 KB for extractVarchar(), and 2 GB for extractClob().

**DXXD000E An invalid XML document is rejected.**

**Explanation:** There was an attempt to store an invalid document into a table. Validation has failed.

**User response:** Check the document with its DTD using an editor that can view invisible invalid characters. To suppress this error, turn off validation in the DAD file.

**DXXD001E** <*location_path*> **occurs multiple times.**

**Explanation:** A scalar extraction function used a location path that occurs multiple times. A scalar function can only use a location path that does not have multiple occurrences.

**User response:** Use a table function (add an 's' to the end of the scalar function name).

---

**DXXD002E A syntax error occurred near position** <*position*> **in the search path.**

**Explanation:** The path expression is syntactically incorrect.

**User response:** Correct the search path argument of the query. Refer to the documentation for the syntax of path expressions.

---

**DXXD003W Path not found. Null is returned.**

**Explanation:** The element or attribute specified in the path expression is missing from the XML document.

**User response:** Verify that the specified path is correct.

---

**DXXG000E The file name** <*file_name*> **is invalid.**

**Explanation:** An invalid file name was specified.

**User response:** Specify a correct file name and try again.

---

**DXXG001E An internal error occurred in build** <*build_ID*>**, file** <*file_name*>**, and line** <*line_number*>**.**

**Explanation:** XML Extender encountered an internal error.

**User response:** Contact your Software Service Provider. When reporting the error, be sure to include all the messages, the trace file and how to reproduce the error.

---

**DXXG002E The system is out of memory.**

**Explanation:** The XML Extender was unable to allocate memory from the operating system.

**User response:** Close some applications and try again. If the problem persists, refer to your operating system documentation for assistance. Some operating systems might require that you reboot the system to correct the problem.

---

**DXXG004E Invalid null parameter.**

**Explanation:** A null value for a required parameter was passed to an XML stored procedure.

**User response:** Check all required parameters in the argument list for the stored procedure call.

---

**DXXG005E Parameter not supported.**

**Explanation:** This parameter is not supported in this release, will be supported in the future release.

**User response:** Set this parameter to NULL.

---

**DXXG006E Internal Error CLISTATE=**<*clistate*>**, RC=**<*cli_rc*>**, build** <*build_ID*>**, file** <*file_name*>**, line** <*line_number*> **CLIMSG=**<*CLI_msg*>**.**

**Explanation:** XML Extender encountered an internal error while using CLI.

**User response:** Contact your Software Service Provider. Potentially this error can be caused by incorrect user input. When reporting the error, be sure to include all output messages, trace log, and how to reproduce the problem. Where possible, send any DADs, XML documents, and table definitions which apply.

---

**DXXG007E Locale** <*locale*> **is inconsistent with DB2 database code page** <*code_page*>**.**

**Explanation:** The server operating system locale is inconsistent with DB2 database code page.

**User response:** Correct the server operating system locale and restart DB2.

---

**DXXG008E Locale** <*locale*> **is not supported.**

**Explanation:** The server operating system locale can not be found in the code page table.

**User response:** Correct the server operating system locale and restart DB2.

---

**DXXG017E The limit for** *XML_Extender_constant* **has been exceeded in build** *build_ID***, file** *file_name***, and line** *line_number***.**

**Explanation:** The limit for the XML Extender constant named was exceeded in the code location specified by the build, file, and line number.

**User response:** Check if your application has exceeded a value in the limits table in the *IBM DB2 XML Extender Administration and Programming Guide*. If no limit has been exceeded, contact your Software Service Provider. When reporting the error, include all output messages, trace files, and information on how to reproduce the problem such as input DADs, XML documents, and table definitions.

---

**DXXM001W A DB2 database error occurred.**

**Explanation:** DB2 encountered the specified error.

**User response:** See any accompanying messages for futher explanation and refer to DB2 database product

messages and codes documentation for your operating system.

---

**DXXQ000E** *<Element>* **is missing from the DAD file.**

**Explanation:** A mandatory element is missing from the document access definition (DAD) file.

**User response:** Add the missing element to the DAD file.

---

**DXXQ001E Invalid SQL statement for XML generation.**

**Explanation:** The SQL statement in the document access definition (DAD) or the one that overrides it is not valid. A SELECT statement is required for generating XML documents.

**User response:** Correct the SQL statement.

---

**DXXQ002E Cannot generate storage space to hold XML documents.**

**Explanation:** The system is running out of space in memory or disk. There is no space to contain the resulting XML documents.

**User response:** Limit the number of documents to be generated. Reduce the size of each documents by removing some unnecessary element and attribute nodes from the document access definition (DAD) file.

---

**DXXQ003W Result exceeds maximum.**

**Explanation:** The user-defined SQL query generates more XML documents than the specified maximum. Only the specified number of documents are returned.

**User response:** No action is required. If all documents are needed, specify zero as the maximum number of documents.

---

**DXXQ004E The column** *<column_name>* **is not in the result of the query.**

**Explanation:** The specified column is not one of the columns in the result of the SQL query.

**User response:** Change the specified column name in the document access definition (DAD) file to make it one of the columns in the result of the SQL query. Alternatively, change the SQL query so that it has the specified column in its result.

---

**DXXQ005E Wrong relational mapping. The element** *<element_name>* **is at a lower level than its child column** *<column_name>***.**

**Explanation:** The mapping of the SQL query to XML is incorrect.

**User response:** Make sure that the columns in the result of the SQL query are in a top-down order of the relational hierarchy. Also make sure that there is a single-column candidate key to begin each level. If such a key is not available in a table, the query should generate one for that table using a table expression and the DB2 database built-in function generate_unique().

---

**DXXQ006E An attribute_node element has no name.**

**Explanation:** An attribute_node element in the document access definition (DAD) file does not have a name attribute.

**User response:** Ensure that every attribute_node has a name in the DAD file.

---

**DXXQ007E The attribute_node** *<attribute_name>* **has no column element or RDB_node.**

**Explanation:** The attribute_node element in the document access definition (DAD) does not have a column element or RDB_node.

**User response:** Ensure that every attribute_node has a column element or RDB_node in the DAD.

---

**DXXQ008E A text_node element has no column element.**

**Explanation:** A text_node element in the document access definition (DAD) file does not have a column element.

**User response:** Ensure that every text_node has a column element in the DAD.

---

**DXXQ009E Result table** *<table_name>* **does not exist.**

**Explanation:** The specified result table could not be found in the system catalog.

**User response:** Create the result table before calling the stored procedure.

---

**DXXQ010E RDB_node of** *<node_name>* **does not have a table in the DAD file.**

**Explanation:** The RDB_node of the attribute_node or text_node must have a table.

**User response:** Specify the table of RDB_node for attribute_node or text_node in the document access definition (DAD) file.

---

**DXXQ011E RDB_node element of** *<node_name>* **does not have a column in the DAD file.**

**Explanation:** The RDB_node of the attribute_node or text_node must have a column.

**User response:** Specify the column of RDB_node for attribute_node or text_node in the document access definition (DAD) file.

**DXXQ012E Errors occurred in DAD.**

**Explanation:** XML Extender could not find the expected element while processing the DAD.

**User response:** Check that the DAD is a valid XML document and contains all the elements required by the DAD DTD. Consult the XML Extender publication for the DAD DTD.

**DXXQ013E The table or column element does not have a name in the DAD file.**

**Explanation:** The element table or column must have a name in the document access definition (DAD) file.

**User response:** Specify the name of table or column element in the DAD.

**DXXQ014E An element_node element has no name.**

**Explanation:** An element_node element in the document access definition (DAD) file does not have a name attribute.

**User response:** Ensure that every element_node element has a name in the DAD file.

**DXXQ015E The condition format is invalid.**

**Explanation:** The condition in the condition element in the document access definition (DAD) has an invalid format.

**User response:** Ensure that the format of the condition is valid.

**DXXQ016E The table name in this RDB_node is not defined in the top element of the DAD file.**

**Explanation:** All tables must be defined in the RDB_node of the top element in the document access definition (DAD) file. Sub-element tables must match the tables defined in the top element. The table name in this RDB_node is not in the top element.

**User response:** Ensure that the table of the RDB node is defined in the top element of the DAD file.

**DXXQ017E The column in the result table** <*table_name*> **is too small.**

**Explanation:** An XML document generated by the XML Extender is too large to fit into the column of the result table.

**User response:** Drop the result table. Create another result table with a bigger column. Rerun the stored procedure.

**DXXQ018E The ORDER BY clause is missing from the SQL statement.**

**Explanation:** The ORDER BY clause is missing from the SQL statement in a document access definition (DAD) file that maps SQL to XML.

**User response:** Edit the DAD file. Add an ORDER BY clause that contains the entity-identifying columns.

**DXXQ019E The element objids has no column element in the DAD file.**

**Explanation:** The objids element does not have a column element in the document access definition (DAD) file that maps SQL to XML.

**User response:** Edit the DAD file. Add the key columns as sub-elements of the element objids.

**DXXQ020I XML successfully generated.**

**Explanation:** The requested XML documents have been successfully generated from the database.

**User response:** No action is required.

**DXXQ021E Table** <*table_name*> **does not have column** <*column_name*>**.**

**Explanation:** The table does not have the specified column in the database.

**User response:** Specify another column name in DAD or add the specified column into the table database.

**DXXQ022E Column** <*column_name*> **of** <*table_name*> **should have type** <*type_name*>**.**

**Explanation:** The type of the column is wrong.

**User response:** Correct the type of the column in the document access definition (DAD).

**DXXQ023E Column** <*column_name*> **of** <*table_name*> **cannot be longer than** <*length*>**.**

**Explanation:** The length defined for the column in the DAD is too long.

**User response:** Correct the column length in the document access definition (DAD).

**DXXQ024E Can not create table** <*table_name*>**.**

**Explanation:** The specified table can not be created.

**User response:** Ensure that the user ID creating the table has the necessary authority to create a table in the database.

**DXXQ025I XML decomposed successfully.**

**Explanation:** An XML document has been decomposed and stored in a collection successfully.

**User response:** No action is required.

**DXXQ026E XML data <*xml_name*> is too large to fit in column <*column_name*>.**

**Explanation:** The specified piece of data from an XML document is too large to fit into the specified column.

**User response:** Increase the length of the column using the ALTER TABLE statement or reduce the size of the data by editing the XML document.

**DXXQ028E Cannot find the collection <*collection_name*> in the XML_USAGE table.**

**Explanation:** A record for the collection cannot be found in the XML_USAGE table.

**User response:** Verify that you have enabled the collection.

**DXXQ029E Cannot find the DAD in XML_USAGE table for the collection <*collection_name*>.**

**Explanation:** A DAD record for the collection cannot be found in the XML_USAGE table.

**User response:** Ensure that you have enabled the collection correctly.

**DXXQ030E Wrong XML override syntax.**

**Explanation:** The XML_override value is specified incorrectly in the stored procedure.

**User response:** Ensure that the syntax of XML_override is correct.

**DXXQ031E Table name cannot be longer than maximum length allowed by DB2.**

**Explanation:** The table name specified by the condition element in the DAD is too long.

**User response:** Correct the length of the table name in document access definition (DAD).

**DXXQ032E Column name cannot be longer than maximum length allowed by DB2.**

**Explanation:** The column name specified by the condition element in the DAD is too long.

**User response:** Correct the length of the column name in the document access definition (DAD).

**DXXQ033E Invalid identifier starting at <*identifier*>**

**Explanation:** The string is not a valid DB2 database SQL identifier.

**User response:** Correct the string in the DAD to conform to the rules for DB2 database SQL identifiers.

**DXXQ034E Invalid condition element in top RDB_node of DAD: <*condition*>**

**Explanation:** The condition element must be a valid WHERE clause consisting of join conditions connected by the conjunction AND.

**User response:** See the XML Extender documentation for the correct syntax of the join condition in a DAD.

**DXXQ035E Invalid join condition in top RDB_node of DAD: <*condition*>**

**Explanation:** Column names in the condition element of the top RDB_node must be qualified with the table name if the DAD specifies multiple tables.

**User response:** See the XML Extender documentation for the correct syntax of the join condition in a DAD.

**DXXQ036E A Schema name specified under a DAD condition tag is longer than allowed.**

**Explanation:** An error was detected while parsing text under a condition tag within the DAD. The condition text contains an id qualified by a schema name that is too long.

**User response:** Correct the text of the condition tags in document access definition (DAD).

**DXXQ037E Cannot generate <*element*> with multiple occurrences.**

**Explanation:** The element node and its descendents have no mapping to database, but its multi_occurrence equals YES.

**User response:** Correct the DAD by either setting the multi_occurrence to NO or create a RDB_node in one of its descendents.

**DXXQ038E The SQL statement is too long: SQL_statement**

**Explanation:** The SQL statement specified in the <*SQL_stmt*> element of DAD exceeds the allowed number of bytes.

**User response:** Reduce the length of the SQL statement to less than or equal to 32765 bytes for Windows and UNIX, or 16380 bytes for OS/390 and iSeries.

**DXXQ039E Too many columns specified for a table in the DAD file.**

**Explanation:** A DAD file used for decomposition or RDB composition can have a maximum of 100 text_node and attribute_node elements that specify unique columns within the same table.

**User response:** Reduce the total number of text_node and attribute_node elements that refer to unique columns within the same table to no more than 100.

---

**DXXQ040E The element name** <*element_name*> **in the DAD file is invalid.**

**Explanation:** The specified element name in the document access definition (DAD) file is wrong.

**User response:** Ensure that the element name is typed correctly in the DAD file. See the DTD for the DAD file.

---

**DXXQ041W XML document successfully generated. One or more override paths specified is invalid and ignored.**

**Explanation:** Specify only one override path.

**User response:** Ensure that the element name is typed correctly in the DAD file. See the DTD for the DAD file.

---

**DXXQ043E Attribute** <*attr_name*> **not found under element** <*elem_name*>**.**

**Explanation:** The attribute <*attr_name*> was not present in element <*elem_name*> or one of its child elements.

**User response:** Ensure the attribute appears in the XML document everywhere that the DAD requires it.

---

**DXXQ044E Element** <*elem_name*> **does not have an ancestor element**<*ancestor*>**.**

**Explanation:** According to the DAD, <*ancestor*> is an ancestor element of <*elem_name*> . In the XML document, one or more element <*elem_name*> does not have such an ancestor.

**User response:** Ensure that the nesting of elements in the XML document conforms to what is specified in the corresponding DAD.

---

**DXXQ045E Subtree under element** <*elem_name*> **contains multiple attributes named**<*attrib_name*>**.**

**Explanation:** A subtree under <*elem_name*> in the XML document contains multiple instances of attribute<*attrib_name*> , which according to the DAD, is to be decomposed into the same row. Elements or attributes that are to be decomposed must have unique names.

**User response:** Ensure that the element or attribute in the subtree has a unique name.

---

**DXXQ046W The DTD ID was not found in the DAD.**

**Explanation:** In the DAD, VALIDATION is set to YES, but the DTDID element is not specified. No validation check is performed.

**User response:** No action is required. If validation is needed, specify the DTDID element in the DAD file.

---

**DXXQ047E Parser error on line <mv> linenumber</mv> column** *colnumber***:** *msg*

**Explanation:** The parser could not parse the document because of the reported error.

**User response:** Correct the error in the document, consulting the XML specifications if necessary.

---

**DXXQ048E Internal error - see trace file.**

**Explanation:** The stylesheet processor returned an internal error. The XML document or the stylesheet might not be valid.

**User response:** Ensure the XML document and the stylesheet are valid.

---

**DXXQ049E The output file already exists.**

**Explanation:** The specified output file already exists in this directory.

**User response:** Change the output path or file name for the output document to a unique name or delete the existing file.

---

**DXXQ050E Unable to create a unique file name.**

**Explanation:** The UDF was unable to create a unique file name for the output document in the specified directory because it does not have access, all file names that can be generated are in use or directory might not exist.

**User response:** Ensure that the UDF has access to the specified directory, change to a directory with available file names.

---

**DXXQ051E No input or output data.**

**Explanation:** One or more input or output parameters have no valid value.

**User response:** Check the statement to see if required parameters are missing.

**DXXQ052E An error occurred while accessing the DB2XML.XML_USAGE table.**

**Explanation:** Either the database has not been enabled or the table DB2XML.XML_USAGE has been dropped.

**User response:** Ensure that the database has been enabled and the table DB2XML.XML_USAGE is accessible.

---

**DXXQ053E An SQL statement failed :** *msg*

**Explanation:** An SQL statement generated during XML Extender processing failed to execute.

**User response:** Examine the trace for more details. If you cannot correct the error condition, contact your softwaresService provider. When reporting the error, be sure to include all the messages, the trace file and how to reproduce the error.

---

**DXXQ054E Invalid input parameter:** *param*

**Explanation:** The specified input parameter to a stored procedure or UDF is invalid.

**User response:** Check the signature of the relevant stored procedure or UDF, and ensure the actual input parameter is correct.

---

**DXXQ055E ICU error:** *uerror*

**Explanation:** ICU error encountered during conversion operation.

**User response:** Report the error to your software service provider. Include trace file, error message, and instructions to reproduce the error.

---

**DXXQ056E Element/attribute** *xmlname* **cannot be mapped to the column designated as part of the foreign key (column** *column* **in table** *table***).**

**Explanation:** The specified element/attribute cannot be mapped to a column that is specified as part of a foreign key. Data values for foreign keys are determined by that of the primary keys; a mapping of the specified element/attribute in the XML document to a table and column is not necessary.

**User response:** Remove the RDB_node mapping to the specified column and table in the DAD.

---

**DXXQ057E The schemabindings and dtdid tags cannot exist together in the DAD file.**

**Explanation:** The schemabindings and dtdid tags cannot exist together in the DAD file.

**User response:** Check that either the schemabindings tag or the dtdid tag exists in the DAD file, but not both.

---

**DXXQ058E The nonamespacelocation tag inside the schemabindings tag is missing in the DAD file.**

**Explanation:** The nonamespacelocation tag inside the schemabindings tag is missing in the DAD file.

**User response:** Add the nonamespacelocation tag to the schemabindings tag.

---

**DXXQ059E The doctype tag cannot be located inside the XCollection tag in the DAD for schema validation.**

**Explanation:** The doctype tag cannot be located inside the XCollection tag in the DAD for schema validation.

**User response:** Remove the doctype tag inside the Xcollection tag for schema validation.

---

**DXXQ060E Attempt to find SCHEMA ID** *schemaid* **failed.**

**Explanation:** The XML Extender could not find the SCHEMA ID while attempting to enable the column. The SCHEMA ID corresponds to the value of the location attribute of the nonamespacelocation tag which is inside the schemabindings tag in the DAD file.

**User response:** Check that the correct value for the SCHEMA ID is specified in the DAD file.

---

**DXXQ061E The format of the string is invalid.**

**Explanation:** The format of the string representation is invalid. If the string is a date, time, or timestamp value, the syntax does not conform to its data type.

**User response:** Check that the format of the date, time, or timestamp value conforms to the format for its data type.

---

**DXXQ062E No rows of result set for** *table* **are left to produce a XML value for** *element***.**

**Explanation:** This error condition is usually caused by a missing multi_occurrence = YES specification on the parent element_node of the given element or attribute.

**User response:** Check the DAD that the value of multi_occurrence on the parent element_node correctly reflects the multiplicity of child element_nodes.

---

**DXXQ063E The multi_occurrence attribute value on** *elementname* **in the DAD file is invalid.**

**Explanation:** The value of the multi_occurrence attribute on the specified element_node in the document access definition (DAD) file is wrong or missing. The value must be 'yes' or 'no', case insensitive.

**User response:** Ensure that the multi_occurrence

attribute is specified correctly in the DAD file.

---

**DXXQ064E Column *column* not found in foreign table *table*.**

**Explanation:** A key column specified in the join condition was not mapped to any element or attribute node.

**User response:** Check to make sure the join condition specified in the DAD file is correct, and all key columns are mapped to element or attribute nodes.

---

**DXXQ065I All triggers relating to XML enabled columns have been successfully regenerated.**

**Explanation:** This is an informational message only.

**User response:** No action required.

---

**DXXQ066E The primary key for table *tablename* does not exist.**

**Explanation:** XML Extender could not determine the primary key for table *tablename*. Check that the primary key for the table was not dropped after the column was enabled for XML.

**User response:** Alter the table to add the primary key specified as the ROOT ID when the column was enabled for XML.

---

**DXXQ067E Attempt to *action* failed.**

**Explanation:** While attempting to *action*, a SQL error occurred.

**User response:** Contact your Software Service Provider. When reporting the error, be sure to include the XML Extender trace file.

---

**DXXQ068E Cannot set current SQLID to [userid]. SQLCODE = [sqlcode].**

**Explanation:** While attempting to set current sqlid to a secondary authorization id, a SQL error occurred.

**User response:** Check that you are specifying a valid secondary authorization id and that you have authorization for the id.

---

**DXXQ069E Cannot find element/attribute *xmlitemname* mapped to table *tabname* as foreign key for table *tab2name*.**

**Explanation:** The mapping, including multi-occurrence usage, as specified by the DAD does not allow the value of xmlitemname to be used to populate a foreign key value of tab2name.

**User response:** Check the following: join condition involving the two tables; the value of multi_occurrence attribute of the highest level element mapping to *tabname* and its wrapper; the presence of other items that is a child of an ancestor of *xmlitemname* mapping to a different table than tabname.

---

**DXXQ070E Cannot open trace file *tracefile* for output.**

**Explanation:** The open of the trace file failed.

**User response:** Ensure that the process writing the trace file has write permission and that the file resides on a read/write file system.

---

**DXXQ071E The number of mapped LOB columns that is allowed for a composition exceeds the maximum of *loblimit*.**

**Explanation:** The number of mapped LOB columns that is specified in a composition DAD exceeds the maximum number allowed. Reduce the number of LOB columns.

**User response:** Check whether the LOB columns that are specified in the DAD have exceeded the maximum number. If the limit has not been exceeded, contact your Software Service Provider. When reporting the error, include all messages, trace files, and information on how to reproduce the problem such as input DADs and table definitions.

---

**DXXQ072E XML Extender for iSeries requires the XML Toolkit for iSeries (*prodid*) with installation of option *parseropt*.**

**Explanation:** The XML Extender uses the XML Parser that is delivered with the XML Toolkit for iSeries.

**User response:** Install the XML Toolkit for iSeries with the required option.

---

**DXXQ073E The database is not enabled.**

**Explanation:** The dxxadm administration command was executed on a database that was not enabled.

**User response:** Enable the database for XML Extender before executing any other XML Extender administration command. Note that disabling a database that has not been XML-enabled will result in the same error.

# Appendix A. Samples

This appendix shows the sample objects that are used with examples in this book.


- "XML DTD sample"
- "XML document sample: getstart.xml"
- "Document access definition files" on page 277
  - "Sample DAD file: XML column" on page 278
  - "Sample DAD file: XML collection: SQL mapping" on page 278
  - "Sample DAD file: XML: RDB_node mapping" on page 280


## XML DTD sample

The following DTD is used for the `getstart.xml` document that is referenced throughout this guide.

```
<!xml encoding="US-ASCII"?>

<!ELEMENT Order (Customer, Part+)>
<!ATTLIST Order key CDATA #REQUIRED>
<!ELEMENT Customer (Name, Email)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Email (#PCDATA)>
<!ELEMENT Part (key, Quantity, ExtendedPrice, Tax, Shipment+)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>
<!ELEMENT ExtendedPrice (#PCDATA)>
<!ELEMENT Tax (#PCDATA)>
<!ATTLIST Part color CDATA #REQUIRED>
<!ELEMENT Shipment (ShipDate, ShipMode)>
<!ELEMENT ShipDate (#PCDATA)>
<!ELEMENT ShipMode (#PCDATA)>
```

*Figure 14. Sample XML DTD: `getstart.dtd`*

## XML document sample: getstart.xml

The following XML document, `getstart.xml`, is the sample XML document that is used in examples throughout this guide. It contains XML tags to form a purchase order.

© Copyright IBM Corp. 1999 - 2006

```
<?xml version="1.0"?>
<!DOCTYPE Order SYSTEM "dxx_install/samples/extenders/db2xml/dtd/getstart.dtd">
<Order key="1">
  <Customer>
    <Name>American Motors</Name>
    <Email>parts@am.com</Email>
  </Customer>
  <Part color="black ">
    <key>68</key>
    <Quantity>36</Quantity>
    <ExtendedPrice>34850.16</ExtendedPrice>
    <Tax>6.000000e-02</Tax>
    <Shipment>
      <ShipDate>1998-08-19</ShipDate>
      <ShipMode>BOAT  </ShipMode>
    </Shipment>
    <Shipment>
       <ShipDate>1998-08-19</ShipDate>
       <ShipMode>AIR   </ShipMode>
    </Shipment>
  </Part>
  <Part color="red   ">
    <key>128</key>
    <Quantity>28</Quantity>
    <ExtendedPrice>38000.00</ExtendedPrice>
    <Tax>7.000000e-02</Tax>
    <Shipment>
      <ShipDate>1998-12-30</ShipDate>
      <ShipMode>TRUCK </ShipMode>
    </Shipment>
  </Part>
</Order>
```

*Figure 15. Sample XML document: getstart.xml*

## Stylesheet sample: getstart.xsl

The following XML stylesheet, `getstart.xsl`, is the sample stylesheet that is included in the samples directory of DB2.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
 <html>
  <head/>
  <body>

    <ol style="list-style:decimal outside">
    <xsl:for-each select="Order">
     <li><b>Orderkey : </b> <xsl:value-of select="@key"/><br/>

      <xsl:for-each select="Customer">
       <b>Customer</b><br/>
        <xsl:for-each select="Name | Email">
     <xsl:value-of select="name()"/><xsl:text> : </xsl:text><xsl:value-of select="."/>
     <xsl:text>, </xsl:text>
        </xsl:for-each>
      </xsl:for-each>

   <br/><br/>
   <ol type="A">
       <xsl:for-each select="Part">
        <li><b>Parts</b><br/>
     Color : <xsl:value-of select="@color"/><xsl:text>, </xsl:text>

          <xsl:for-each select="key | Quantity | ExtendedPrice | Tax">
      <xsl:value-of select="name()"/><xsl:text> : </xsl:text><xsl:value-of select="."/>
      <xsl:text>, </xsl:text>
     </xsl:for-each>

     <br/><br/>
     <ol type="a">
          <xsl:for-each select="Shipment">
           <li><b>Shipment</b><br/>
           <xsl:for-each select="ShipDate | ShipMode">
       <xsl:value-of select="name()"/><xsl:text> : </xsl:text><xsl:value-of select="."/>
       <xsl:text>, </xsl:text>
          </xsl:for-each>
          </li>
          </xsl:for-each>
          </ol>
          <br/>
         </li>
       </xsl:for-each>
        </ol>
     </li>
   </xsl:for-each>
    </ol>

  </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

*Figure 16. Sample stylesheet: getstart.xsl*

## Document access definition files

The following sections contain document access definition (DAD) files that map XML data to DB2 database relational tables, using either XML column or XML collection access modes.


- "Sample DAD file: XML column" on page 278

- "Sample DAD file: XML collection: SQL mapping" shows a DAD file for an XML collection using SQL mapping.
- "Sample DAD file: XML: RDB_node mapping" on page 280 show a DAD for an XML collection that uses RDB_node mapping.

## Sample DAD file: XML column

This DAD file contains the mapping for an XML column, and it definines the table, side tables, and columns that are to contain the XML data.

```
<?xml version="1.0"?>
<!DOCTYPE Order SYSTEM  "dxx_install/samples/extenders/db2xml/dtd/dad.dtd">
<DAD>
   <dtdid> "dxx_install/samples/extenders/db2xml/dtd/getstart.dtd"
   </dtdid>
   <validation>YES</validation>

   <Xcolumn>
     <table name="order_side_tab">
         <column name="order_key"
            type="integer"
            path="/Order/@Key"
            multi_occurrence="NO"/>
         <column name="customer"
            type="varchar(50)"
            path="/Order/Customer/Name"
            multi_occurrence="NO"/>
     </table>
     <table name="part_side_tab">
         <column name="price"
            type="decimal(10,2)"
            path="/Order/Part/ExtendedPrice"
            multi_occurrence="YES"/>
     </table>
     <table name="ship_side_tab">
         <column name="date"
            type="DATE"
            path="/Order/Part/Shipment/ShipDate"
            multi_occurrence="YES"/>
     </table>

   </Xcolumn>

</DAD>
```

*Figure 17. Sample DAD file for an XML column: getstart_xcolumn.dad*

## Sample DAD file: XML collection: SQL mapping

This DAD file contains an SQL statement that specifies the DB2 database tables, columns, and conditions that are to contain the XML data.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "dxx_install/samples/extenders/db2xml/dtd/dad.dtd">
<DAD>
<validation>NO</validation>
<Xcollection>
<SQL_stmt>SELECT o.order_key, customer_name, customer_email, p.part_key, color,
   quantity, price, tax, ship_id, date, mode from order_tab o, part_tab p,
   table(select substr(char(timestamp(generate_unique())),16)
     as ship_id, date, mode, part_key from ship_tab) s
               p.price > 20000 and
               p.order_key = o.order_key and
               s.part_key = p.part_key
         ORDER BY order_key, part_key, ship_id</SQL_stmt>
<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE Order SYSTEM
"dxx_install/samples/extenders/db2xml/dtd/getstart.dtd
"</doctype>
```

*Figure 18. Sample DAD file for an XML collection using SQL mapping: order_sql.dad (Part 1 of 2)*

```
<root_node>
<element_node name="Order">
  <attribute_node name="key">
    <column name="order_key"/>
  </attribute_node>
  <element_node name="Customer">
    <element_node name="Name">
      <text_node><column name="customer_name"/></text_node>
    </element_node>
    <element_node name="Email">
      <text_node><column name="customer_email"/></text_node>
    </element_node>
  </element_node>
  <element_node name="Part">
    <attribute_node name="color">
      <column name="color"/>
    </attribute_node>
    <element_node name="key">
      <text_node><column name="part_key"/></text_node>
    </element_node>
    <element_node name="Quantity">
      <text_node><column name="quantity"/></text_node>
    </element_node>
    <element_node name="ExtendedPrice">
      <text_node><column name="price"/></text_node>
    </element_node>
    <element_node name="Tax">
      <text_node><column name="tax"/></text_node>
    </element_node>
    <element_node name="Shipment" multi_occurrence="YES">
      <element_node name="ShipDate">
        <text_node><column name="date"/></text_node>
      </element_node>
      <element_node name="ShipMode">
        <text_node><column name="mode"/></text_node>
      </element_node>
    </element_node>
  </element_node>
</element_node>
</root_node>
</Xcollection>
</DAD>
```

*Figure 18. Sample DAD file for an XML collection using SQL mapping: order_sql.dad (Part 2 of 2)*

## Sample DAD file: XML: RDB_node mapping

This DAD file uses <RDB_node> elements to define the DB2 database tables, columns, and conditions that are to contain XML data.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "SQLLIB/samples/extenders/db2xml/dtd/dad.dtd">
<DAD>
 <dtdid>E:\dtd\lineItem.dtd</dtdid>
 <validation>YES</validation>
 <Xcollection>
 <prolog>?xml version="1.0"?</prolog>
 <doctype>!DOCTYPE Order SYSTEM
           "SQLLIB/samples/extenders/db2xml/dtd/getstart.dtd"</doctype>
 <root_node>
 <element_node name="Order">
 <RDB_node>
 <table name="order_tab"/>
 <table name="part_tab"/>
```

```
<table name="ship_tab"/>
<condition>order_tab.order_key=part_tab.order_key AND
             part_tab.part_key=ship_tab.part_key </condition>
</RDB_node>
<attribute_node name="Key">
<RDB_node>
<table name="order_tab"/>
<column name="order_key"/>
</RDB_node>
</attribute_node>
<element_node name="Customer">

       <element_node name="Name">
           <text_node>
              <RDB_node>
                 <table name="order_tab"/>
                 <column name="customer_name"/>
               </RDB_node>
          </text_node>
       </element_node>
       <element_node name="Email">
          <text_node>
              <RDB_node>
                  <table name="order_tab"/>
                  <column name="customer_email"/>
              </RDB_node>
          </text_node>
       </element_node>
 </element_node>
     <element_node name="Part">
         <attribute_node name="Key">
               <RDB_node>
                   <table name="part_tab"/>
                   <column name="part_key"/>
               </RDB_node>
         </attribute_node>
         <element_node name="ExtendedPrice">
               <text_node>
                  <RDB_node>
                     <table name="part_tab"/>
                     <column name="price"/>
                     <condition>price > 2500.00</condition>
                  </RDB_node>
                </text_node>
          </element_node>
          <element_node name="Tax">
               <text_node>
                  <RDB_node>
                      <table name="part_tab"/>
                      <column name="tax"/>
                  </RDB_node>
               </text_node>
          </element_node>

          <element_node name="Quantity">
               <text_node>
                  <RDB_node>
                      <table name="part_tab"/>
                      <column name="qty"/>
                  </RDB_node>
               </text_node>
         </element_node>
         <element_node name="Shipment" multi_occurrence="YES">
              <element_node name="ShipDate">
                  <text_node>
                     <RDB_node>
                         <table name="ship_tab"/>
```

```
                        <column name="date"/>
                        <condition>date > '1966-01-01'</condition>
                    </RDB_node>
                </text_node>
            </element_node>
            <element_node name="ShipMode">
                <text_node>
                    <RDB_node>
                        <table name="ship_tab"/>
                        <column name="mode"/>
                    </RDB_node>
                </text_node>
            </element_node>
            <element_node name="Comment">
                <text_node>
                    <RDB_node>
                        <table name="ship_tab"/>
                        <column name="comment"/>
                    </RDB_node>
                </text_node>
            </element_node>
        </element_node> <!-- end of element Shipment-->
    </element_node> <!-- end of element Part -->
</element_node> <!-- end of element Order -->
</root_node>

</Xcollection>

</DAD>
```

# Appendix B. Code page considerations

XML documents and other related files must be encoded properly for each client or server that accesses the files. The XML Extender makes some assumptions when processing a file, you need to understand how it handles code page conversions. The primary considerations are:

- Ensuring that the actual code page of the client retrieving an XML document from DB2 database matches the encoding of the document.
- Ensuring that, when the document is processed by an XML parser, the encoding declaration of the XML document is also consistent with the document's actual encoding.

The following topics describe the issues for these considerations, how you can prepare for possible problems, and how the XML Extender and DB2 database support code pages when documents are passed from client to server, and to the database.

## Terminology for XML code pages

The following terms are used in the topics about XML code pages:

**document encoding**
The code page of an XML document.

**document encoding declaration**
The name of the code page specified in the XML declaration. For example, the following encoding declaration specifies `ibm-1047`:

```
<?xml version="1.0" encoding="ibm-1047"?>
```

**consistent document**
A document in which the code page matches the encoding declaration.

**inconsistent document**
A document in which the code page does not match the encoding declaration.

**DB2CODEPAGE registry (environment) variable**
Specifies the code page of the data presented to DB2 database from a database client application. DB2 database gets the client's code page from the client's operating system locale, unless this variable is set. To DB2, this value overrides the client operating system locale if it is set.

**client code page**
The application code page. If the DB2CODEPAGE variable is set, the client code page is the value of DB2CODEPAGE. Otherwise, the client code page is the client's operating system locale.

**server code page, or server operating system locale code page**
The operating system locale on which the DB2 database database is installed.

**database code page**
The encoding of the stored data, determined at database create time. If not explicitly specified with the USING CODESET clause, this value defaults to the operating system locale of the server.

© Copyright IBM Corp. 1999 - 2006

# DB2 and XML Extender code page assumptions

When DB2 database sends or receives an XML document, it does not check the encoding declaration. Rather, it checks the code page for the client to see if it matches the database code page. If they are different, DB2 database converts the data in the XML document to match the code page of:

- The database, when importing the document, or a document fragment, into a database table.
- The database, when decomposing a document into one or more database tables.
- The client, when exporting the document from the database and presenting the document to the client.
- The server, when processing a file with a UDF that returns data in a file on the server's file system.

## Assumptions for importing an XML document

When an XML document is imported into the database, it is generally imported as an XML document to be stored in an XML column, or for decomposition for an XML collection, where the element and attribute contents will be saved as DB2 database data. When a document is imported, DB2 database converts the document encoding to that of the database. DB2 database assumes that the document is in the code page specified in the "Source code page" column in the table below. Table 93 summarizes the conversions that DB2 database makes when importing an XML document.

*Table 93. Using UDFs and stored procedures when the XML file is imported into the database*

| Task | This is the source code page for conversion | This is the target code page for conversion | Comments |
|---|---|---|---|
| Inserting DTD file into DTD_REF table | Client code page | Database code page | |
| Enabling a column or a collection using stored procedures, or using administration commands that import DAD files | Client code page (the code page used to bind DXXADMIN during installation), | Database code page | |
| Using user-defined functions:<br>• XMLVarchar FromFile()<br>• XMLCLOB FromFile()<br>• Content(): retrieve from XMLFILE to a CLOB | Server code page | Database code page | The database code page is converted to the client code page when the data is presented to the client |
| Using stored procedures for decomposition | Client code page | Database code page | • Document to be decomposed is assumed to be in client code page. Data from decomposition is stored in tables in database code page |

## Assumptions for exporting an XML document

When an XML document is exported from the database, it is exported based on a client request to present one of the following objects:

- An XML document from an XML column
- The query results of XML documents in an XML column
- A composed XML document from an XML collection

When a document is exported, DB2 database converts the document encoding to that of the client or server, depending on where the request originated and where the data is to be presented. Table 94 summarizes the conversions that DB2 database makes when exporting an XML document.

*Table 94. Using UDFs and stored procedures when the XML file is exported from the database*

| Task | DB2 converts the ... | Comments |
|---|---|---|
| Using user-defined functions:<br>• XMLFileFromVarchar()<br>• XMLFileFromCLOB()<br>• Content(): retrieve from XMLVARCHAR to an external server file | Database code page to the server code page | |
| Using stored procedures to compose and store XML documents in a result table where it can be queried and exported. | Database code page to the client code page when the result set is presented to the client | • When composing documents, XML Extender copies the encoding declaration specified by the tag in the DAD, to the newly created document. It should match the client code page when presented. |

# Encoding declaration considerations for XML Extender

The *encoding declaration* specifies the code page of the XML document's encoding and appears on the XML declaration statement. When using XML Extender, it is important to ensure that the encoding of the document matches the code page of the client or the server, depending on where the file is located.

## Legal encoding declarations

You can use any encoding declaration in XML documents, within some guidelines. In this section, these guidelines are defined, along with the supported encoding declarations.

The recommended portable encodings for XML data are UTF-8 and UTF-16, according to the XML specification. Your application is interoperable between companies, if you use these encodings.

For all operating systems, the following encoding declarations are supported. The following list describes the meaning of each column:

- **Encoding** specifies the encoding string to be used in the XML declaration.
- **Category** shows the operating system on which DB2 database supports the given code page.

- **Code page** shows the IBM-defined code page associated with the given encoding

*Table 95. Encoding declarations supported by XML Extender (partial list)*

| Category | Encoding | Code page |
|---|---|---|
| Unicode | UTF-8 | 1208 |
| | UTF-16 | 1200 |
| ASCII | iso-8859-1 | 819 |
| | ibm-1252 | 1252 |
| | iso-8859-2 | 912 |
| | iso-8859-5 | 915 |
| | iso-8859-6 | 1089 |
| | iso-8859-7 | 813 |
| | iso-8859-8 | 916 |
| | iso-8859-9 | 920 |
| MBCS | gb2312 | 1386 |
| | ibm-932, shift_jis78 | 932 |
| | Shift_JIS | 943 |
| | IBM-eucCN | 1383 |
| | ibm-1388 | 1388 |
| | IBM-eucJP, EUC-JP | 954, 33722 |
| | ibm-930 | 930 |
| | ibm-939 | 939 |
| | ibm-1390 | 1390 |
| | ibm-1399 | 1399 |
| | ibm-5026 | 5026 |
| | ibm-5035 | 5035 |
| | euc-tw, IBM-eucTW | 964 |
| | ibm-937 | 937 |
| | euc-kr, IBM-eucKR | 970 |
| | big5 | 950 |

The encoding string must be compatible with the code page of the document's destination. If a document is being returned from a server to a client, then its encoding string must be compatible with the client's code page. See "Consistent encodings and encoding declarations" for the consequences of incompatible encodings.

## Consistent encodings and encoding declarations

When an XML document is processed or exchanged with another system, it is important that the encoding declaration corresponds to the actual encoding of the document. Ensuring that the encoding of a document is consistent with the client is important because XML tools, like parsers, generate an error for an entity that includes an encoding declaration other than that named in the declaration.

Figure 19 shows that clients have consistent code pages with the document encoding and declared encoding.

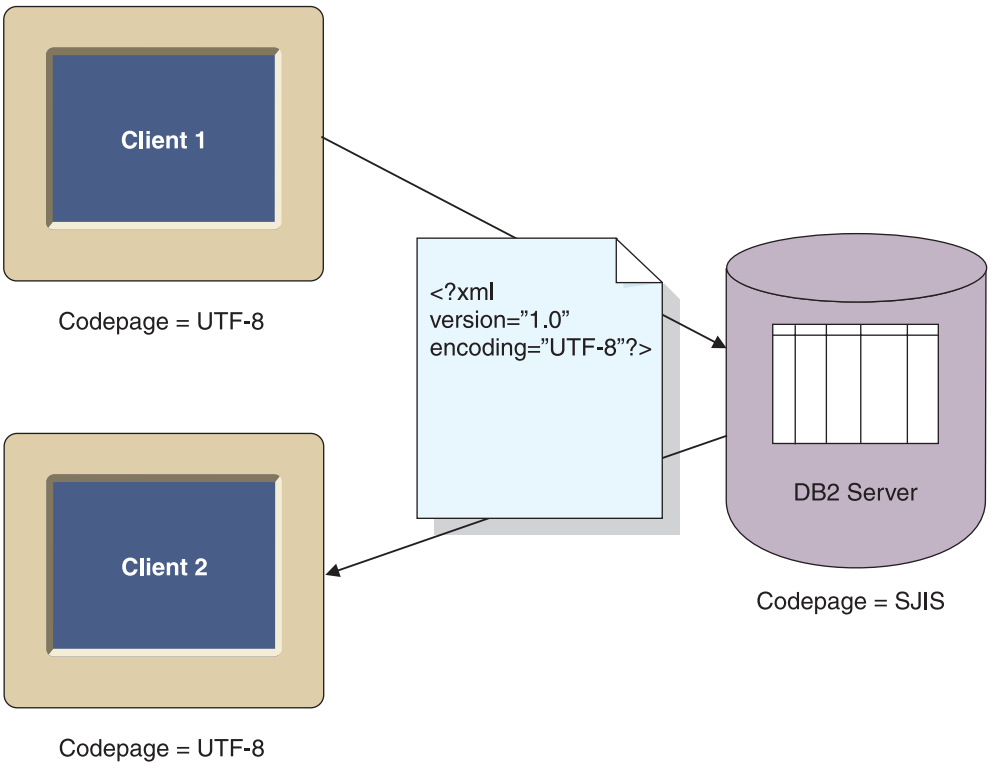



*Figure 19. Clients have matching code pages*

The consequences of having different code pages are the following possible situations:

- A conversion in which data is lost might occur, particularly if the source code page is Unicode and the target code page is not Unicode. Unicode contains the full set of characters. If a file is converted from UTF-8 to a code page that does not support all the characters used in the document, then data might be lost during the conversion.
- The declared encoding of the XML document might no longer be consistent with the actual document encoding, if the document is retrieved by a client with a different code page than the declared encoding of the document.

Figure 20 on page 288 shows an environment in which the code pages of the clients are inconsistent.

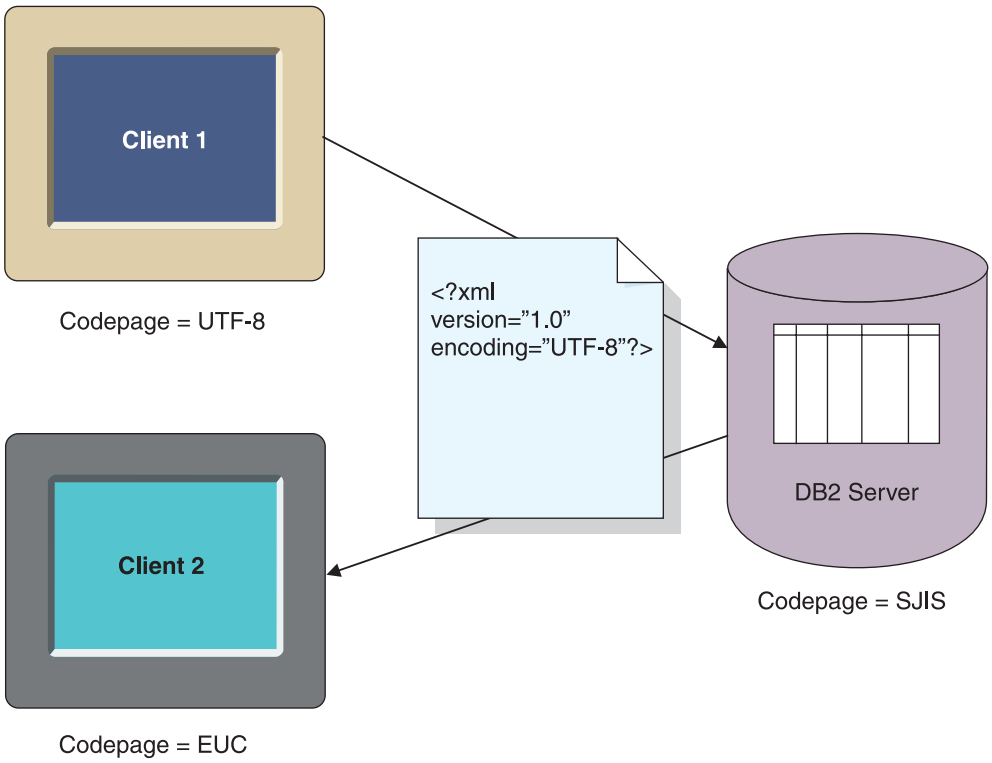



*Figure 20. Clients have mismatching code pages*

Client 2 receives the document in EUC, but the document will have an encoding declaration of UTF-8.

## Declaring an encoding

The default value of the encoding declaration is UTF-8, and the absence of an encoding declaration means the document is in UTF-8.

**To declare an encoding value:**

In the XML document declaration specify the encoding declaration with the name of the code page of the client. For example:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

# Conversion scenarios

The XML Extender processes XML documents when:

- Storing and retrieving XML column data, using the XML column storage and access method
- Composing and decomposing XML documents

Documents undergo code page conversion when passed from a client or server, to a database. Inconsistencies or damage of XML documents is most likely to occur during conversions from code pages of the client, server, and database. When choosing the encoding declaration of the document, as well as planning what clients and servers can import or export documents from the database, consider the conversions described in the above tables, and the scenarios described below.

The following scenarios describe common conversion scenarios that can occur:

**Scenario 1:** This scenario is a configuration with consistent encodings, no DB2 database conversion, and a document imported from the server. The document encoding declaration is UTF-8, the server is UTF-8, and the database is UTF-8. DB2 database does not need to convert the document because the server code page and database code page are identical. The encoding and declaration are consistent.

1. The document is imported into DB2 database using the XMLClobfromFile UDF.
2. The document is extracted to the server.

**Scenario 2:** This scenario is a configuration with consistent encodings, DB2 database conversion, and a document imported from server and exported to client. The document encoding and declaration is SJIS the client and server code pages are SJIS, and the database code pages are UTF-8.

1. The document is imported into DB2 database using XMLClobfromfile UDF from the server. DB2 converts the document from SJIS and stores it in UTF-8. The encoding declaration and encoding are inconsistent in the database.
2. A client using SJIS requests the document for presentation at the Web browser. DB2 database converts the document to SJIS, the client's code page. The document encoding and the declaration are now consistent at the client.

**Scenario 3:** This scenario is a configuration with inconsistent encodings, DB2 database conversion, a document imported from the server and exported to a client. The document encoding declaration is SJIS for the incoming document. The server code page is SJISibm-1047 and the client and database are UTF-8.

1. The document is imported into the database using a storage UDF. DB2 database converts the document to UTF-8 from SJIS. The encoding and declaration are inconsistent.
2. A client with a UTF-8 code page requests the document for presentation at a Web browser. DB2 does not convert because the client and the database code pages are the same. The document encoding and declaration are inconsistent because the declaration is SJIS and the encoding is UTF-8. The document cannot be processed by an XML parser or other XML processing tools.

**Scenario 4:** This scenario is a configuration with data loss, DB2 database conversion, and a document imported from a UTF-8 server. The document encoding declaration is UTF-8, the server is UTF-8 and the database is SJIS.

The document is imported into DB2 database using the XMLClobfromFile UDF. DB2 database converts the encoding to SJIS. When the document is imported, the document stored in the database might be corrupted because characters represented in UTF-8, might not have a representation in SJIS.

**Scenario 5:** This scenario is a configuration with a Windows NT limitation. On Windows NT, operating system locales cannot be set to UTF-8, however, DB2 database allows the client to set the code page to UTF-8 using `db2set DB2CODEPAGE=1208`. In this scenario, the client and server are on the same system. The client is UTF-8, but the server cannot be set to UTF-8; its code page is 1252. The document is encoded as 1252 and the encoding declaration is ibm-1252. The database code page is UTF-8.

1. The document is imported from the server by a storage UDF and converted from 1252 to 1208.
2. The document is exported from DB2 database using the Content() UDF that returns an XML file. DB2 database converts the document from UTF-8 to 1252, even though client might expect 1208 because the client is on the same system as the server and is set to 1208.

## Recommendations for preventing inconsistent XML documents

The above sections have discussed how an XML document can have an inconsistent encoding, that is, the encoding declaration conflicts with the document's encoding. Inconsistent encodings can cause the loss of data and or unusable XML documents.

Use one of the following recommendations for ensuring that the XML document encoding is consistent with the client code page, before handing the document to an XML processor, such as a parser:

- When exporting a document from the database using the XML Extender UDFs, try one of the following techniques (assuming XML Extender has exported the file, in the server code page, to the file system on the server):
  - Convert the document to the declared encoding code page
  - Override the declared encoding, if the tool has an override facility
  - Manually change the encoding declaration of the exported document to the document's actual encoding (that is, the server code page)
- When exporting a document from the database using the XML Extender stored procedures, try one of the following techniques (assuming the client is querying the result table, in which the composed document is stored):
  - Convert the document to the declared encoding code page
  - Override the declared encoding, if the tool has an override facility
  - Before querying the result table, have the client set the environment variable environment variable DB2CODEPAGE to force the client code page to a code page that is compatible with the encoding declaration of the XML document.
  - Manually change the encoding declaration of the exported document to the document's actual encoding (that is, the client code page)

**Limitation when using Unicode and a Windows NT client:** On Windows NT, the operating system locale cannot be set to UTF-8. Use the following guidelines when importing or exporting documents:

- When importing files and DTDs encoded in UTF-8, set the client code page to UTF-8, using:

  ```
  db2set DB2CODEPAGE=1208
  ```

  Use this technique when:
  - Inserting a DTD into the DB2XML.DTD_REF table
  - Enabling a column or collection
  - Decomposing stored procedures
- When using the Content() or XMLFromFile UDFs to import XML documents, documents must be encoded in the code page of the server's operating system locale, which cannot be UTF-8.
- When exporting an XML file from the database, set the client code page with the following command to have DB2 database encode the resulting data in UTF-8:

  ```
  db2set DB2CODEPAGE=1208
  ```

  Use this technique when:
  - Querying the result table after composition
  - Extracting data from an XML column using the extract UDFs

- When using the Content() or XMLxxxfromFile UDFs to export XML documents to files on the server file system, resulting documents are encoded in the code page of the server's operating system locale, which cannot be UTF-8.

# Appendix C. XML Extender limits

This topic describes the limits for:

- XML Extender objects
- Values returned by user-defined functions
- Stored procedures parameters
- Administration support table columns
- Composition and decomposition
- The following environment variables:
  - DB2DXX_MIN_TMPFILE_SIZE
  - DB2_DXX_PATHS_ALLOWED_READ
  - DB2_DXX_PATHS_ALLOWED_WRITE
  - DB2DXXTEMP

The following table describes the limits for XML Extender objects.

*Table 96. Limits for XML Extender objects*

| Object | Limit |
|---|---|
| Maximum number of rows in a table in a decomposition XML collection | 10240 rows from each decomposed XML document |
| Maximum bytes in XML file path name specified as a parameter value | 512 bytes |
| Length of the sql_stmt element in a DAD file for SQL composition | Windows and UNIX operating systems: 32,765 bytes. OS/390 and iSeries operating systems: 16,380 bytes. |
| Maximum number of columns for one table that are specified for one table in the DAD file for RDB_node decomposition | 500 columns (columns for a table) are specified by text_node and attribute_node elements in a DAD file. |
| Maximum number of columns in a table that can be mapped in a DAD Xcollection | Each table that is mapped in a Xcollection in the DAD file can have a maximum of 500 columns mapped to either text or attribute nodes. If you map to more than 500 columns in a table, DB2 issues message DXXQ039E, "Too many columns specified for a table in the DAD file." |

The following table describes the limits values returned by XML Extender user-defined functions.

*Table 97. Limits for user-defined function value*

| User-defined functions returned values | Limit |
|---|---|
| Maximum bytes returned by an extractCHAR UDF | 254 bytes |
| Maximum bytes returned by an extractCLOB UDF | 2 gigabytes |

© Copyright IBM Corp. 1999 - 2006

*Table 97. Limits for user-defined function value (continued)*

| User-defined functions returned values | Limit |
|---|---|
| Maximum bytes returned by an extractVARCHAR UDF | 4 kilobytes |

The following table describes the limits for parameters of XML Extender stored procedures.

*Table 98. Limits for stored procedure parameters*

| Stored procedure parameters | Limit |
|---|---|
| Maximum size of an XML document CLOB[1] | 1 MB |
| Maximum size of a Document Access Definition (DAD) CLOB[1] | 100 KB |
| Maximum size of *collectionName* | 30 bytes |
| Maximum size of *dbName* | 8 bytes |
| Maximum size of *tbName*[2] | 128 bytes |
| Maximum size of *colName* | 30 bytes |
| Maximum size of *tablespace* | 18 bytes |
| Maximum size of *defaultView* | 128 bytes |
| Maximum size of *rootID* | 128 bytes |
| Maximum size of *resultTabName* | 128 bytes |
| Maximum size of *resultColumn* | 128 bytes |
| Maximum size of *validColumn* | 128 bytes |
| Maximum size of *varchar_value* | 32672 bytes |

**Notes:**

1. This size can be changed for dxxGenXMLClob and dxxRetrieveXMLCLOB.
2. If the value of the *tbName* parameter is qualified by a schema name, the entire name (including the separator character) must be no longer than 128 bytes.

The following table describes the limits for the DB2XML.DTD_REF table.

*Table 99. XML Extender limits*

| DB2XML.DTD_REF table columns | Limit |
|---|---|
| Size of AUTHOR column | 128 bytes |
| Size of CREATOR column | 128 bytes |
| Size of UPDATOR column | 128 bytes |
| Size of DTDID column | 128 bytes |
| Size of CONTENT column | 100 KB |

Names can undergo expansion when DB2 database converts them from the client code page to the database code page. A name might fit within the size limit at the client, but exceed the limit when the stored procedure gets the converted name.

The following table describes limits for composition and decomposition.

*Table 100. Limits for XML Extender composition and decomposition*

| Object | Limit |
|---|---|
| Maximum number of rows inserted into a table in a decomposition XML collection | 10240 rows from each decomposed XML document |
| Maximum length of the name attribute in elements_node or attribute_node within a DAD | 63 bytes |
| Maximum bytes in XMLFile path name specified as a parameter value | 512 bytes |
| Maximum number of columns in a key | 16 columns |

### The DB2DXX_MIN_TMPFILE_SIZE environment variable

XML Extender places large documents in temporary files to avoid using too much memory during processing. On systems with large amounts of physical memory, it is possible to avoid moving documents to temporary files, reducing the amount of input/output activity. The environment variable DB2DXX_MIN_TMPFILE_SIZE instructs XML Extender to use memory buffers, rather than temporary files, for processing documents smaller than the specified value. The variable is applicable only on the server, not on a client. If multiple physical nodes participate in a multi-node partition, you can set the variable differently for each node, accurately reflecting the amount of memory that is installed on each system. If you do not set this variable, documents that exceed 128 KB are automatically placed into temporary files during processing, while documents smaller than 128 KB are processed in memory.

### DB2_DXX_PATHS_ALLOWED_READ and DB2_DXX_PATHS_ALLOWED_WRITE environment variables

These DB2 variables restrict the XML Extender UDFs read and write access to the file system. You must set the two new DB2 registry variables before you use XML Extender UDFs that read or write to the server file system. The new variables are described in the following table:

*Table 101. DB2 variables to set before using XML Extender UDFs to read or write to the file system*

| Variable | Values |
|---|---|
| DB2_DXX_PATHS_ALLOWED_READ<br><br>Specifies which locations on the server file system XML Extender UDFs are authorized to read. When the registry is not set (contains a null value), XML Extender UDFs cannot read from the server file system. A value of ANY means that UDFs can read from any location on the server file system. | Default = null<br><br>Values: ANY, or a valid path |

*Table 101. DB2 variables to set before using XML Extender UDFs to read or write to the file system (continued)*

| Variable | Values |
|---|---|
| DB2_DXX_PATHS_ALLOWED_WRITE<br><br>Specifies which locations on the server file system XML Extender UDFs are authorized to write. When the registry is not set (contains a null value), XML Extender UDFs cannot write to the server file system. A value of ANY means that UDFs can write to any location on the server file system. | Default = null<br><br>Values: ANY, or a valid path |

Before you start working with DB2 XML Extender, you might want to set the location where temporary files will be stored. The DB2DXXTEMP environment variable controls the location of XML Extender temporary files. If the variable is not set, the value of the system variable TMP determines the location of the temporary files. To set the value of DB2DXXTEMP on Windows XP:

1. Make sure you are logged on with the user ID that you use with DB2.
2. Click **Start—>Control Panel.**
3. Double–click the **System** icon.
4. Go to the **Advanced** page.
5. Click **Environment Variables**.
6. Click **New** in the **System variables** section. Enter `DB2DXXTEMP` for the variable name and enter a value for the variable such as `C:\temp`.
7. Click **OK**, close all windows, and restart the system.

# XML Extender glossary

**absolute location path.** The full path name of an object. The absolute path name begins at the highest level, or ″root″ element, which is identified by the forward slash (/) or back slash (\) character.

**access and storage method.** Associates XML documents to a DB2 database through two major access and storage methods: XML columns and XML collections. See also *XML column* and *XML collection*.

**access function.** A user-provided function that converts the data type of text stored in a column to a type that can be processed by Net Search Extender.

**administration.** The task of preparing text documents for searching, maintaining indexes, and getting status information.

**administrative support tables.** Tables used by a DB2 database extender to process user requests on XML objects. Some administrative support tables identify user tables and columns that are enabled for XML Extender. Other administrative support tables contain attribute information about objects in enabled columns. Synonymous with metadata table.

**analyze.** To calculate numeric values for the features of an image and add the values to a QBIC catalog.

**API.** See *application programming interface*.

**application programming interface (API).**

1. A functional interface supplied by the operating system or by a separately orderable licensed program. An API allows an application program that is written in a high-level language to use specific data or functions of the operating system or the licensed programs.
2. The DB2 database extenders provide APIs for requesting user-defined functions, administrative operations, display operations, and video scene change detection. The DB2 Net Search Extender provides APIs for requesting user-defined functions, administrative operations, and information retrieval services. In DB2, a function within the interface. For example, the get error message API.

**attribute.** See *XML attribute*.

**attribute_node.** A representation of an attribute of an element.

**binary large object (BLOB).** A binary string whose length can be up to 2 GB. Image, audio, and video objects are stored in a DB2 database as BLOBs.

**Boolean search.** A search in which one or more search terms are combined using Boolean operators.

**browse.** To view text displayed on a computer monitor.

**browser.** A Net Search Extender function that enables you to display text on a computer monitor.See *Web browser*.

**B-tree indexing.** The native index scheme provided by the DB2 database engine. It builds index entries in the B-tree structure. Supports DB2 base data types.

**cast function.** A function that is used to convert instances of a (source) data type into instances of a different (target) data type. In general, a cast function has the name of the target data type. It has one single argument whose type is the source data type; its return type is the target data type.

**CCSID.** Coded Character Set Identifier.

**character large object (CLOB).** A character string of single-byte characters, where the string can be up to 2 GB. CLOBs have an associated code page. Text objects that contain single-byte characters are stored in a DB2 database as CLOBs.

**CLOB.** Character large object.

© Copyright IBM Corp. 1999 - 2006

**code page.** An assignment of graphic characters and control function meanings to all code points. For example, assignment of characters and meanings to 256 code points for an 8-bit code.

**column data.** The data stored inside of a DB2 database column. The type of data can be any data type supported by DB2.

**compose.** To generate XML documents from relational data in an XML collection.

**condition.** A specification of either the criteria for selecting XML data or the way to join the XML collection tables.

**DAD.** See *Document access definition*.

**data interchange.** The sharing of data between applications. XML supports data interchange without needing to go through the process of first transforming data from a proprietary format.

**data source.** A local or remote relational or nonrelational data manager that is capable of supporting data access via an ODBC driver that supports the ODBC APIs.

**data stream.** Information returned by an API function, comprising text (at least one paragraph) containing the term searched for, and information for highlighting the found term in that text.

**data type.** An attribute of columns and literals.

**database partition.** A part of the database that consists of its own user data, indexes, configuration files, and transaction logs. Sometimes called a node or database node.

**database partition server.** Manages a *database partition*. A database partition server is composed of a database manager and the collection of data and system resources that it manages. Typically, one database partition server is assigned to each system.

**DBCLOB.** Double-byte character large object.

**DBCS.** Double-byte character support.

**decompose.** Separates XML documents into a collection of relational tables in an XML collection.

**default casting function.** Casts the SQL base type to a UDT.

**default view.** A representation of data in which an XML table and all of its related side tables are joined.

**disable.** To restore a database or column to its condition before it was enabled for XML Extender by removing the items created during the enabling process.

**distinct type.** See *user-defined type*.

**document.** See *text document*.

**Document Access Definition (DAD).** An XML document used to define the indexing scheme for an XML column or mapping scheme of an XML collection. It can be used to enable an XML Extender column or an XML collection.

**Document type definition (DTD).** A set of declarations for XML elements and attributes. The DTD defines what elements are used in the XML document, in what order they can be used, and which elements can contain other elements. You can associate a DTD with a document access definition (DAD) file to validate XML documents.

**double-byte character large object (DBCLOB).** A character string of double-byte characters, or a combination of single-byte and double-byte characters, where the string can be up to 2 GB. DBCLOBs have an associated code page. Text objects that include double-byte characters are stored in a DB2 database as DBCLOBs.

**DTD.** See *Document type definition*.

**DTD reference table (DTD_REF table).** A table that contains DTDs used to validate XML documents. Users can insert their own DTDs into the DTD_REF table. This table is created when a database is enabled for XML.

**DTD_REF table.** DTD reference table.

**DTD repository.** A DB2 database table, called DTD_REF, where each row of the table represents a DTD with additional metadata information.

**EDI.** Electronic Data Interchange.

**Electronic Data Interchange (EDI).** A standard for electronic data interchange for business-to-business (B2B) applications.

**element.** See *XML element*.

**element_node.** A representation of an element. An element_node can be the root element or a child element.

**embedded SQL.** SQL statements coded within an application program. See *static SQL*.

**enable.** To prepare a database or a column for use by XML Extender.

**escape character.** A character indicating that the subsequent character is not to be interpreted as a *masking character*.

**Extensible Stylesheet Language (XSL).** A language used to express stylesheets. XSL consists of two parts: a language for transforming XML documents, and an XML vocabulary for specifying formatting semantics.

**Extensible Stylesheet Language Transformation (XSLT).** XSLT is a utility with which you apply XSL stylesheets to XML documents, transforming the source XML documents to HTML, text, or other XML document types.

**external file.** A file that exists in a file system external to DB2.

**file reference variable.** A programming variable that is useful for moving a LOB to and from a file on a client workstation.

**foreign key.** A key that is part of the definition of a referential constraint and that consists of one or more columns of a dependent table.

**gigabyte (GB).** One billion ($10^9$) bytes. When referring to memory capacity, 1 073 741 824 bytes.

**host variable.** A variable in an application program that can be referred to in embedded SQL statements. Host variables are the primary mechanism for transmitting data between a database and application program work areas.

**Java Database Connectivity (JDBC).** An application programming interface (API) that has the same characteristics as Open Database Connectivity (ODBC) but is specifically designed for use by Java database applications. Also, for databases that do not have a JDBC driver, JDBC includes a JDBC to ODBC bridge, which is a mechanism for converting JDBC to ODBC; JDBC presents the JDBC API to Java database applications and converts this to ODBC. JDBC was developed by Sun Microsystems, Inc. and various partners and vendors.

**JDBC.** Java Database Connectivity.

**join.** A relational operation that allows for retrieval of data from two or more tables based on matching column values.

**joined view.** A DB2 database view created by the ″CREATE VIEW″ statement which join one more tables together.

**kilobyte (KB).** One thousand ($10^3$) bytes. When referring to memory capacity, 1024 bytes.

**large object (LOB).** A sequence of bytes, where the length can be up to 2 GB. A LOB can be of three types: *binary large object* (BLOB), *character large object (CLOB)*, or *double-byte character large object (DBCLOB)*.

**LOB.** Large object.

**LOB locator.** A small (4-byte) value stored in a host variable that can be used in a program to refer to a much larger LOB in a DB2 database. Using a LOB locator, a user can manipulate the LOB as if it was stored in a regular host variable, and without the need to transport the LOB between the application on the client system and the database server.

**location path.** Location path is a sequence of XML tags that identify an XML element or attribute. The location path identifies the structure of the XML document, indicating the context for the element or attribute. A single slash (/) path indicates that the context is the whole document. The location path is used in the extracting UDFs to identify

the elements and attributes to be extracted. The location path is also used in the DAD file to specify the mapping between an XML element, or attribute, and a DB2 database column when defining the indexing scheme for XML column.

**locator.** A pointer which can be used to locate an object. In DB2, the large object block (LOB) locator is the data type which locates LOBs.

**logical node.** A node on a processor when more than one node is assigned to that processor.

**mapping scheme.** A definition of how XML data is represented in a relational database. The mapping scheme is specified in the DAD. The XML Extender provides two types of mapping schemes: *SQL mapping* and *relational database node (RDB_node) mapping*.

**megabyte (MB).** One million ($10^6$) bytes. When referring to memory capacity, 1 048 576 bytes.

**metadata table.** See *administrative support table*.

**multiple occurrence.** An indication of whether an element can occur more than once in a document. Multiple occurrence is specified in the DAD.

**object.** In object-oriented programming, an abstraction consisting of data and the operations associated with that data.

**ODBC.** Open Database Connectivity.

**Open Database Connectivity.** A standard application programming interface (API) for accessing data in both relational and nonrelational database management systems. Using this API, database applications can access data stored in database management systems on a variety of computers even if each database management system uses a different data storage format and programming interface. ODBC is based on the call level interface (CLI) specification of the X/Open SQL Access Group and was developed by Digital Equipment Corporation (DEC), Lotus, Microsoft, and Sybase. Contrast with *Java Database Connectivity*.

**overloaded function.** A function name for which multiple function instances exist.

**path expression.** See *location path*.

**predicate.** An element of a search condition that expresses or implies a comparison operation.

**primary key.** A unique key that is part of the definition of a table. A primary key is the default parent key of a referential constraint definition.

**procedure.** See *stored procedure*.

**RDB_node.** Relational database node.

**RDB_node mapping.** The location of the content of an XML element, or the value of an XML attribute, which are defined by the RDB_node. The XML Extender uses this mapping to determine where to store or retrieve the XML data.

**relational database node (RDB_node).** A node that contains one or more element definitions for tables, optional columns, and optional conditions. The tables and columns are used to define how the XML data is stored in the database. The condition specifies either the criteria for selecting XML data or the way to join the XML collection tables.

**result set.** A set of rows returned by a stored procedure.

**result table.** A table which contains rows as the result of an SQL query or an execution of a stored procedure.

**root element.** The top element of an XML document.

**root ID.** A unique identifier that associates all side tables with the application table.

**SBCS.** Single-byte character support.

**scalar function.** An SQL operation that produces a single value from another value and is expressed as a function name, followed by a list of arguments enclosed in parentheses.

**schema.** A collection of database objects such as tables, views, indexes, or triggers. It provides a logical classification of database objects.

**search argument.** The conditions specified when making a search, consisting of one or several search terms, and search parameters.

**section search.** Provides the text search within a section which can be defined by the application. To support the structural text search, a section can be defined by the XPath's abbreviated location path.

**side table.** Additional tables created by the XML Extender to improve performance when searching elements or attributes in an XML column.

**simple location path.** A sequence of element type names connected by a single slash (/).

**SQL mapping.** A definition of the relationship of the content of an XML element or value of an XML attribute with relational data, using one or more SQL statements and the XSLT data model. The XML Extender uses the definition to determine where to store or retrieve the XML data. SQL mapping is defined with the SQL_stmt element in the DAD.

**static SQL.** SQL statements that are embedded within a program, and are prepared during the program preparation process before the program is executed. After being prepared, a static SQL statement does not change, although values of host variables specified by the statement might change.

**stored procedure.** A block of procedural constructs and embedded SQL statements that is stored in a database and can be called by name. Stored procedures allow an application program to be run in two parts. One part runs on the client and the other part runs on the server. This allows one call to produce several accesses to the database.

**structural text index.** To index text keys based on the tree structure of the XML document, using the IBM DB2 Net Search Extender.

**subquery.** A full SELECT statement that is used within a search condition of an SQL statement.

**table space.** An abstraction of a collection of containers into which database objects are stored. A table space provides a level of indirection between a database and the tables stored within the database. A table space:

- Has space on media storage devices assigned to it.
- Has tables created within it. These tables will consume space in the containers that belong to the table space. The data, index, long field, and LOB portions of a table can be stored in the same table space, or can be individually broken out into separate table spaces.

**terabyte.** A trillion ($10^{12}$) bytes. Ten to the twelfth power bytes. When referring to memory capacity, 1 099 511 627 776 bytes.

**text_node.** A representation of the CDATA text of an element.

**top element_node.** A representation of the root element of the XML document in the DAD.

**tracing.** The action of storing information in a file that can later be used in finding the cause of an error.

**trigger.** The definition of a set of actions to be taken when a table is changed. Triggers can be used to perform actions such as validating input data, automatically generating a value for a newly inserted row, reading from other tables for cross-referencing purposes, or writing to other tables for auditing purposes. Triggers are often used for integrity checking or to enforce business rules.

**UDF.** See *user-defined function*.

**UDT.** See *user-defined type*.

**uniform resource locator (URL).** An address that names an HTTP server and optionally a directory and file name, for example: http://www.ibm.com/software /data/db2/extenders.

**UNION.** An SQL operation that combines the results of two select statements. UNION is often used to merge lists of values that are obtained from several tables.

**URL.** Uniform resource locator.

**user-defined distinct type (UDT).** A data type created by a user of DB2, in contrast to a data type provided by DB2 database such as LONG VARCHAR.

**user-defined function (UDF).** A function that is defined by a user to DB2. Once defined, the function can be used in SQL queries. and video objects. For example, UDFs can be created to get the compression format of a video or return the sampling rate of an audio. This provides a way of defining the behavior of objects of a particular type.

**user-defined function (UDF).** An SQL function created by a user of DB2, in contrast to an SQL function provided by DB2. Net Search Extender provides search functions, such as CONTAINS, in the form of UDFs.

**user-defined type (UDT).** A data type that is defined by a user to DB2. UDTs are used to differentiate one LOB from another. For example, one UDT can be created for image objects and another for audio objects. Though stored as BLOBs, the image and audio objects are treated as types distinct from BLOBs and distinct from each other.

**user-defined function (UDF).** A function that is defined to the database management system and can be referenced thereafter in SQL queries. It can be one of the following functions:

- An external function, in which the body of the function is written in a programming language whose arguments are scalar values, and a scalar result is produced for each invocation.
- A sourced function, implemented by another built-in or user-defined function that is already known to the DBMS. This function can be either a scalar function or column (aggregating) function, and returns a single value from a set of values (for example, MAX or AVG).

**user-defined type (UDT).** A data type that is not native to the database manager and was created by a user. See *distinct type*.

**user table.** A table that is created for and used by an application.

**validation.** The process of using a DTD to ensure that the XML document is valid and to allow structured searches on XML data. The DTD is stored in the DTD repository.

**valid document.** An XML document that has an associated DTD. To be valid, the XML document cannot violate the syntactic rules specified in its DTD.

**video.** Pertaining to the portion of recorded information that can be seen.

**video clip.** A section of filmed or videotaped material.

**video index.** A file that the Video Extender uses to find a specific *shot* or frame in a video clip.

**Web browser.** A client program that initiates requests to a Web server and displays the information that the server returns.

**well-formed document.** A document that matches the production labeled document, meets all the well-formedness constraints given in the w3.org specification, and each of the parsed entities which is referenced directly or indirectly within the document is well-formed.

**wildcard character.** See *masking character*.

**XML.** eXtensible Markup Language.

**XML attribute.** Any attribute specified by the ATTLIST under the XML element in the DTD or XML Schema. The XML Extender uses the location path to identify an attribute.

**XML collection.** A collection of relation tables which presents the data to compose XML documents, or to be decomposed from XML documents.

**XML column.** A column in the application table that has been enabled for XML Extender.

**XML element.** Any XML tag or ELEMENT as specified in the XML DTD or XML schema. The XML Extender uses the location path to identify an element.

**XML object.** Equivalent to an XML document.

**XPath.** (XML Path Language) A language for addressing parts of an XML document. XML Path Language is designed to be used by XSLT. Every location path can be expressed using the syntax defined for XPath.

**XML table.** An application table which includes one or more XML Extender columns.

**XML tag.** Any valid XML markup language tag, mainly the XML element. The terms tag and element are used interchangeably.

**XML UDF.** A DB2 database user-defined function provided by the XML Extender.

**XML UDT.** A DB2 database user-defined type provided by the XML Extender.

**XPath.** A language for addressing parts of an XML document.

**XPath data model.** The tree structure used to model and navigate an XML document using nodes.

**XSL.** Extensible Stylesheet Language.

**XSLT.** Extensible Stylesheet Language Transformation.

# Notices

IBM® may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:**
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 1999 - 2006

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs, in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _*enter the year or years*_. All rights reserved.

## Trademarks

Company, product, or service names identified in the documents of the DB2® Version 9 documentation library may be trademarks or service marks of International Business Machines Corporation or other companies. Information on the trademarks of IBM Corporation in the United States, other countries, or both is located at http://www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 documentation library:

Microsoft®, Windows®, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel®, Itanium®, Pentium®, and Xeon® are trademarks of Intel Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

Linux® is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

# Index


## A

access and storage method
  choosing an 36
  planning 36
  XML collections 40, 41, 163
  XML columns 40, 41, 163
access method
  choosing an 36
  introduction 5
  planning an 36
  XML collections 87
  XML column 71
adding
  nodes 65
administration
  dxxadm command 121
  support tables
    DTD_REF 251
    XML_USAGE 251
  tools 36
administration stored procedures
  dxxDisableCollection() 186
  dxxDisableColumn() 184
  dxxDisableDB() 182
  dxxEnableCollection() 185
  dxxEnableColumn() 183
  dxxEnableDB() 182
administration wizard
  Enable a Column window 53
  logging in 36
  specifying address 36
  specifying JDBC driver 36
  specifying user ID and password 36
administrative support tables
  DTD_REF 251
  XML_USAGE 251
attribute_node 41, 49, 103, 163

## B

B-tree indexing 73
binding
  stored procedures 179

## C

casting function
  retrieval 76, 135
  storage 74, 132
  update 80, 152
CCSID (coded character set identifier)
  declare in USS 88, 92, 283
client code page 283
code pages
  client 283
  configuring locale settings 283
  consistent encoding in USS 283
  consistent encodings and declarations 283
code pages *(continued)*
  conversion
    scenarios 283
  data loss 283
  database 283
  DB2 assumptions 283
  DB2CODEPAGE registry variable 283
  declaring an encoding 283
  document encoding consistency 283
  encoding declaration 283
  exporting documents 283
  importing documents 283
  legal encoding declarations 283
  line endings 283
  preventing inconsistent documents 283
  server 283
  supported encoding declarations 283
  terminology 283
  UDFs and stored procedures 283
  Windows NT UTF-8 limitation 283
  XML Extender assumptions 283
column data
  available UDTs 38
column type, for decomposition 48
column types
  decomposition 103
command options
  disable_collection 127
  disable_column 125
  disable_db 123
  enable_collection 126
  enable_column 124
  enable_db 122
complexType element 113
composing XML documents 17
composite key
  for decomposition 47
  XML collections 47
composite keys
  for decomposition 103
  XML collections 103
composition
  dxxGenXML() 88
  dxxRetrieveXML() 88
  overriding the DAD file 171
  stored procedures
    dxxGenXML() 17, 187, 193
    dxxmqGen() 223
    dxxmqRetrieve() 228
    dxxRetrieveXML() 190, 195
  XML collection 88
conditions
  optional 47
  RDB_node mapping 47, 103
  SQL mapping 43, 46, 99, 101
consistent documents 283
Content() function
  for retrieval 76
  retrieval functions using 135
Content() function *(continued)*
  XMLFile to a CLOB 135
conversions
  code pages 283
creating
  nodes 65
  XML tables 51

## D

DAD
  node definitions
    RDB_node 47
DAD (Document Access Definition)
  file
    attribute_node 163
    bind step for USS encodings 283
    CCSIDs in USS 88, 92, 283
    creating for XML collections 61
    declaring the encoding 283
    DTD for the 166
    editing for XML collections 61
    element_node 103, 163
    examples 275
    for XML columns 161, 163
    introduction 5
    node definitions 163
    overriding 171
    RDB_node 103
    root element_node 103
    root_node 163
    samples 275
    size limit 163, 293
    text_node 163
DAD file
  attribute_node 41
  element_node 41, 47
  for XML columns 39, 40
  node definitions
    attribute_node 41
    element_node 41
    root_node 41
    text_node 41
  planning for the 39, 40
    XML collections 39
    XML column 39
  RDB_node 47
  root element_node 47
  root_node 41
  size limit 39, 40
  text_node 41
data loss, inconsistent encodings 283
database
  relational 42
databases
  code page 283
  enabling for XML 50
  relational 99
DB2CODEPAGE
  registry variable 283
DB2XML 251


© Copyright IBM Corp. 1999 - 2006

DB2XML *(continued)*
DTD_REF table schema 251
schema for stored procedures 87
schema for UDFs and UDTs 132
XML_USAGE table schema 251
decomposing an XML collection
collection table limit 293
composite key 103
DB2 table sizes 92
dxxInsertXML() 92
dxxShredXML() 92
of XML collections 92
specifying the column type for 103
specifying the orderBy attribute 103
specifying the primary key for 103
stored procedures
dxxInsertXML() 199
dxxmqInsert() 237
dxxmqInsertAll 241
dxxmqInsertAllCLOB() 242
dxxmqInsertCLOB() 239
dxxmqShred() 232
dxxmqShredAll() 234
dxxShredXML() 197
using RDB_node mapping 65
decomposition
composite key 47
DB2 table sizes 49
specifying the column type for 48
specifying the orderBy attribute 48
specifying the primary key for 47
deleting
nodes 65
XML collections 95
disable_collection command 127
disable_column command 125
disable_db command 123
disabling
administration command 121
databases for XML, stored procedure 182
disable_collection command 127
disable_column command 125
disable_db command 123
stored procedure 182, 184, 186
XML collections 110
stored procedure 186
XML columns
stored procedure 184
document encoding declaration 283
document structure, maintaining 71
document type definition 52
DTD
availability 4
for getting started lessons 17
for the DAD 166
planning 17
publication 4
repository
DTD_REF 5, 251
storing in 52
using multiple 40, 50
DTD_REF table 52
column limits 293
inserting a DTD 52
schema 251
DTDID 251
DVALIDATE 156
DXX_SEQNO for multiple occurrence 56
dxxadm command
disable_collection command 127
disable_column command 125
disable_db command 123
enable_collection command 126
enable_column command 124
enable_db command 122
introduction to 121
syntax 121
dxxDisableCollection() stored procedure 186
dxxDisableColumn() stored procedure 184
dxxDisableDB() stored procedure 182
dxxEnableCollection() stored procedure 185
dxxEnableColumn() stored procedure 183
dxxEnableDB() stored procedure 182
dxxGenXML() 17
dxxGenXML() stored procedure 88, 187, 193
dxxInsertXML() stored procedure 92, 199
dxxmqGen() stored procedure 223
dxxmqInsert() stored procedure 237
dxxmqInsertAll() stored procedure 241
dxxmqInsertAllCLOB() stored procedure 242
dxxmqInsertCLOB() stored procedure 239
dxxmqRetrieve() stored procedure 228
dxxmqShred() stored procedure 232
dxxRetrieveXML() stored procedure 88, 190, 195
DXXROOT_ID 73
dxxShredXML() stored procedure 92, 197
dxxtrc command 253, 254
dynamically overriding the DAD file, composition 171

## E

element_node 41, 48, 103, 163
Enable a Column window 53
enable_collection keyword 126
enable_column keyword 124
enable_db keyword
creating XML_USAGE table 251
option 122
enabling
XML collections 109
encoding
CCSID declarations in USS 88, 92, 283
XML documents 283
environment variables
CLASSPATH 36
existing DB2 data 87
Extensible Markup Language (XML)
in XML documents 3
extractChar() function 145
extractChars() function 145
extractCLOB() function 147
extractCLOBs() function 147
extractDate() function 148
extractDates() function 148
extractDouble() function 142
extractDoubles() function 142
extracting functions
description 131
extractChar() 145
extractChars() 145
extractCLOB() 147
extractCLOBs() 147
extractDate() 148
extractDates() 148
extractDouble() 142
extractDoubles() 142
extractReal() 143
extractReals() 143
extractSmallint() 141
extractSmallints() 141
extractTime() 149
extractTimes() 149
extractTimestamp() 150
extractTimestamps() 150
extractVarchar() 146
extractVarchars() 146
introduction to 140
table of 76
extractReal() function 143
extractReals() function 143
extractSmallint() function 141
extractSmallints() function 141
extractTime() function 149
extractTimes() function 149
extractTimestamp() function 150
extractTimestamps() function 150
extractVarchar() function 146
extractVarchars() function 146

## F

FROM clause 46
SQL mapping 101
function path
adding DB2XML schema 132
functions
casting 74, 76, 80
Content(): from XMLFILE to CLOB 135
extractChar() 145
extractChars() 145
extractCLOB() 147
extractCLOBs() 147
extractDate() 148
extractDates() 148
extractDouble() 142
extractDoubles() 142
extracting 140
extractReal() 143
extractReals() 143
extractSmallint() 141
extractSmallints() 141
extractTime() 149
extractTimes() 149
extractTimestamp() 150
extractTimestamps() 150
extractVarchar() 146
extractVarchars() 146

functions *(continued)*
generate_unique 131
limitations when invoking from JDBC 84
limits 293
MQReadAllXML 205
MQReadAllXMLCLOB 208
MQReadXML 204
MQReadXMLCLOB 207
MQReceiveAllXML 212
MQReceiveXML 210
MQReceiveXMLCLOB 215
MQSENDXML 217
MQSENDXMLFILE 219
MQSendXMLFILECLOB 220
MQSeries 202
retrieval
description 131
from external storage to memory pointer 135
from internal storage to external server file 135
introduction 135
XML data 76
storage 74, 131, 132
update 80, 131, 152
XML columns 131
XMLCLOBFromFile() 132
XMLFile to a CLOB 135
XMLFileFromCLOB() 132, 133
XMLFileFromVarchar() 132, 134
XMLVarcharFromFile() 132, 135

## H

highlighting conventions vii

## I

importing
DTD 52
include files
for stored procedures 179
inconsistent
document 283
indexing 73
side tables 57, 73
structural-text 73
XML columns 73
XML documents 73
Information Center, including this book
in vii
installing
the 35

## J

Java database connectivity (JDBC)
limitations when invoking UDFs 84
JDBC (Java database connectivity)
limitations when invoking UDFs 84
JDBC address, for wizard 36
JDBC driver, for wizard 36
join conditions
RDB_node mapping 47, 103
SQL mapping 46, 101

## L

limits
stored procedure parameters 88, 251
XML Extender 293
line
endings, code page considerations 283
locales
settings 283
location path
introduction 106
syntax 107
XPath 5
XSL 5
logging
in, for wizard 36

## M

maintaining
document structure 71
mapping scheme
determining RDB_node mapping 44, 99
determining SQL mapping 43, 99
figure of DAD file 36
figure of DAD for the 37
for XML collections 36, 37
for XML columns 36, 37
FROM clause 46, 101
introduction 87
ORDER BY clause 46, 101
RDB_node mapping requirements 47, 103
requirements 45
SELECT clause 45, 101
SQL mapping requirements 45, 101
SQL mapping scheme 45, 99
SQL_stmt 42, 99
WHERE clause 46, 101
messages
XML 260
migrating
XML Extender to Version 8 35
MQPublishXML function 202
MQRcvAllXML function 214
MQRcvXMLCLOB function 216
MQReadAllXML function 205
MQReadAllXMLCLOB function 208
MQReadXML function 204
MQReadXMLCLOB function 207
MQReceiveAllXML function 212
MQReceiveXML function 210
MQReceiveXMLCLOB function 215
MQSENDXML function 217
MQSENDXMLFILE function 219
MQSendXMLFILECLOB function 220
MQSeries
functions 202
multiple DTDs
XML collections 40
XML columns 50
multiple occurrence
affecting table size 49, 92
deleting elements and attributes 95
DXX_SEQNO 56
multiple occurrence *(continued)*
one column per side table 56
order of elements and attributes 92
orderBy attribute 47, 103
preserving the order of elements and attributes 95
recomposing documents with 47, 103
searching elements and attributes 82
updating collections 95
updating elements and attributes 80, 95, 152
updating XML documents 80, 152
multiple-occurrence attribute 17

## N

nodes
adding 65
attribute_node 41, 163
creating 65
DAD file configuration 17, 58, 61, 65
deleting 65
element_node 41, 163
RDB_node 47, 103
removing 65
root_node 41, 163
text_node 41, 163
nonamespacelocation 50

## O

operating systems
supported by DB2 3
Operations Navigator
starting the trace 253
stopping the trace 254
ORDER BY clause 46
SQL mapping 101
orderBy attribute
for decomposition 48, 103
for multiple occurrence 47, 103
XML collections 48, 103
overloaded function
Content() 135
overrideType
No override 171
SQL override 171
XML override 171
overriding
DAD file 171

## P

parameter markers in functions 84
performance
indexing side tables 73
searching XML documents 73
stopping the trace 254
planning
a mapping scheme 42
access methods 36
choosing to validate XML data 40
DAD 163
determining column UDT 38
DTD 17
for the DAD 39, 40

planning *(continued)*
for XML collections 40
for XML columns 38, 39
how to search XML column data 39
indexing XML columns 73
mapping schemes 99
mapping XML document and database 17
side tables 56
storage methods 36
the XML collections mapping scheme 42
validating with multiple DTDs 40, 50
XML collections 163
XML collections mapping scheme 99
primary key for decomposition 47
primary keys
decomposition 103
side tables 73
problem determination 253
processing instructions 106, 163

# R

RDB_node mapping
composite key for decomposition 47
conditions 47
decomposition requirements 47
determining for XML collections 44
requirements 47, 103
specifying column type for decomposition 48
registry variables
DB2CODEPAGE 283
removing
nodes 65
repository, DTD 52
retrieval functions
Content() 135
description of 131
from external storage to memory pointer 135
from internal storage to external server file 135
introduction to 135
XMLFile to a CLOB 135
retrieving data
attribute values 76
return codes
stored procedures 255
UDF 254
ROOT ID
indexing considerations 73
specifying 53
root_node 41, 163

# S

samples
creating
XML 17
document access definition (DAD) files 275
getstart.xml sample XML document 275
schema names
for stored procedures 87
schemabindings 50
schemas
attributes 114
DB2XML 50, 132
declaring data types in 114
declaring elements in 114
DTD_REF table 52, 251
validating using 50
XML_USAGE table 251
searching
XML documents
by structure 82
using DB2 Text Extender 82
SELECT clause 45, 101
server code page 283
side tables
indexing 57, 73
planning 56
searching 82
specifying ROOT ID 53
updating 80
size limits
stored procedures 88, 251
XML Extender 293
software requirements
XML Extender 35
SQL mapping 58
creating a DAD file 17
determining for XML collections 43, 99
FROM clause 46
ORDER BY clause 46
requirements 45, 101
SELECT clause 45
SQL mapping scheme 45
WHERE clause 46
SQL override 171
SQL_stmt
FROM clause 46, 101
ORDER_BY clause 46, 101
SELECT clause 45, 101
WHERE clause 46, 101
starting
XML Extender 35
storage
functions
description 131
introduction 132
storage UDF table 74
XMLCLOBFromFile() 132
XMLFileFromCLOB() 132, 133
XMLFileFromVarchar() 132, 134
XMLVarcharFromFile() 132, 135
methods
choosing 36
introduction 5
planning 36
XML collections 87
XML column 71
storage UDFs 74, 80
stored procedures
administration
dxxDisableCollection() 186
dxxDisableColumn() 184
dxxDisableDB() 182
stored procedures *(continued)*
administration *(continued)*
dxxEnableCollection() 185
dxxEnableColumn() 183
dxxEnableDB() 182
XML Extender, list 181
binding 179
calling
XML Extender 179
CLOBs 180
code page considerations 283
composition
dxxGenXML() 187, 193
dxxmqGen() 223
dxxmqRetrieve() 228
dxxRetrieveXML() 190, 195
XML Extenders 186
decomposition
dxxInsertXML() 199
dxxmqInsert() 237
dxxmqInsertAll 241
dxxmqInsertAllCLOB() 242
dxxmqInsertCLOB() 239
dxxmqShred() 232
dxxmqShredAll() 234
dxxShredXML() 197
XML Extenders 197
dxxDisableCollection() 186
dxxDisableColumn() 184
dxxDisableDB() 182
dxxEnableCollection() 185
dxxEnableColumn() 183
dxxEnableDB() 182
dxxGenXML() 17, 88, 187, 193
dxxInsertXML() 92, 199
dxxmqGen() 223
dxxmqInsert() 237
dxxmqInsertAll() 241
dxxmqInsertAllCLOB() 242
dxxmqInsertCLOB() 239
dxxmqRetrieve() 228
dxxmqShred() 232
dxxRetrieveXML() 88, 190, 195
dxxShredXML() 92, 197
include files 179
initializing
DXXGPREP 179
return codes 255
XML Extender 179
storing the DTD 52
storing XML data 74
structure
DTD 17
hierarchical 17
mapping 17
relational tables 17
XML document 17
stylesheets 106, 163
SVALIDATE 156
syntax
disable_collection command 127
disable_column command 125
disable_db command 123
dxxadm 121
enable_collection command 126
enable_column command 124
enable_db command 122

syntax *(continued)*
extractChar() function 145
extractChars() function 145
extractCLOB() function 147
extractCLOBs() function 147
extractDate() function 148
extractDates() function 148
extractDouble() function 142
extractDoubles() function 142
extractInteger() function 140
extractIntegers() function 140
extractReal() function 143
extractReals() function 143
extractSmallint() function 141
extractSmallints() function 141
extractTime() function 149
extractTimes() function 149
extractTimestamp() function 150
extractTimestamps() function 150
extractVarchar() function 146
extractVarchars() function 146
location path 107
Update() function 152
XMLCLOBFromFile() function 132
XMLFile to a CLOB Content() function 135
XMLFileFromCLOB() function 132, 133
XMLFileFromVarchar() function 132, 134
XMLVarcharFromFile() function 135

# T

tables 92
tables sizes, for decomposition 49
text_node 41, 49, 103, 163
traces
starting 253
stopping 254
transfer of documents between client and server, considerations 283
transforming XML to HTML
XSLTransformToCLOB 246
XSLTransformToFile 247
troubleshooting
stored procedure return codes 255
strategies 253
UDF return codes 254

# U

UDFs (user-defined functions)
code page considerations 283
DVALIDATE() 156
extractChar() 145
extractChars() 145
extractCLOB() 147
extractCLOBs() 147
extractDate() 148
extractDates() 148
extractDouble() 142
extractDoubles() 142
extracting functions 140
extractReal() 143
extractReals() 143
UDFs (user-defined functions) *(continued)*
extractSmallint() 141
extractSmallints() 141
extractTime() 149
extractTimes() 149
extractTimestamp() 150
extractTimestamps() 150
extractVarchar() 146
extractVarchars() 146
for XML columns 131
from external storage to memory pointer 135
from internal storage to external server file 135
retrieval functions 135
return codes 254
searching with 82
storage 80
SVALIDATE() 156
Update() 80, 152
XMLCLOBFromFile() 132
XMLFile to a CLOB 135
XMLFileFromCLOB() 132, 133
XMLFileFromVarchar() 132, 134
XMLVarcharFromFile() 132, 135
UDTs
summary table of 38
XMLCLOB 38
XMLFILE 38
XMLVARCHAR 38
Update() function
document replacement behavior 152
introduction 152
XML 80, 131
updates
side tables 80
XML collection 95
XML column data
attributes 80
description 80
entire document 80
multiple occurrence 152
specific elements 80
XML document replacement by Update() UDF 152
user IDs
Administration wizard 36
user-defined functions (UDFs)
for XML columns 131
searching with 82
Update() 80, 152
user-defined types (UDTs)
for XML columns 71
XML 129
XMLCLOB 71
XMLFILE 71
XMLVARCHAR 71

# V

validate XML data
considerations 40
deciding to 40
DTD requirements 40
validating
performance impact 40
using schemas 50
validating *(continued)*
XML DTDs 52

# W

WHERE clause 46
requirements for SQL mapping 101
Windows
UTF-8 limitation, code pages Windows NT 283

# X

XML
data, storing 74
override 171
repository 36
tables, creating 51
XML collections
composition 88
creating the DAD (command line) 61
DAD file, planning for 39
decomposing using RDB_node mapping 65
decomposition 92
definition 5
determining a mapping scheme 99
determining a mapping scheme for 42
disabling 110
DTD for validation 52
editing the DAD (command line) 61
enabling 109
introduction 87
mapping scheme 42
mapping schemes 43, 99
RDB_node mapping 44, 99
scenarios 38
SQL mapping 43, 99
storage and access methods 5, 87
validation 52
when to use 38
XML columns
creating a DAD file for 161
DAD file, planning for 39
defining and enabling 72
definition 5
determining column UDT 38
elements and attributes to be searched 39
enabling 53
figure of side tables 56
indexing 73
introduction 71
location path 106
maintaining document structure 71
planning 38
retrieving data
attribute values 76
element contents 76
entire document 76
retrieving XML data 76
sample DAD file 275
scenarios 38
storage and access methods 5, 71
the DAD for 39

XML columns *(continued)*
UDFs 131
updating XML data
attributes 80
entire document 80
specific elements 80
when to use 38
with side tables 73
XML documents
B-tree indexing 73
code page assumptions 283
code page consistency 283
code page conversion, exporting 283
code page conversion, importing 283
composing 17, 88
decomposition 92
deleting 84
encoding declarations 283
indexing 73
introduction 3
legal encoding declarations 283
mapping to tables 17
searching
direct query on side tables 82
document structure 82
from a joined view 82
multiple occurrence 82
structural text 82
with extracting UDFs 82
stored in DB2 3
supported encoding declarations 283
XML DTD repository
description 5
DTD Reference Table (DTD_REF) 5
XML Extender
available operating systems 3
functions 131
introduction 3
stored procedures 179
XML Path Language 5
XML schemas
advantages 113
example 115
validating 156
XML Toolkit for OS/390 and z/OS 6
XML_USAGE table 251
XMLClobFromFile() function 132
XMLFile to a CLOB function 135
XMLFileFromCLOB() function 132, 133
XMLFileFromVarchar() function 132, 134
XMLVarcharFromFile() function 132, 135
XPath 5
XSLT 43, 99
using 17
XSLTransformTOClob() 246
XSLTransformToFile 247

# Contacting IBM

To contact IBM in your country or region, check the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide

To learn more about DB2 products, go to http://www.ibm.com/software/data/db2/.

© Copyright IBM Corp. 1999 - 2006

Printed in USA

SC18-9750-00

Spine information:

IBM DB2 **DB2 Version 9** **XML Extender Administration and Programming** IBM