SYBASE®

XML Services

Adaptive Server® Enterprise

15.5

DOCUMENT ID: DC30020-01-1550-01

LAST REVISED: October 2009

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book

Audience

Customers who want to:

- Store complete XML documents in a SQL database
- Test and extract data from XML documents in a SQL database
- Store data extracted from XML documents
- Generate XML documents from SQL data
- Process SQL data as XML

How to use this book

The information in this book is organized as follows:

- Chapter 1, “Introduction to XML Services,” introduces XML in the database and the new XML capabilities of the Sybase XML Services.
- Chapter 2, “XML Query Functions,” addresses processing and querying XML documents within SQL statements. You can apply these query functions both to stored XML documents (that is, normal user documents), and to SQLX-XML documents generated by the for xml clause or forxmlj function, or by any similar tools that provide an XML view of SQL data. For detailed information about these functions see Chapter 4, “XML Mapping Functions.”
- Chapter 3, “XML Language and XML Query Language,” describes the XML document and query language features supported by XML query functions, including specification of the XPath language subset supported.
- Chapter 4, “XML Mapping Functions,” describes functions that map between SQL data and XML documents in the SQLX-XML format.
- Chapter 5, “XML Mappings,”describes the SQLX-XML format of the XML documents that the XML mapping functions support.
- Chapter 6, “Support for I18N,”describes the extension of XML Services to support non-ASCII data.
- Appendix A, “Setting Up XML Services,” includes guidelines for installing both the native, C++ processor and the Java processor included with Adaptive Server version 12.5 and later.

- Appendix B, “The *sample_docs* Example Table,” is a description of the sample_docs table used in the function examples.
- Appendix C, “XML Services and External File System Access”, contains examples of how to use the XML features with XFS.
- Appendix D, “The Java-Based XQL Processor,” describes using XQL to select raw data from Adaptive Server, using XQL, and displaying the results as an XML document.
- Appendix E, “Migrating Between the Java-Based XQL Processor and the Native XML Processor,” describes the different functions and methods used to implement query languages and return documents in parsed form, and how to switch from one to another.
- Appendix F, a sample application for xmltable().

Related documents

The Adaptive Server® Enterprise documentation set consists of the following:

- The release bulletin for your platform – contains last-minute information that was too late to be included in the books.

  A more recent version of the release bulletin may be available on the World Wide Web. To check for critical product or document information that was added after the release of the product CD, use the Sybase Technical Library.
- The *Installation Guide* for your platform – describes installation, upgrade, and configuration procedures for all Adaptive Server and related Sybase products.
- *What's New in Adaptive Server Enterprise?* – describes the new features in Adaptive Server version 15.0, the system changes added to support those features, and changes that may affect your existing applications.
- *ASE Replicator User's Guide* – describes how to use the Adaptive Server Replicator feature of Adaptive Server to implement basic replication from a primary server to one or more remote Adaptive Servers.
- *Component Integration Services User's Guide* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- The *Configuration Guide* for your platform – provides instructions for performing specific configuration tasks for Adaptive Server.
- *Enhanced Full-Text Search Specialty Data Store User's Guide* – describes how to use the Full-Text Search feature with Verity to search Adaptive Server Enterprise data.

- *Glossary* – defines technical terms used in the Adaptive Server documentation.
- *Historical Server User's Guide* – describes how to use Historical Server to obtain performance information for SQL Server® and Adaptive Server.
- *Java in Adaptive Server Enterprise* – describes how to install and use Java classes as datatypes, functions, and stored procedures in the Adaptive Server database.
- *Job Scheduler User's Guide* – provides instructions on how to install and configure, and create and schedule jobs on a local or remote Adaptive Server using the command line or a graphical user interface (GUI).
- *Messaging Service User's Guide* – describes how to useReal Time Messaging Services to integrate TIBCO Java Message Service and IBM WebSphere MQ messaging services with all Adaptive Server database applications.
- *Monitor Client Library Programmer's Guide* – describes how to write Monitor Client Library applications that access Adaptive Server performance data.
- *Monitor Server User's Guide* – describes how to use Monitor Server to obtain performance statistics from SQL Server and Adaptive Server.
- *Performance and Tuning Series* – a series of books that explain how to tune Adaptive Server for maximum performance:
    - *Basics* – the basics for understanding and investigating performance questions in Adaptive Server.
    - *Locking and Concurrency Control* – describes how the various locking schemas can be used for improving performance in Adaptive Server, and how to select indexes to minimize concurrency.
    - *Query Processing and Abstract Plans* – describes how the optimizer processes queries and how abstract plans can be used to change some of the optimizer plans.
    - *Physical Database Tuning* – describes how to manage physical data placement, space allocated for data, and the temporary databases.
    - *Monitoring Adaptive Server with sp_sysmon* – describes how to monitor Adaptive Server's performance with sp_sysmon.
    - *Improving Performance with Statistical Analysis* – describes how Adaptive Server stores and displays statistics, and how to use the set statistics command to analyze server statistics.

    - *Using the Monitoring Tables* – describes how to query Adaptive Server’s monitoring tables for statistical and diagnostic information.

- *Quick Reference Guide* – provides a comprehensive listing of the names and syntax for commands, functions, system procedures, extended system procedures, data types, and utilities in a pocket-sized book (regular size when viewed in PDF format).

- *Reference Manual* – is a series of four books that contains the following detailed Transact-SQL® information:

    - *Building Blocks* – Transact-SQL datatypes, functions, global variables, expressions, identifiers and wildcards, and reserved words.

    - *Commands* – Transact-SQL commands.

    - *Procedures* – Transact-SQL system procedures, catalog stored procedures, system extended stored procedures, and dbcc stored procedures.

    - *Tables* – Transact-SQL system tables and dbcc tables.

- *System Administration Guide* –

    - *Volume 1* – provides an introduction to the basics of system administration, including a description of configuration parameters, resource issues, character sets, sort orders, and diagnosing system problems. The second part of this book is an in-depth description of security administration.

    - *Volume 2* – includes instructions and guidelines for managing physical resources, mirroring devices, configuring memory and data caches, managing multiprocessor servers and user databases, mounting and unmounting databases, creating and using segments, using the reorg command, and checking database consistency. The second half of this book describes how to back up and restore system and user databases.

- *System Tables Diagram* – illustrates system tables and their entity relationships in a poster format. Full-size available only in print version; a compact version is available in PDF format.

- *Transact-SQL User’s Guide* – documents Transact-SQL, the Sybase enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the pubs2 and pubs3 sample databases.

- *Troubleshooting Series* (for release 15.0) –
  - *Troubleshooting: Error Messages Advanced Resolutions* – contains troubleshooting procedures for problems that you may encounter when using Sybase® Adaptive Server® Enterprise. The problems addressed here are those which the Sybase Technical Support staff hear about most often
  - *Troubleshooting and Error Messages Guide* – contains detailed instructions on how to resolve the most frequently occurring Adaptive Server error messages. Most of the messages presented here contain error numbers (from the master..sysmessages table), but some error messages do not have error numbers, and occur only in Adaptive Server's error log.
- *User Guide for Encrypted Columns* – describes how configure and use encrypted columns with Adaptive Server
- *Using Adaptive Server Distributed Transaction Management Features* – explains how to configure, use, and troubleshoot Adaptive Server DTM features in distributed transaction processing environments.
- *Using Sybase Failover in a High Availability System* – provides instructions for using Sybase Failover to configure an Adaptive Server as a companion server in a high availability system.
- *Unified Agent and Agent Management Console* – describes the Unified Agent, which provides runtime services to manage, monitor and control distributed Sybase resources.
- *Utility Guide* – documents the Adaptive Server utility programs, such as isql and bcp, which are executed at the operating system level.
- *Web Services User's Guide* – explains how to configure, use, and troubleshoot Web Services for Adaptive Server.
- *XA Interface Integration Guide for CICS, Encina, and TUXEDO* – provides instructions for using the Sybase DTM XA interface with X/Open XA transaction managers.
- *XML Services in Adaptive Server Enterprise* – describes the Sybase native XML processor and the Sybase Java-based XML support, introduces XML in the database, and documents the query and mapping functions that comprise XML Services.

Other sources of information

Use the Sybase Getting Started CD, the SyBooks™ CD, and the Sybase® Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.
- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ Finding the latest information on product certifications

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click Certification Report.
3. In the Certification Report filter select a product, platform, and timeframe and then click Go.
4. Click a Certification Report title to display the report.

❖ Finding the latest information on component certifications

1. Point your Web browser to Availability and Certification Reports at http://certification.sybase.com/.
2. Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.

3. Select Search to display the availability and certification report for the selection.

❖ Creating a personalized view of the Sybase Web site (including support pages)

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ Finding the latest information on EBFs and software maintenance

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.
4. Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the “Technical Support Contact” role to your MySybase profile.
5. Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

The following sections describe conventions used in this manual.

SQL is a free-form language. There are no rules about the number of words you can put on a line or where you must break a line. However, for readability, all examples and most syntax statements in this manual are formatted so that each clause of a statement begins on a new line. Clauses that have more than one part extend to additional lines, which are indented. Complex commands are formatted using modified Backus Naur Form (BNF) notation.

Table 1 shows the conventions for syntax statements that appear in this manual:

Table 1: Font and syntax conventions for this manual

Element	Example
Command names,procedure names, utility names, and other keywords display in sans serif font.	select sp_configure
Database names and datatypes are in sans serif font.	master database
Book names, file names, variables, and path names are in italics.	*System Administration Guide* *sql.ini* file *column_name* *$SYBASE/ASE* directory

Variables—or words that stand for values that you fill in—when they are part of a query or statement, are in italics in Courier font.	`select` *`column_name`*
	    `from` *`table_name`*
	    `where` *`search_conditions`*

Type parentheses as part of the command.	compute *row_aggregate* (*column_name*)
Double colon, equals sign indicates that the syntax is written in BNF notation. Do not type this symbol. Indicates “is defined as”.	`::=`
Curly braces mean that you must choose at least one of the enclosed options. Do not type the braces.	`{cash, check, credit}`
Brackets mean that to choose one or more of the enclosed options is optional. Do not type the brackets.	`[cash | check | credit]`
The comma means you may choose as many of the options shown as you want. Separate your choices with commas as part of the command.	`cash, check, credit`
The pipe or vertical bar (|) means you may select only one of the options shown.	`cash | check | credit`

An ellipsis (...) means that you can *repeat* the last unit as many times as you like.	`buy thing = price [cash | check | credit]`
	`[, thing = price [cash | check | credit]]...`
	You must buy at least one thing and give its price. You may choose a method of payment: one of the items enclosed in square brackets. You may also choose to buy additional things: as many of them as you like. For each thing you buy, give its name, its price, and (optionally) a method of payment.

- Syntax statements (displaying the syntax and all options for a command) appear as follows:

  sp_dropdevice [*device_name*]

  For a command with more options:

  select *column_name*
  from *table_name*
  where *search_conditions*

In syntax statements, keywords (commands) are in normal font and identifiers are in lowercase. Italic font shows user-supplied words.

- Examples showing the use of Transact-SQL commands are printed like this:

```
select * from publishers
```

- Examples of output from the computer appear as follows:

```
pub_id    pub_name                 city          state
-------   ---------------------    -----------   -----
0736      New Age Books            Boston        MA
0877      Binnet & Hardley         Washington    DC
1389      Algodata Infosystems     Berkeley      CA

(3 rows affected)
```

In this manual, most of the examples are in lowercase. However, you can disregard case when typing Transact-SQL keywords. For example, SELECT, Select, and select are the same.

Adaptive Server sensitivity to the case of database objects, such as table names, depends on the sort order installed on Adaptive Server. You can change case sensitivity for single-byte character sets by reconfiguring the Adaptive Server sort order. For more information, see the *System Administration Guide*.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Adaptive Server HTML documentation has been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1 Introduction to XML Services

This chapter describes the XML Services feature of Adaptive Server Enterprise.

XML capabilities

XML Services provides the following capabilities:

- Generating XML: A for xml clause in select commands, which returns the result set as an XML document in the standard SQLX format.
- Storing XML:
    - Support for XML documents stored as either character data in char, varchar, text, unichar, univarchar, or unitext columns, or as parsed XML.
    - xmlparse, which parses and indexes and XML document and generates a parsed and indexed representation for storage.
    - xmlvalidate, which validates the XML document against DTD or XML schema definitions.
- Querying and shredding XML: xmltest and xmlextract, which query and extract data from XML documents.
- I18N support: Support for Unicode and non-ASCII server character sets in XML documents, including support for generating, storing, querying and extracting XML documents containing non-ASCII data.

Overview

Like HTML (Hypertext Markup Language, XML is a markup language and a subset of SGML (Standardized General Markup Language). XML, however, is more complete and disciplined, and it allows you to define your own application-oriented markup tags. These properties make XML particularly suitable for data interchange.

You can generate XML-formatted documents from data stored in Adaptive Server and, conversely, store XML documents and data extracted from them in Adaptive Server. You can also use Adaptive Server to search XML documents stored on the Web.

XML is a markup language and subset of SGML, created to provide functionality beyond that of HTML for Web publishing and distributed document processing.

XML in the database

- XML documents possess a strict phrase structure that makes it easy to find and access data. For instance, all elements must have both an opening tag and a corresponding closing tag:

  <p> A paragraph.*</p>*.

- XML lets you develop and use tags that distinguish different types of data, such as customer numbers or item numbers.
- XML lets you create an application-specific document type, making it possible to distinguish one kind of document from another.
- XML documents allow different displays of the XML data. XML documents, like HTML documents, contain only markup and content; they do not contain formatting instructions. Formatting instructions are normally provided on the client.

XML is less complex than SGML, but more complex and flexible than HTML. Although XML and HTML can usually be read by the same browsers and processors, certain XML characteristics enable it to share documents more efficiently that HTML.

You can store XML documents in Adaptive Server as:

- Character data in columns of datatypes char, varchar, unichar, univarchar, text, unitext, java.lang.String, or image.

- Parsed XML in an image column

A sample XML document

This sample Order document is designed for a purchase order application. Customers submit orders, which are identified by a date and a customer ID. Each o rder item has an item ID, an item name, a quantity, and a unit designation.

It might display on your screen like this:

ORDER

Date: July 4, 2003

Customer ID: 123

Customer Name: Acme Alpha

Items:

Item ID	Item Name	Quantity
987	Coupler	5
654	Connector	3 dozen
579	Clasp	1

The following is one representation of this data in XML:

```
<?xml version="1.0"?>
    <Order>
 <Date>2003/07/04</Date>
 <CustomerId>123</CustomerId>
 <CustomerName>Acme Alpha</CustomerName>
   <Item>
 <ItemId> 987</ItemId>
 <ItemName>Coupler</ItemName>
 <Quantity>5</Quantity>
 </Item>
<Item>
 <ItemId>654</ItemId>
 <ItemName>Connector</ItemName>
 <Quantity unit="12">3</Quantity>
 </Item>
<Item>
 <ItemId>579</ItemId>
 <ItemName>Clasp</ItemName>
 <Quantity>1</Quantity>
 </Item>
</Order>
```

The XML document has two unique characteristics:

- The XML document does not indicate type, style, or color for specifying item display.
- The markup tags are strictly nested. Each opening tag (<*tag*>) has a corresponding closing tag (</*tag*>).

The XML document for the order data consists of four main elements:

- The XML declaration, <?xml version="1.0"?>, identifying "Order" as an XML document.

  The XML declaration for each document specifies the character encoding (character set), either explicitly or implicitly. XML represents documents as character data.To explicitly specify the character set, include it in the XML declaration. For example:

  ```
  <?xml version="1.0" encoding="ISO-8859-1">
  ```

  If you do not include the character set in the XML declaration, XML in Adaptive Server uses the default character set, UTF8.

  Note When the default character sets of the client and server differ, Adaptive Server bypasses normal character-set translations. The declared character set continues to match the actual character set. See "XML Support for I18N" on page 113.

- User-created element tags, such as <Order>…</Order>, <CustomerId>…</CustomerId>, <Item>….</Item>.
- Text data, such as "Acme Alpha," "Coupler," and "579."
- Attributes embedded in element tags, such as <Quantity unit = "12">. This embedding allows you to customize elements.

If your document contains these components, and the element tags are strictly nested, it is called a **well-formed XML documen**t. In the example above, element tags describe the data they contain, and the document contains no formatting instructions.

Here is another example of an XML document:

```
<?xml version="1.0"?>
 <Info>
```

```
   <OneTag>1999/07/04</OneTag>
   <AnotherTag>123</AnotherTag>
   <LastTag>Acme Alpha</LastTag>
  <Thing>
      <ThingId> 987</ThingId>
      <ThingName>Coupler</ThingName>
      <Amount>5</Amount>
      <Thing/>
  <Thing>
   <ThingId>654</ThingId>
   <ThingName>Connecter</ThingName>
</Thing>
   <Thing>
      <ThingId>579</ThingId>
      <ThingName>Clasp</ThingName>
      <Amount>1</Amount>
   </Thing>
</Info>
```

This example, called “Info,” is also a well-formed XML document, and has the same structure and data as the XML Order document. However, it would not be recognized by a processor designed for Order documents because the document type definition (DTD) that Info uses is different from that of the Order document. For more information about DTDs, see “XML document types” on page 7.

HTML display of Order data

Consider a purchase order application. Customers submit orders, which are identified by a Date and the CustomerID, and which list one or more items, each of which has an ItemID, ItemName, Quantity, and units.

The data for such an order might be displayed on a screen as follows:

ORDER

Date: July 4, 1999

Customer ID: 123

Customer Name: Acme Alpha

Items:

Item ID	Item Name	Quantity
987	Coupler	5
654	Connector	3 dozen

Item ID	Item Name	Quantity
579	Clasp	1

This data indicates that the customer named Acme Alpha, whose Customer ID is 123, submitted an order on 1999/07/04 for couplers, connectors, and clasps.

The HTML text for this display of order data is as follows:

```
<html>
<body>
<p>ORDER
<p>Date:&nbsp;&nbsp;July 4, 1999
<p>Customer ID:&nbsp;&nbsp;123
<p>Customer Name:&nbsp;&nbsp;Acme Alpha
<p>Items:</p>
<table bgcolor=white align=left border=”3”
   cellpadding=3>
<tr><td><B>Item ID&nbsp;&nbsp;&nbsp;</B></tr>
   <td><B>Item Name&nbsp;&nbsp;&nbsp;</B></tr>
   <td><B>Quantity&nbsp;&nbsp;&nbsp;</B>
   </td></td></tr>
<tr><td>987</td>
   <td>Coupler</td>
   <td>5</td></tr>
<tr><td>654</td>
   <td>Connector</td>
   <td>3 dozen</td></tr>
<tr><td>579</td>
   <td>Clasp</td>
   <td>1</td></tr>
</table>
</body>
</html>
```

This HTML text has certain limitations:

- It contains both data and formatting specifications.
  - The data is the Customer ID, and the various Customer names, item names, and quantities.
  - The formatting specifications indicate type style (*<b>....</b>*), color (*bcolor=white*), and layout (*<table>....</table>*, as well as the supplementary field names, such as *Customer Name*, and so on.
- The structure of HTML documents is not well suited for extracting data.

Some elements, such as tables, require strictly bracketed opening and closing tags, but other elements, such as paragraph tags ("*<p>*"), have optional closing tags.

Some elements, such as paragraph tags ("*<p>*") are used for many sorts of data, so it is difficult to distinguish between 123, a Customer ID, and 123, an Item ID, without inferring the context from surrounding field names.

This merging of data and formatting, and the lack of strict phrase structure, makes it difficult to adapt HTML documents to different presentation styles, and makes it difficult to use HTML documents for data interchange and storage. XML is similar to HTML, but includes restrictions and extensions that address these drawbacks.

XML document types

A **document type definition** (DTD) defines the structure of a class of XML documents, making it possible to distinguish between classes. A DTD is a list of element and attribute definitions unique to a class. Once you have set up a DTD, you can reference that DTD in another document, or embed it in the current XML document.

The DTD for XML Order documents, discussed in "A sample XML document" on page 3 looks like this:

```
<!ELEMENT Order (Date, CustomerId, CustomerName, Item+)>
 <!ELEMENT Date (#PCDATA)>
 <!ELEMENT CustomerId (#PCDATA)>
 <!ELEMENT CustomerName (#PCDATA)>
 <!ELEMENT Item (ItemId, ItemName, Quantity)>
 <!ELEMENT ItemId (#PCDATA)>
 <!ELEMENT ItemName (#PCDATA)>
 <!ELEMENT Quantity (#PCDATA)>
 <!ATTLIST Quantity units CDATA #IMPLIED>
```

Line by line, this DTD specifies that:

- An order must consist of a date, a customer ID, a customer name, and one or more items. The plus sign, "+", indicates one or more items. Items signaled by a plus sign are required. A question mark in the same place indicates an optional element. An asterisk in the element indicates that an element can occur zero or more times. (For example, if the word "Item*" in the first line above were starred, there could be no items in the order, or any number of items.)

- Elements defined by “(#PCDATA)” are character text.
- The “<ATTLIST…>” definition in the last line specifies that quantity elements have a “units” attribute; “#IMPLIED”, at the end of the last line, indicates that the “units” attribute is optional.

The character text of XML documents is not constrained. For example, there is no way to specify that the text of a quantity element should be numeric, and thus the following display of data would be valid:

```
<Quantity unit=”Baker’s dozen”>three</Quantity>
<Quantity unit=”six packs”>plenty</Quantity>
```

Restrictions on the text of elements must be handled by the applications that process XML data.

An XML’s DTD must follow the <?xml version="1.0"?> instruction. You can either include the DTD within your XML document, or you can reference an external DTD.

- To reference a DTD externally, use something similar to:

```
<?xml version="1.0"?>
 <!DOCTYPE Order SYSTEM  "Order.dtd”>
 <Order>
…
 </Order>
```

- Here’s how an embedded DTD might look:

```
<?xml version="1.0"?>
 <!DOCTYPE Order [
 <!ELEMENT Order (Date, CustomerId, CustomerName,
 Item+)>
 <!ELEMENT Date (#PCDATA)
 <!ELEMENT CustomerId (#PCDATA)>
 <!ELEMENT CustomerName (#PCDATA)>
 <!ELEMENT Item (ItemId, ItemName, Quantity)>
 <!ELEMENT ItemId (#PCDATA)>
 <!ELEMENT ItemName (#PCDATA)>
 <!ELEMENT Quantity (#PCDATA)>
 <!ATTLIST Quantity units CDATA #IMPLIED>

]>
 <Order>
      <Date>1999/07/04</Date>
      <CustomerId>123</CustomerId>
      <CustomerName>Acme Alpha</CustomerName>

   <Item>
      …
 </Item>
 </Order>
```

DTDs are not required for XML documents. However, a valid XML document has a DTD and conforms to that DTD.

CHAPTER 2 XML Query Functions

This chapter describes the XML query functions in detail, and describes the general format of the *option_string* parameter.

XML query functions

This section describes the SQL extensions for accessing and processing XML documents in SQL statements. xmlextract, xmlparse, and xmltest are new reserved words, introduced by XML services.

The functions are:

Table 2-1: XML query functions

Function	Description
xmlextract	A built-in function that applies an XML query expression to an XML document and returns the selected result.
xmltest	A SQL predicate that applies an XML query expression to an XML document and returns the boolean result.
xmlparse	A built-in function that parses and indexes an XML document for more efficient processing.
xmlrepresentation	A built-in function that determines whether a given image column contains a parsed XML document.
xmlvalidate	A built-in function that validates an XML document against a DTD or XML schema.

Example sections

The descriptions of these functions include examples that reference Appendix A, “The sample_docs Example Table” which includes a script for creating and populating the table.

xmlextract

A built-in function that applies the *XML_query_expression* to the *xml_data_expression* and returns the result. This function resembles a SQL substring operation.

Syntax

```
xmlextract_expression ::=
     xmlextract (xml_query_expression,xml_data_expression
     [optional_parameters])
xml_query_expression ::=basic_string_expression
xml_data_expression ::= general_string_expression
optional_parameters ::=
    options_parameter
    |returns_type
    | options_parameter returns_type
options_parameter ::= [,] option option_string
returns_type ::= [,] returns dataype
```

```
datatype ::= {string_type | computational_type | date_time_type }
string_type ::= char (integer) | varchar (integer)
   | unichar (integer) | univarchar (integer)
   | text | unitext | image
computational_type ::= integer_type | decimal_type | real_type
   | date_time_type
integer_type ::= [ unsigned ] {integer | int | tinyint | smallint | bigint}
decimal_type ::= {decimal | dec | numeric } [ (integer [, integer ] ) ]
real_type ::= real | float | double precision
date_time_type ::= date | time | datetime
option_string ::= [,] basic_string_expression
```

Description

Note For information on I18N data, see “XML Support for I18N” on page 113.

- A *basic_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, or *java.lang.String*.
- A *general_string_expression* is a *sql_query_expression* whose datatype is text, image, character, varchar, unitext, unichar, univarchar,or *java.lang.String*.
- An *xmlextract* expression can be used in SQL language wherever a character expression is allowed.
- The default value of *options_parameter* is an empty string. A null options parameter is treated as an empty string.
- If the value of the *xml_query_expression*, or the document argument of xmlextract() is null, the result of xmlextract() is null.
- The value of the *xml_data_expression* parameter is the runtime context for execution of the XML query expression.
- The datatype of xmlextract() is specified by the *returns_type*.
- The default value of *returns_type* is text.
- If the *returns_type* specifies varchar without an integer, the default value is 255.
- If the *returns_type* specifies numeric or decimal without a precision (the first integer), the default value is 18. If it is specified without a scale (the second integer), the default is 0.
- If either the query or document argument is null, xmlextract returns null.
- If the XPath query is invalid, xmlextract raises an exception.

- The initial result of xmlextract is the result of applying the *xml_query_expression* to the *xml_data_expression*. That result is specified by the XPath standard.
- If the *returns_type* specifies a *string_type*, the initial result value is returned as a character-string document of that datatype.
- If the *returns_type* specifies a *computational_type* or *datetime_type* datatype, the initial result is converted to that datatype and returned. The conversion follows the rules specified for the convert built-in function.

  Note The initial result must be a value suitable for the convert built-in function. This requires using the text() reference in the XML query expression. See the examples following.

Note See Chapter 3, “XML Language and XML Query Language,”for the following topics:

- Restrictions on external URI references, XML namespaces, and XML schemas.
- Treatment of predefined entities and their corresponding characters: *&amp*; (&), *&lt*; (<), *&gt;* (>), *&quote;* (“), and *&apos;* (’). Be careful to include the semicolon as part of the entity.
- Treatment of whitespace.
- Treatment of empty elements.

option_string

The general format of the *option_string* is described in “option_strings: general format” on page 39.

The options supported for the xmlextract function are:

```
xmlerror = {exception | null | message}
ncr = {no | non_ascii | non_server}
```

For a description of the ncr option, including its default value, see “XML Support for I18N” on page 113.

Exceptions

If the value of the *xml_data_expression* is not not valid XML, or is an all blank or empty string:

- If the explicit or default option specifies that xmlerror=exception, an exception is raised.

- If the explicit or default option specifies xmlerror=null a null value is returned.
- If the explicit or default options specifies xmlerror=message, a character string containing an XML element, which contains the exception message, is returned. This value is valid XML.
- Global variable *@@error* returns the error number of the last error, whether the value of xmlerror is exception, null, or message.

If the *returns_type* of the *xmlextract_expression* is a *string_type* and the runtime result of evaluating the *xml_query_expression* parameter is longer than the maximum length of a that type, an exception is raised.

Examples

The following examples use the *sample_docs* table described in Appendix A, "The sample_docs Example Table".

This example selects the title of documents that have a bookstore/book/price of 55 or a bookstore/book/author/degree whose from attribute is "Harvard".

```
select xmlextract('/bookstore/book[price=55
   | author/degree/[@from="Harvard"] ]/title'
     text_doc )
from sample_docs
-----------------------------------------------------
<title>History of Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>

NULL

NULL
```

The following example selects the row/pub_id elements of documents whose row elements either have a price element that is less than 10 or a city element equal to "Boston".

This query returns three rows:

- A null value from the bookstore row
- A single "<row>...</row>" element from the publishers row
- 4 "<row>...</row>" elements from the titles row

```
select xmlextract('//row[price<10 | city="Boston" ]/pub_id',
     text_doc) from sample_docs2>
------------------------------------
NULL
XML Services<pub_id>0736</pub_id>
```

```
<pub_id>0736</pub_id>
<pub_id>0877</pub_id>
<pub_id>0736</pub_id>
<pub_id>0736</pub_id>

(3 rows affected)
```

The following example selects the price of “Seven Years in Trenton” as an integer. This query has a number of steps.

1 To select the price of “Seven Years in Trenton” as an XML element:

```
select xmlextract
('/bookstore/book[title="Seven Years in Trenton"]/price',text_doc)
from sample_docs
where name_doc='bookstore'
--------------------------------------
<price>12</price>
```

2 The following attempts to select the full price as an integer by adding a returns integer clause:

```
select xmlextract
     ('/bookstore/book[title="Seven Years in Trenton"]/price',
    text_doc returns integer)
    from sample_docs
    where name_doc='bookstore'
Msg 249, Level 16, State 1:
Line 1:
Syntax error during explicit conversion of VARCHAR value
'<price>12</price>' to an INT field.
```

3 To specify a returns clause with a numeric, money, or date-time datatype, the XML query must return value suitable for conversion to the specified datatype. The query must therefore use the text() reference to remove the XML tags:

```
select xmlextract
  ('/bookstore/book[title="Seven Years in Trenton"]/price/text()',
  text_doc returns integer)
  from sample_docs
  where name_doc='bookstore'
-----------
         12
```

4 To specify a returns clause with a numeric, money, or date-time datatype, the XML query must also return a single value, not a list. For example, the following query returns a list of prices:

```
select xmlextract
    ('/bookstore/book/price',
    text_doc)
    from sample_docs
    where name_doc='bookstore'
-----------
<price>12</price>
<price>55</price>
<price intl="canada" exchange="0.7">6.50</price>
```

5. Adding the text() reference yields the following result:

```
select xmlextract
('/bookstore/book/price/text()',
text_doc)
from sample_docs
where name_doc='bookstore'
-----------------------------
12556.50
```

6. Specifying the returns integer clause produces an exception, indicating that the combined values aren’t suitable for conversion to integer:

```
select xmlextract
   ('/bookstore/book/price/text()',
   text_doc returns integer)
   from sample_docs
   where name_doc='bookstore'
Msg 249, Level 16, State 1:
Line 1:
Syntax error during explicit conversion of VARCHAR
value '12556.50' to an INT field.
```

To illustrate the xmlerror options, the following command inserts an invalid document into the *sample_doc*s table:

```
insert into sample_docs (name_doc, text_doc)
values ('invalid doc', '<a>unclosed element<a>')

(1 row affected)
```

In the following example, the xmlerror options determine the treatment of invalid XML documents by the xmlextract function:

- If xmlerror=exception (this is the default), an exception is raised:

```
select xmlextract('//row', text_doc
   option 'xmlerror=exception')
from sample_docs
```

```
Msg 14702, Level 16, State 0:
Line 2:
XMLPARSE(): XML parser fatal error
   <<The input ended before all started tags
were ended. Last tag started was 'a'>>
   at line 1, offset 23.
```

- If xmlerror=null, a null value is returned:

```
select xmlextract('//row', test_doc
  option 'xmlerror=null')
from sample_docs

(0 rows affected)
```

- If xmlerror=message, a parsed XML document with an error message will be returned:

```
select xmlextract('//row', test_doc
  option 'xmlerror=message')
from sample_docs
------------------------------------

<xml_parse_error>The input ended before all
startedtags were ended. Last tag started was
'a'</xml_parse_error>
```

The xmlerror option doesn't apply to a document that is a parsed XML document or to a document returned by an explicit nested call by xmlparse.

For example, in the following xmlextract call, the xml_data_expression is an unparsed character-string document, so the xmlerror option applies to it. The document is invalid XML, so an exception is raised, and the xmlerror option indicates that the exception message should be returned as an XML document with the exception message:

```
select xmlextract('/',  '<a>A<a>'  option'xmlerror=message')
---------------------------------------------------
<xml_parse_error>The input ended before all started tags were ended.
   Last tag started was 'a'</xml_parse_error>
```

In the following xmlextract call, the xml_data_expression is returned by an explicit call by the xmlparse function (see section “xmlparse” on page 23). Therefore, the default xmlerror option of the explicit xmlparse call applies, rather than the xmlerror option of the outer xmlextract call. That default xmlerror option is exception, so the explicit xmlparse call raises an exception:

```
select xmlextract('/', xmlparse('<a>A<a>')
     option 'xmlerror=message'))
---------------------------------------------------
```

```
Msg 14702, Level 16, State 0:
Line 2:
XMLPARSE(): XML parser fatal error
 <<The input ended before all started tags were ended.
 Last tag started was 'a'>> at line 1, offset 8.
```

To apply the xmlerror=message option to the explicit nested call of xmlparse, specify it as an option in that call:

```
select xmlextract('/',
   xmlparse('<a>A<a>' option 'xmlerror=message'))
----------------------------------------------------
<xml_parse_error>The input ended before all started
tags were ended. Last tag started was
'a'</xml_parse_error>
```

To summarize the treatment of the xmlerror option for unparsed XML documents and nested calls of xmlparse:

- The xmlerror option is used by xmlextract only when the document operand is an unparsed document.
- When the document operand is an explicit xmlparse call, the implicit or explicit xmlerror option of that call overrides the implicit or explicit xmlerror option of the xmlextract.

This command restores the *sample_docs* table to its original state:

```
delete from sample_docs
where na_doc='invalid doc'
```

xmltest

A predicate that evaluates the XML query expression, which can reference the XML document parameter, and returns a Boolean result. Similar to a SQL like predicate.

Syntax

```
xmltest_predicate ::=
     xml_query_expression [not] xmltest xml_data
     [option option_string]
 xml_data ::=
     xml_data_expression | (xml_data_expression)
 xml_query_expression::= basic_string_expression
 xml_data_expression ::= general_string_expression
 option_string ::= basic_string_expression
```

Description

Note For information on processing I18N data, see Chapter 6, "XML Support for I18N."

- A *basic_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, or java.lang.String.
- A *general_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, text, unitext, or java.lang.String.
- An xmltest predicate can be used in SQL language wherever a SQL predicate is allowed.
- An xmltest call specifying that:

  ```
  X not xmltest Y options Z
  ```

  is equivalent to:

  ```
  not X xmltest Y options Z
  ```

- If the *xml_query_expression* or *xml_data_expression* of xmltest() is null, then the result of xmltest() is unknown.
- If the value of the *xml_query_expression*, or the document argument of xmlextract() is null, the result of xmlextract() is null.
- The value of the *xml_data_expression* parameter is the runtime context for execution of the *XPath* expression.
- xmltest() evaluates to boolean *true* or *false*, as follows:
  - The *xml_query_expression* of xmltest() is an XPath expression whose result is *empty* (*not empty*), then xmltest() returns *false* (*true*).
  - If the *xml_query_expression* of xmltest() is an XPath expression whose result is a Boolean *false* (*true*), then xmltest() returns *false* (*true*).

- If the XPath expression is invalid, xmltest raises an exception.

Note See Chapter 3, “XML Language and XML Query Language,” for the following topics:

- Restrictions on external URI references, XML namespaces, and XML schemas.
- Treatment of predefined entities and their corresponding characters: *&amp*; (&), *&lt*; (<), *&gt;* (>), *&quote;* (“), and *&apos;* (’). Be careful to include the semicolon as part of the entity.
- Treatment of whitespace.
- Treatment of empty elements.

Options

The general format of the option_string is described in “option_strings: general format” on page 39.

The option supported for the xmltest predicate is xmlerror = {*exception* | *null*}.

The message alternative, which is supported for xmlextract and xmlparse, is not valid for xmltest. See the Exceptions section.

Exceptions

If the value of the *xml_data_expression* is not valid XML, or is an all blank or empty string:

- If the explicit or default option specifies xmlerror=exception, an exception is raised.
- If the explicit or default options specifies xmlerror=null a null value is returned.
- If you specify xmlerror=message, a null value is returned.

Examples

These examples use the *sample_docs* table described in Appendix A, “The sample_docs Example Table”.

This example selects the *name_doc* of each row whose *text_doc* contains a row/city element equal to “Boston”.

```
select name_doc from sample_docs
where  '//row[city="Boston"]' xmltest text_doc
  name_doc
------------------------
  publishers

(1 row affected)
```

In the following example the xmltest predicate returns *false*/*true*, for a Boolean *false*/*true* result and for an *empty*/*not-empty* result.

```
-- A boolean true is 'true':
select case when '/a="A"' xmltest '<a>A</a>'
            then 'true' else 'false' end2>
-----
true

-- A boolean false is 'false'
select case when '/a="B"' xmltest '<a>A</a>'
            then 'true' else 'false' end
-----
false

-- A non-empty result is 'true'
select case when '/a' xmltest '<a>A</a>'
             then 'true' else 'false' end

-----  true
-- An empty result is 'false'
select case when '/b' xmltest '<a>A</a>'
            then 'true' else 'false' end
-----
false

-- An empty result is 'false' (second example)
select case when '/b="A"' xmltest '<a>A</a>'
            then 'true' else 'false' end
-----
false
```

To illustrate the xmlerror options, the following command inserts an invalid document into the *sample_docs* table:

```
insert into sample_docs (name_doc, text_doc)
values ('invalid doc', '<a>unclosed element<a>')

(1 row affected)
```

In the following examples, the xmlerror options determine the treatment of invalid XML documents by the xmltest predicate.

- If xmlerror=exception (the default result), an exception is raised, and global variable *@@error* contains error message 14702.

```
select name_doc from sample_docs
where '//price<10/*' xmltest text_doc
   option 'xmlerror=exception'

Msg 14702, Level 16, State 0:
Line 2:
XMLPARSE(): XML parser fatal error
   <<The input ended before all started tags were
ended. Last tag started was 'a'>> at line 1,
offset 23.
```

To display the contents of *@@error*, enter:

```
select @@error
------------
     14702
(1 row affected)
```

- If xmlerror=null or xmlerror=message, a null (unknown) value is returned, and global variable *@@error* contains error message 14701.

```
select name_doc from sample_docs
where '//price<10/*' xmltest text_doc
   option 'xmlerror=null'

(0 rows affected)
```

To display the contents of @@error, enter:

```
select @@error
---------
     14701
(1 row affected)
```

This command restores the *sample_docs* table to its original state:

```
delete from sample_docs
where name_doc='invalid doc'
```

xmlparse

A built-in function that parses the XML document passed as a parameter, and returns an image value that contains a parsed form of the document.

Syntax

xmlparse_call ::=
   xmlparse(*general_string_expression*
   [*options_parameter*][*returns_type*])

options_parameter ::= [,] option *option_string*
option_string ::= *basic_string_expression*
returns type ::= [,] returns {image | binary | varbinary [(*integer*)]}

Description

Note For information on processing I18N data, see Chapter 6, "XML Support for I18N."

- If you omit the returns clause, the default is returns image.
- A *basic_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, or java.lang.String.
- A *general_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, text, unitext, image, or java.lang.String.
- If any parameter of xmlparse() is null, the result of the call is null.
- If the *general_string_expression* is an all-blank string, the result of xmlparse is an empty XML document.
- xmlparse() parses the *general_string_expression* as an XML document and returns an image value containing the parsed document.
- If the *general_string_expression* is an image expression, it is assumed to consist of characters in the server character set.

Note See Chapter 3, "XML Language and XML Query Language," for the following topics:

- Restrictions on external URI references, XML namespaces, and XML schemas.
- Treatment of predefined entities and their corresponding characters: *&amp*; (&), *&lt*; (,), *&gt;* (>), *&quote;* ("), and *&apos;* (;). Be careful to include the semicolon as part of the entity.
- Treatment of whitespace.
- Treatment of empty elements.

Options

- The general format of the option_string is described in "option_strings: general format" on page 39. The options supported for the xmlparse function are:

  dtdvalidate = {*yes* | *no*}

  xmlerror = {*exception* | *null* | *message* }

If dtdvalidate=yes is specified, the XML document is validated against its embedded DTD (if any). This option is for compatibility with the Java-based XQL processor of Adaptive Server Enterprise 12.5.

If dtdvalidate=no is specified, no DTD validation is performed. This is the default.

xmlerror = {*exception* | *null* | *message*}

For the xmlerror option, see "Exceptions" below.

Exceptions

If the value of the *xml_data_expression* is not valid XML:

- If the explicit or default options specifies xmlerror=exception, an exception is raised.
- If the explicit or default options specifies xmlerror=null, a null value will be returned.
- If the explicit or default options specifies xmlerror=message, a character string containing an XML element with thee exception messages is returned. This value is valid parsed XML.
- Global variable *@@error* returns the error number of the last error, whether the value of xmlerror is exception, null, or message.

If the value of the *xml_data_expression* is not valid XML:

- If the explicit or default options specifies xmlerror=exception, an exception is raised.
- If the explicit or default options specifies xmlerror=null, a null value will be returned.
- If the explicit or default options specifies xmlerror=message, then a character string containing an XML element with the exception message is returned. This value is valid parsed XML.

Examples

These examples use the *sample_docs* table described in

As created and initialized, the *text_doc* column of the *sample_docs* table contains documents, and the *image_doc* column is null. You can update the *image_doc* columns to contain parsed XML versions of the *text_doc* columns:

```
update sample_docs
set image_doc = xmlparse(text_doc)

(3 rows affected)
```

You can then apply the xmlextract function to the parsed XML documents in the image column in the same way as you apply it to the unparsed XML documents in the text column. Operations on parsed XML documents generally execute faster than on unparsed XML documents.

```
select name_doc,
  xmlextract('/bookstore/book[title="History of Trenton"]/price', text_doc)
    as extract_from_text_doc,
  xmlextract('/bookstore/book[title="History of Trenton"]/price',
image_doc)
    as extract_from_image_doc
from sample_docs

name_doc   extract_from_text_doc  extract_from_image_doc
----------  ---------------------  ------------------------
bookstore   <price>55</price>     <price>55</price>
publishers  NULL                  NULL
titles      NULL                  NULL
(3 rows affected)
```

To illustrate the xmlerror options, this command inserts an invalid document into the *sample_docs* table

```
insert into sample_docs (name_doc, text_doc) ,
values ('invalid doc', '<a>unclosed element<a>')

(1 row affected)
```

In the following example, the xmlerror options determine the treatment of invalid XML documents by the xmlparse function:

- If xmlerror=exception (the default), an exception is raised:

```
update sample_docs
set image_doc = xmlparse(text_doc option 'xmlerror=exception')

Msg 14702, Level 16, State 0:
Line 2:
XMLPARSE(): XML parser fatal error
  <<The input ended before all started tags were ended. Last tag started
was 'a'>> at line 1, offset 23.
```

- If xmlerror=null, a null value is returned:

```
update sample_docs
set image_doc = xmlparse(text_doc option 'xmlerror=null')

select image_doc from sample_docs
where name_doc='invalid doc'
```

```
------
NULL
```

- If xmlerror=message, then parsed XML document with the error message is returned:

```
update sample_docs
set image_doc = xmlparse(text_doc option 'xmlerror=message')

select xmlextract('/', image_doc)
from sample_docs
where name_doc = 'invalid doc'
------------------------
<xml_parse_error>The input ended before all started tags were ended.
Last tag started was 'a'</xml_parse_error>
```

This command restores the sample_docs table to its original state:

```
delete from sample_docs
where name_doc='invalid doc'
```

xmlrepresentation

Examines the *image* parameter, and returns an integer value indicating whether the parameter contains parsed XML data or other sorts of image data.

Syntax

xmlrepresentation_call::=
    xmlrepresentation(*parsed_xml_expression*)

Description

- A *parsed_xml_expression* is a *sql_query_expression* whose datatype is image, binary, or varbinary.
- If the parameter of xmlrepresentation() is null, the result of the call is null.
- xmlrepresentation returns an integer 0 if the operand is parsed XML data, and a positive integer if the operand is either not parsed XML data or an all blank or empty string.

Examples

These examples use the *sample_docs* table described in Appendix A, “The sample_docs Example Table”.

Example 1 This example illustrates the basic xmlrepresentation function.

```
-- Return a non-zero value
-- for a document that is not parsed XML
```

```
select xmlrepresentation(
        xmlextract('/', '<a>A</a>' returns image)

-----------
1

-- Return a zero for a document that is parsed XML
select xmlrepresentation(
        xmlparse(
          xmlextract('/', '<a>A</a>' returns image))
-----------
0
```

Example 2 Columns of datatype image can contain both parsed XML documents (generated by the xmlparse function) and unparsed XML documents. After the update commands in this example, the image_doc column of the sample_docs table contains a parsed XML document for the titles document, an unparsed (character-string) XML document for the bookstore document, and a null for the publishers document (the original value).

```
update sample_docs
set image_doc = xmlextract('/', text_doc returns image)
where name_doc = 'bookstore'

update sample_docs
set image_doc = xmlparse(text_doc)
where name_doc = 'titles'
```

Example 3 You can use the xmlrepresentation function to determine whether the value of an image column is a parsed XML document:

```
select name_doc, xmlrepresentation(image_doc)from
sample_docs

name_doc
---------     -----------
bookstore     1
publishers    NULL
titles        0

(3 rows affected)
```

Example 4 You can update an image column and set all of its values to parsed XML documents. If the image column contains a mixture of parsed and unparsed XML documents, a simple update raises an exception.

```
update sample_docs set image_doc = xmlparse(image_doc)
```

```
Msg 14904, Level 16, State 0:
Line 1:
XMLPARSE: Attempt to parse an already parsed XML
   document.
```

Example 5 You can avoid such an exception by using the xmlrepresentation function:

```
update sample_docs
set image_doc = xmlparse(image_doc)
where xmlrepresentation(image_doc) != 0

(1 row affected)
```

Example 6 This command restores the sample_docs table to its original state.

```
update sample_docs
set image_doc = null
```

xmlvalidate

Validates an XML document.

Syntax

xmlvalidate_call ::=
  xmlvalidate (*general_string_expression*, [*optional_parameters*])
optional_parameters ::= *options_parameter*
  | *returns_type*
  |*options_parameter* returns type
options_parameter ::= [,] option *option_string*
options_string ::= *basic_string_expression*
returns_type ::= [,] returns *string_type*
string_type ::=*char* (*integer*) | *varchar* (*integer*)
  | *unichar* (*integer*) |*univarchar* (*integer*)
  | *text* | *unitext* |*image* | *java.lang.String*

Description

Note For information on validating Unicode, see Chapter 6, “XML Support for I18N.”

- A *basic_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, or java.lang.String.
- A *general_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, text, unitext, or java.lang.String.

- If any parameter of xmlvalidate() is null, the result of the call is null.
- The result datatype of an xmlvalidate_call is the datatype specified by the *returns_type*.

Options

The general format of the *option_string* is described in “option_strings: general format” on page 39. The options supported for xmlvalidate are:

```
validation_options::=
     [dtdvalidate = {no | yes | strict}]
     [schemavalidate = {no | yes}]
     [nonamespaceschemalocation = 'schema_uri_list']
     [schemalocation = 'namespace_schema_uri_list']
     [xmlerror = {exception | null | message }]
     [xmlvalid = {document | message}]
 schema_uri_list::=
     schema_uri [schema_uri]...
 namespace_schema_uri_list ::=
     namespacename schema_uri
     [ namespacename schema_uri]...
 schema_uri ::= character_string
 namespacename ::= character_string
```

Options description

- The defaults for *validation_options* are:

  dtdvalidate = See below
  schemavalidate = *no*
  schemalocation = " "
  nonamespaceschemalocation = " "
  xmlerror = exception
  xmlvalid = document

- Keywords in a *validation_option* are not case-sensitive, but the *schema_uri_list* and *namespace_schema_uri_list* are case-sensitive.
- Refer to the document you parse or store as the subject XML document.
- The default for dtdvalidate depends on the implicit or explicit value of the schemavalidate option. If the schemavalidate option value is *no*, the default value of dtdvalidate is *no*. If the schemavalidate option value is *yes*, the default dtdvalidate option value is *strict*.
- If you specify schemavalidate = *yes*, you must either specify dtdvalidate = *strict* or omit dtdvalidate.
- If you specify dtdvalidate = *no* with schemavalidate = *no*, the document is checked for well-formedness only.

- If you specify schemavalidate = *no*, the clauses nonamespaceschemalocation and schemalocation are ignored.
- The values specified in the clauses nonamespaceschemalocation and schemalocation are character literals. If the Transact-SQL quoted_identifier option is *off*, you can choose either apostrophes (') or quotation marks (") to surround the *option_string*, and use the other to surround the values specified by nonamespacescemalocation and schemalocation. If the Transact-SQL quoted_identifier option is *on*, you must surround the *option_string* with apostrophes ('), and you must surround the values specified by nonamespacescemalocation and schemalocation by quotation marks (").
- nonamespaceschemalocation specifies a list of schema URIs, which overrides the list of schema uris specified in the xsi:noNameSpaceschemalocation clause in the subject XML document.
- schemalocation specifies a list of pairs, each pair consisting of a namespace name and a schema URI.
  - a namespace name is the name an xmlns attribute specifies for a namespace. *http://acme.com/schemas.contract* is declared as the default namespace in this example:

```
<contract xmlns="http://acme.com/schemas.contract">
```

In this example, however, it is declared as the namespace for the prefix "co":

```
<co:contract xmlns:co="http://acme.com/schemas.contract">
```

The namespace name is the URI specified in a namespace declaration itself, not the prefix.

- A *schema_uri* is a character string literal that contains a schema URI. The maximum length of a *URI_string* is 1927 characters, and it must specify *http*. The schema referenced by a *schema_uri* must be encoded as either UTF8 or UTF16.
- The dtdvalidate option values are:

  dtdvalidate=*no*: No DTD or schema validation is performed; the document is checked to ensure that it is well-formed.

  dtdvalidate=*yes*: The document is validated against any DTD the document specifies.

  dtdvalidate=*strict*: This option depends on the schemavalidate option.

  - schemavalidate=*no*:
    You must specify a DTD in the subject XML document, and the document is validated against that DTD.
  - schemavalidate=*yes*:
    You must declare every element in the subject XML document in a DTD or a schema, and each element is validated against those declarations.

- The schemavalidate option values are:

  If you specify schemavalidate=*no*, no schema validation is performed for the subject XML document.

  If you specify schemavalidate=*yes*, schema validation is performed.

- The following results apply when a *general_string_expression*, for instance *XC*, is an XML document that passes the validation options specified in the *option_string* clause:

  If xmlvalid specifies doc, the result of xmlvalidate is:

  ```
  convert(text, XC)
  ```

  If xmlvalid specifies message, the result of xmlvalidate is this XML document:

  ```
  <xmlvalid/>
  ```

- The following results apply when a *general_string_expression* is not an XML document that passes the validation options specified in the *option_string* clause:

  If the *option_string* specifies xmlerror=*exception*, an exception is raised carrying the *exception* message.

  If *option_string* specifies xmlerror=message, an XML document of the following form is returned. E1, E2, and so forth are messages that describe the validation errors.

```
<xml_validation_errors>
     <xml_validation_error>E1</xml_validation_error>
     <xml_validation_error>E2</xml_validation_error>
...
     <xml_validation_warning>W1</xml_validation_warning>
     <xml_vvalidation_fatal_error>E3<xml_validation_fatal_error>
</xml_validation_errors>
```

If *option_string* specifies xmlerror=*null*, a null value is returned.

Exceptions

If the value of the *xml_data_expression* is not valid XML:

- If the explicit or default options specifies xmlerror=exception, an exception is raised.
- If the explicit or default options specifies xmlerror=null, a null value will be returned.
- If the explicit or default options specifies xmlerror=message, a character string containing an XML element with all the exception messages is returned. This value is valid parsed XML.
- Global variable *@@error* returns the error number of the last error, whether the value of xmlerror is exception, null, or message.
- If a web resource required for validation is unavailable, an exception occurs.
- If the source XML document is either invalid or not well-formed, an exception occurs. Its message describes the validation failure.

Examples

The XML DTDs and schemas shown in Table 2-2 illustrate validation clauses.

- *dtd_emp* and *schema_emp* define a single text element, "<*emp_name*>"
- *dtd_cust* and *schema_cust* define a single text element , "<*cust_name*>"
- *ns_schema_emp* and *ns_schema_cust* are variants that specify a target namespace.

Table 2-2: Example DTDs and schemas, and their URIs

URI	Document
http://test/dtd_emp.dtd	<!ELEMENT emp_name (#PCDATA)>
http://test/dtd_cust.dtd	<!ELEMENT cust_name (#PCDATA)>
http://test/schema_emp.xsd	<xsd:schema xmlns:xsd ="http://www.w3.org/2001/XMLSchema" targetNamespace ="http://test/ns_schema_emp"> <xsd:element name="emp_name" type="xsd:string"/> </xsd:schema>
http://test/ns_schema_emp.xsd	<xsd:schema xmlns:xsd ="http://www.w3.org/2001/XMLSchema" targetNamespace ="http://test/ns_schema_emp"> <xsd:element name="emp_name" type="xsd:string"/> </xsd:schema>

URI	Document
http://test/schema_cust.xsd	<xsd:schema xmlns:xsd ="http://www.w3.org/2001/XMLSchema"> <xsd:element name="cust_name" type="xsd:string"/> </xsd:schema>

http://test/ns_schema_cust.xsd	<xsd:schema xmlns:xsd
	   ="http://www.w3.org/2001/XMLSchema"
	   targetNamespace
	   ="http://test/ns_schema_cust">
	<xsd:element name="cust_name"
	   type="xsd:string"/>
	
	</xsd:schema>

Example 1 This example creates a table in which to store XML documents in a text column. Use this table to show example calls of xmlvalidate. In other words, xmlvalidate explicitly validates documents stored in the text column.

```
create table text_docs(xml_doc text null)
```

Example 2 This example shows xmlvalidate specifying a document with no DTD declaration, and the validation option *dtdvalidate=yes*. The command succeeds because the inserted document is well-formed, and dtdvalidate is not specified as *strict*.

```
insert into text_docs
values (xmlvalidate(
  '<employee_name>John Doe</employee_name>',
  option 'dtdvalidate=yes'))
---------
(1 row inserted)
```

Example 3 This example shows xmlvalidate specifying a document with no DTD declaration and the validation option *dtdvalidate=strict*. xmlvalidate raises an exception, because strict DTD validation requires every element in the document to be specified by a DTD.

```
insert into text_docs
values(xmlvalidate(
  '<emp_name>John Doe</emp_name>',
option 'dtdvalidate=strict'))
--------
EXCEPTION
```

Example 4 The last example raised an exception when validation failed. Instead, you can use the option xmlerror to specify that xmlvalidate should return null when validation fails.

```
insert into text_docs
values(xmlvalidate(
  '<emp_name>John Doe</emp_name>'
option 'dtdvalidate=strict xmlerror=null'))
-------
null
```

Example 5 You can also use xmlerror to specify that xmlvalidate should return the XML error message as an XML document when validation fails:

```
insert into text_docs
values(xmlvalidate(
  '<emp_name>John Doe</emp_name>'
option 'dtdvalidate=strict xmlerror=message'))
--------
<xml_validation_errors>
<xml_validation_error>(1:15)Document is invalid:
  no grammar found.<xml_validation_error>
<xml_validation_error>(1:15)Document root element
   "employee name",must match DOCTYPE root
   "null."</xml_validation_error>
</xml_validation_errors>
```

Example 6 This example shows xmlvalidate specifying a document that references both a DTD and the validation option *dtdvalidate=yes*. This command succeeds.

```
insert into text_docs
values(xmlvalidate(
  '<DOCTYPE emp_name PUBLIC "http://test/dtd_emp.dtd">
  <emp_name>John Doe</emp_name>',
option 'dtdvalidate=yes'))
-------
(1 row inserted)
```

Example 7 This example shows xmlvalidate specifying a document that references a DTD and the validation option *dtdvalidate=yes*. xmlvalidate raises an exception, because the inserted document does not match the DTD referenced in the document.

```
insert into text_docs
values(xmlvalidate(
  '<DOCTYPE emp_name PUBLIC "http://test/dtd_cust.dtd">
<emp_name>John Doe</emp_name>',
option 'dtdvalidate=yes'))
```

```
--------
EXCEPTION
```

Example 8 This example shows xmlvalidate specifying a document with no schema declaration and the validation option *schemavalidate=yes*. This command fails because the '<emp_name>' element has no declaration.

```
insert into text_docs
values(xmlvalidate('<emp_name>John Doe</emp_name>',
  option 'schemavalidate=yes'))
-------
EXCEPTION
```

Example 9 This example shows xmlvalidate specifying a document with a schema declaration and the validation option *schemavalidate=yes*. This document does not use namespaces. The command succeeds, because the document matches the schema referenced in the document.

```
insert into text_docs
values(xmlvalidate(
  '<emp_name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi: noNamespaceSchemaLocation="http://test/schema_emp.xsd">
  John Doe</emp_name>'
 option 'schemavalidate=yes'))
--------
(1 row inserted)
```

Example 10 This example shows xmlvalidate specifying a document that specifies a namespace and the validation option *schemavalidate=yes*. The command succeeds, because the document matches the schema referenced in the document.

```
insert into text_docs
values(xmlvalidate(
  '<emp:emp_name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:emp="http://test/ns_schema_emp"
   xsi: SchemaLocation="http://test/ns_schema_emp
      http://test/ns_schema_emp.xsd">
  John Doe</emp:emp_name>'
 option 'schemavalidate=yes'))
--------
(1 row inserted)
```

Example 11 This example shows xmlvalidate specifying a document with a schema declaration and the validation option *schemavalidate=yes*. This command fails, because the document doesn’t match the schema referenced in the document.

```
insert into text_docs
```

```
values (xmlvalidate(
  '<emp_name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="http://test/schema_cust.xsd">
   John Doe</emp_name>'
 option 'schemavalidate=yes'))
-------
EXCEPTION
```

Example 12 This example shows xmlvalidate specifying a document with a schema declaration and the validation option *schemavalidate=yes*. This document specifies a namespace. The command fails, because the document doesn't match the schema referenced in the document.

```
insert into text_docs
values(xmlvalidate(
  '<emp:emp_name
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:emp="http://test/ns_schema_cust"
    xsi:schemaLocation=
       "http://test/ns_schema_cust http://test/ns_schema_cust.xsd">
    John Doe</emp:emp_name>',
   option 'schemavalidate=yes'))
------------
EXCEPTION
```

The validation options of xmlvalidate specify a nonamespaceschemalocation of *http://test/ns_schema_emp.xsd*.

Example 13 This example shows xmlvalidate specifying a document with a schema declaration and the validation option *schemavalidate=yes*, as well as the clauses schemalocation and nonamespaceschemalocation.

The document specifies a schemaLocation of *http://test/schema_cust.xsd*, and the validation option in xmlvalidate specifies a schemalocation of *http://test/ns_schema_emp.xsd*.

This command succeeds, because the document matches the schema referenced in xmlvalidate, which overrides the schema referenced in the document.

```
insert into text_docs
values (xmlvalidate(
  '<emp:emp_name
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:emp="http://test/schema_emp"
    xsi:schemaLocation="http://test/ns_schema_emp
         http://test/schema_cust.xsd">
    John Doe</emp:emp_name>',
   option 'schemavalidate=yes,
    schemalocation= "http://test/ns_schema_emp
         http://test/ns_schema_emp.xsd"
    nonamespaceschemalocation="http://test/schema_emp.xsd" '))
-------------
(1 row inserted)
```

Example 14 This example shows xmlvalidate specifying a document with a schema declaration and the validation option *schemavalidate=yes*, as well as the clauses schemalocation and nonamespaceschemalocation.

The document specifies a noNamespaceSchemaLocation of *http://test/schema_cust.xsd*, and the validation option in xmlvalidate specifies a nonamespaceschemalocation of *http://test/ns_schema_emp.xsd*.

This command fails, because the document doesn't match the schema referenced in xmlvalidate. The document does, however, match the schema referenced in the document.

```
insert into text_docs
values(xmlvalidate(
  '<customer_name
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://test/schema_cust.xsd">
  John Doe</customer_name>'
  option 'schemavalidate=yes,
schemalocation="http://test/ns_schema_emp http://test/ns_schema_emp.xsd"
    nonamespaceschemalocation="http://test/schema_emp.xsd" '))
-----------
EXCEPTION
```

Example 15 This example shows xmlvalidate specifying a document with a schema declaration and the validation option *schemavalidate=yes*, as well as the clauses schemalocation and nonamespaceschemalocation

The document specifies a *schemaLocation* of *http://test/schema_cust.xsd*, and the validation option of xmlvalidate specifies a *schemalocation* of *http://test/ns_schema_emp.xsd*.

This command fails, because the document doesn't match the schema referenced in xmlvalidate. The document does, however, match the schema referenced in the document.

```
insert into text_docs
values(xmlvalidate(
  '<cust:cust_name
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cust="http://test/schema_cust"
    xsi:schemaLocation="http://test/schema_cust
          http://test/schema_cust.xsd">
   John Doe</cust:cust_name>',
  option 'schemavalidate=yes,
   schemalocation="http://test/ns_schema_emp
http://test/ns_schema_emp.xsd"
   nonamespaceschemalocation="http://test/schema_emp.xsd" '))
-------
(1 row inserted)
```

option_strings: general format

This section specifies the general format, syntax and processing of option string parameters in XML Services. Actions of individual options are described in the functions that reference them.

Any function that has an *option_string* parameter accepts the union of all options, and ignores any options that do not apply to that particular function.

For example, forxmlj does not have an *XML document* parameter, but it still accepts an *option_string* containing the xmlerror option (which specifies actions for invalid XML operands).

This "union options" approach lets you use a single *option_string* variable for all XML Services functions.

Syntax

option_string::= *basic_string_expression*

Description

- The complete syntax of the runtime value of the *option_string* parameter is:

  option_string_value ::= *option* [[,] *option*] …
  option ::= *name* = *value*
  name ::= option name as listed below
  value ::= *simple_identifier* | *quoted_string*

- If an option_string parameter is null, the empty strings are all blanks.
- You can use any amount of white space before the first option, after the last option, between options, and around the equals signs.
- You can separate options using commas or by white space.

- An *option_value* can be either a simple identifier, beginning with a letter and continuing with letters, digits, and underscores, or a quoted string. Quoted strings are formed using the normal SQL conventions for embedded quotes.
- The set of options, and the functions to which they are applicable, are shown in Table 2-3. See specific function descriptions for descriptions of options.

Option values for query functions.

Note Underlining indicates the default values of options that specify keywords in this table. Parentheses show the default values of options specifying SQL names. The empty string, or a single-space character, specifies the default values of options specifying string values.

Table 2-3: Option string values

Option name	Option value	Function
binary	hex | base64	forxmlj and for xml clause
columnstyle	element | attribute	forxmlj and for xml clause
dtdvalidate	yes | no	xmlvalidate
entitize	yes | no | conditional	forxmlj and for xml clause
format	yes | no	forxmlj and for xml clause
header	yes | no |encoding	forxmlj and for xml clause
incremental	yes | no	for xmlj clause
nonamespaceschemalocation	See xmlvalidate	xmlvalidate
ncr	non_ascii | non_server | no See function description for default value.	forxmlj, for xml clause, xmlextract
nullstyle	attribute | omit	forxmlj and for xml clause
nullclause	null | empty	forsqlcreatej forsqlscriptj
prefix	SQL name (C) The default value is C.	forxmlj and for xml clause
root	yes | no	forxmlj and for xml clause
rowname	SQL name (row)	forxmlj and for xml clause
schemalocation	See xmlvalidate	forxmlj and for xml clause
schemavalidate	yes | no	xmlvalidate

Option name	Option value	Function
statement	yes | no	for xmlj and forxml clause
tablename	SQL name (resultset)	forxmlj and for xml clause
targetns	quoted string with a URI	forxmlj and for xml clause
xmlerror	exception | null | message	all functions with XML operands
xmlvalid	document | message	xmlvalidate
xsidecl	yes | no	forxmlj and for xml clause

CHAPTER 3

XML Language and XML Query Language

The XML query functions support the XML 1.0 standard for XML documents and the XPath 1.0 standard for XML queries. This chapter describes the subsets of those standards that XML Services support.

Character set support

XML Services supports the character sets supported by the SQL server. For more information on I18N, see Chapter 6, “XML Support for I18N.”

URI support

XML documents specify URIs (Universal Resource Indicators) in two contexts, as href attributes or document text, and as external references for DTDs, entity definitions, XML schemas, and namespace declarations.

There are no restrictions on the use of URIs as href attributes or document text, and XML Services resolves external reference URIs that specify http URIs.

External-reference URIs that specify file, ftp, or relative URIs are not supported.

Namespace support

You can parse and store XML documents with namespace declarations and references with no restriction.

However, when XML element and attribute names that have namespace prefixes are referenced in XM expressions in xmlextract and in xmltest, the namespace prefix and colon are treated as part of the element or attribute name. They are not processed as namespace references.

XML schema support

See Chapter 7, “xmltable()” for information on xmlvalidate.

Predefined entities in XML documents

The special characters for quote ("), apostrophe ('), less-than (<), greater-than (>), and ampersand (&) are used for punctuation in XML, and are represented with predefined entities: *&quot;*, *&apos;*, *&lt;*, *&gt;*, and *&amp;*. Notice that the semicolon is part of the entity.

You cannot use "<" or "&" in attributes or elements, as the following series of examples demonstrates.

```
select xmlparse("<a atr='<'/>")

Msg 14702, Level 16, State 0:
Line 1:
XMLPARSE(): XML parser fatal error <<A '<' character
cannot be used in attribute 'atr', except through <&gt;>
at line 1, offset 14.
```

```
select xmlparse("<a atr1='&'>")

Msg 14702, Level 16, State 0:
Line 1:
XMLPARSE(): XML parser fatal error
<<Expected entity name for reference>>
at line 1, offset 11

select xmlparse("<a> < </a>")

Msg 14702, Level 16, State 0:
Line 2:
XMLPARSE(): XML parser fatal error
<<Expected an element name>>
at line 1, offset 6.

select xmlparse(" & ")
Msg 14702, Level 16, State 0:
Line 1:
XMLPARSE(): XML parser fatal error
<<Expected entity name for reference>>
at line 1, offset 6.
```

Instead, use the predefined entities *&lt;* and *&amp;*, as follows:

```
select xmlextract("/",
    "<a atr='&lt; &amp;'> &lt; &amp; </a>" )
--------------------------------
          <a atr="&lt; &amp;"> &lt; &amp; </a>
```

You can use quotation marks within attributes delimited by apostrophes, and vice versa. These marks are replaced by the predefined entities *&quot;* or *&apos;*. In the following examples, notice that the quotation marks or apostrophes surrounding the word 'yes' are doubled to comply with the SQL character literal convention:

```
select xmlextract("/", "<a atr=' ""yes"" '/> " )
---------------------------------
           <a atr=" "yes" "></a>

select xmlextract('/', '<a atr=" ''yes'' "/> ' )
----------------------------
            <a atr=" 'yes' "></a>
```

You can use quotation marks and apostrophes within elements. They are replaced by the predefined entities *&quot;* and *&apol:*, as the following example shows:

```
select xmlextract("/", " ""yes"" and 'no' " )
-------------------------------------
          &quot;yes&quot; and 'no'
```

Predefined entities in XPath queries

When you specify XML queries with character literals that contain the XML special characters, you can write them as either plain characters or as predefined entities. The following example shows two points:

- The XML document contains an element "<a>" whose value is the XML special characters &<>", represented by their predefined entities, *&amp;&lt;&gt;&quot;*
- The XML query specifies a character literal with those same XML special characters, also represented by their predefined entities.

```
select xmlextract('/a="&amp;&lt;&gt;&quot;"',
      "<a>&amp;&lt;&gt;&quot;</a>")
----------------------------------
          <a>&amp;&lt;&gt;&quot;</a>
```

The following example is the same, except that the XML query specifies the character literal with the plain XML special characters. Those XML special characters are replaced by the predefined entities before the query is evaluated.

```
select xmlextract("/a='&<>""' " ,
            "<a>&amp;&lt;&gt;&quot;</a>")
----------------------------------
          <a>&amp;&lt;&gt;&quot;</a>
```

White space

All white space is preserved, and is significant in queries.

```
select xmlextract("/a[@atr='  this  or  that  ' ]",
     "<a atr='  this  or  that  '><b>  which  or  what
      </b></a>")
-------------------------------------------------
     <a atr="  this  or  that  ">
     <b>  which  or  what  </b></a>
```

```
select xmlextract("/a[b='  which  or  what  ']",
      "<a atr='  this  or  that  '><b>  which  or  what
       </b></a>")
---------------------------------------------
      <a atr='  this  or  that  '>
       <b>  which  or  what  </b></a>
```

Empty elements

Empty elements that are entered in the style "<a/>" are stored and returned in the style "<a></a>":

```
select xmlextract("/",
       "<doc><a/> <b></b></doc>")
-----------------------------------------
          <doc>
          <a></a>
          <b></b></doc>
```

XML Query Language

XML Services supports a subset of the standard XPath Language. That subset is defined by the syntax and tokens in the following section.

XPath-supported syntax and tokens

XML Services supports the following XPath syntax:

```
xpath::= or_expr
or_expr::= and_expr | and_expr TOKEN_OR or_expr
and_expr::= union_expr | union_expr TOKEN_AND and_expr
union_expr::= intersect_expr
      | intersect_expr TOKEN_UNION union_expr
intersect_expr::= comparison_expr
      | comparison_expr TOKEN_INTERSECT intersect_expr
comparison_expr::= range_exp
      | range_expr general_comp comparisonRightHandSide
general_comp::= TOKEN_EQUAL | TOKEN_NOTEQUAL
```

```
    | TOKEN_LESSTHAN | TOKEN_LESSTHANEQUAL
    | TOKEN_GREATERTHAN | TOKEN_GREATERTHANEQUAL
range_expr::= unary_expr | unary_expr TOKEN_TO unary_expr
unary_expr::= TOKEN_MINUS path_expr
    | TOKEN_PLUS path_expr
    | path_expr
comparisonRightHandSide::= literal
path_expr::= relativepath_expr | TOKEN_SLASH
    | TOKEN_SLASH relativepath_expr
    | TOKEN_DOUBLESLASH relativepath_expr
relativepath_expr::= step_expr
    | step_expr TOKEN_SLASH relativepath_expr
    | step_expr TOKEN_DOUBLESLASH relativepath_expr
step_expr::= forward_step predicates
    | primary_expr predicates
    | predicates
primary_expr::= literal | function_call | (xpath)
function_call::=
    tolower([xpath])
    | toupper([xpath])
    | normalize-space([xpath])
    | concat([xpath [,xpath]...])
forward_step::= abbreviated_forward_step
abbreviated_forward_step::= name_test
    | TOKEN_ATRATE name_test
    | TOKEN_PERIOD
name_test::= q_name | wild_card | text test
text_test ::= TOKEN_TEXT TOKEN_LPAREN TOKEN_RPAREN
literal::= numeric_literal | string_literal
wild_card::= TOKEN_ASTERISK
q_name::= TOKEN_ID
string_literal::= TOKEN_STRING
numeric_literal::= TOKEN_INT | TOKEN_FLOATVAL|
    | TOKEN_MINUS TOKEN_INT
    | TOKEN_MINUSTOKEN_FLOATVAL
predicates::=
    | TOKEN_LSQUARE expr TOKEN_RSQUARE predicates
    | TOKEN_LSQUARE expr TOKEN_RSQUARE
```

The following tokens are supported by the XML Services subset of XPath:

```
APOS ::= '''
DIGITS ::= [0-9]+
NONAPOS ::= '^''
NONQUOTE ::= '^"'
NONSTART ::= LETTER | DIGIT | '.' | '-' | '_' | ':'
QUOTE ::= '"'
START ::= LETTER | '_'
TOKEN_AND ::= 'and'
TOKEN_ASTERISK  ::= '*'
TOKEN_ATRATE  ::= '@'
TOKEN_COMMA ::= ','
```

```
TOKEN_DOUBLESLASH  ::= '//'
TOKEN_EQUAL ::=  '='
TOKEN_GREATERTHAN ::=  '>'
TOKEN_GREATERTHANEQUAL ::=  '>='
TOKEN_INTERSECT ::=   'intersect'
TOKEN_LESSTHAN ::= '<'
TOKEN_LESSTHANEQUAL ::= '<='
TOKEN_LPAREN  ::= '('
TOKEN_LSQUARE  ::= '['
TOKEN_MINUS  ::= '-'
TOKEN_NOT ::= 'not'
TOKEN_NOTEQUAL ::= '!='
TOKEN_OR ::= 'or'
TOKEN_PERIOD  ::= '.'
TOKEN_PLUS  ::= '+'
TOKEN_RPAREN  ::= ')'
TOKEN_RSQUARE  ::= ']'
TOKEN_SLASH  ::= '/'
TOKEN_TO ::= 'to'
TOKEN_UNION ::= '|'  | 'union'
TOKEN_ID  ::= START [NONSTART...]
TOKEN_FLOATVAL  ::= DIGITS | '.'DIGITS | DIGITS'.'DIGITS
TOKEN_INT  ::= DIGITS
TOKEN_STRING  ::=
       QUOTE NONQUOTE... QUOTE
      | APOS NONAPOS... APOS
TOKEN_TEXT ::=  'text'
```

XPath operators

This section specifies the XPath subset supported by the XML processor.

XPath basic operators

Table 3-1 shows the supported basic XPath operators.

Table 3-1: XPath basic operators

Operator	Description
/	Path (Children): the child operator ('/') selects from immediate children of the left-side collection.
//	Descendants: the descendant operator ('//') selects from arbitrary descendants of the left-side collection.
*	Collecting element children: an element can be referenced without using its name by substituting the '*' collection
@	Attribute: attribute names are preceded by the '@' symbol
[]	Filter: You can apply constraints and branching to any collection by adding a filter clause '[ ]' to the collection. The filter is analogous to the SQL where clause with any semantics. The filter contains a query within it, called the sub-query. If a collection is placed within the filter, a Boolean “true” is generated if the collection contains any members, and a “false” is generated if the collection is empty.
[n]	Index: index is mainly use to find a specific node within a set of nodes. Enclose the index within square brackets. The first node is index 1.
text()	Selects the text nodes of the current context node.

XPath set operators

Table 3-2 on page 51, shows the supported XPath set operators.

Table 3-2: XPath set operators

Operator	Description
union |	Union: union operator (shortcut is '|') returns the combined set of values from the query on the left and the query on the right. Duplicates are filtered out and resulting list is sorted in document order.
intersect	Intersection: intersect operator returns the set of elements in common between two sets.
( )	Group: you can use parentheses to group collection operators.
. (dot)	Period: dot term is evaluated with respect to a search context. The term evaluates to a set that contains only the reference node for this search context.
Boolean Operators (*and* and *or*)	Boolean expressions can be used within subqueries.
and	Boolean “and”.
or	Boolean “or”.

XPath comparison operators

Table 3-3 shows the supported XPath comparison operators.

Table 3-3: XPath comparison operators

Operator	Description
=	equality
!=	non-equality
<	less than
>	greater than
>=	less than equal
<=	greater than equal

XPath functions

Adaptive Server supports the following XPath string functions:

- toupper
- tolower

- normalize-space
- concat

General guidelines and examples

This section describes general guidelines for using functions in XPath expressions. These guidelines apply to all the functions listed. All these examples use tolower, which returns a single argument in lowercase.

You can use a function call wherever you would use a step expression.

Example 1

Functions used as the top level of an XPath query are called top-level function calls. The following query shows tolower as a top-level function call:

```
select xmlextract
(’tolower(//book[title="Seven Years in Trenton"]//first-name)’, text_doc)
from sample_docs where name_doc=’bookstore’
------------------------------------------
joe
```

The parameters of a top-level function call must be an absolute path expression; that is, the parameter must begin with a slash (/) or a double slash (//).

Example 2

The parameters of a function call can be complex XPath expressions that include predicates. They can also be nested function calls:

```
select xmlextract
('//book[normalize-space(tolower(title))="seven years in trenton"]/author',
text_doc)
from sample_docs where name_doc='bookstore'

-----------------------------------------
<author>
     <first-name>Joe</first-name>
     <last-name>Bob</last-name>
     <award>Trenton Literary Review
     Honorable Mention</award>
</author>
```

Example 3

You can use a function as a relative step, also called a relative function call. The following query shows tolower as a relative function call:

```
select xmlextract
( '//book[title="Seven Years in Trenton"]//tolower(first-name)', text_doc)
from sample_docs where name_doc='bookstore'

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

```
joe
```

This example shows that the parameters of a relative function must be a relative path expression; that is, it cannot begin with a slash (/) or a double slash(//).

Example 4

Both top-level and relative functions can use literals as parameters. For example:

```
select xmlextract( 'tolower("aBcD")' ,text_doc),
    xmlextract( '/bookstore/book/tolower("aBcD")', text_doc)
from sample_docs where name_doc='bookstore'

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

 abcd      abcd
```

Example 5

String functions operate on the text of their parameters. This is an implicit application of text(). For example, this query returns a first-name element as an XML fragment:

```
select xmlextract
( '//book[title="Seven Years in Trenton"]//firstname’, text_doc)
from sample_docs where name_doc='bookstore'

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

<first-name>Joe</first-name>
```

The following query returns the text of that first-name XML fragment:

```
select xmlextract
( '//book[title="Seven Years in Trenton"]//first-name/text()', text_doc)
from sample_docs where name_doc='bookstore'

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

Joe
```

The next query applies tolower to the first-name element. This function operates implicitly on the text of the element:

```
select xmlextract
('//book[title="Seven Years in Trenton"] //tolower(first-name)', text_doc)
from sample_docs where name_doc='bookstore'

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

joe
```

This has the same effect as the next example, which explicitly passes the text of the XML element as the parameter:

```
select xmlextract
( '//book[title="Seven Years in Trenton"]//tolower(first-name/text())',
text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------------
joe
```

Example 6

You apply a relative function call as a step in a path. Evaluating that path produces a sequence of XML nodes, and performs a relative function call for each node.The result is a sequence of the function call results. For example, this query produces a sequence of first_name nodes:

```
select xmlextract( '/bookstore/book/author/first-name', text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------
<first-name>Joe</first-name><first-name>Mary</first-name>
<first-name>Toni</first-name>
```

The query below replaces the last step of the previous query with a call to toupper, producing a sequence of the results of both function calls.

```
select xmlextract('/bookstore/book/author/toupper(first-name)', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------
JOEMARYTONI
```

Now you can use concat to punctuate the sequence of the function results. See the example in “concat” on page 56.

Example 7

tolower, toupper, and normalize-space each have a single parameter. If you omit the parameter when you specify these functions in a relative function call, the current node becomes the implicit parameter. For instance, this example shows a relative function call of tolower, explicitly specifying the parameter:

```
select xmlextract
('//book[title="Seven Years in Trenton"]//tolower(first-name)', text_doc)
from sample_docs where name_doc='bookstore'
-----------------------------------------------------
joe
```

This example of the same query specifies the parameter implicitly:

```
select xmlextract
('//book[title="Seven Years in Trenton"]//first-name/tolower()', text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------------------
joe
```

You can also specify parameters implicitly in relative function calls when the call applies to multiple nodes. For example:

```
select xmlextract('//book//first-name/tolower()', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------------------
joemarymarytoni
```

Functions

This section describes the individual functions that enhance XML Services.

tolower and toupper

Description

tolower and toupper return their argument values in lowercase and uppercase, respectively.

Syntax

tolower(*string-parameter*)

toupper(*string-parameter*)

Example

This example uses toupper to return the argument value in uppercase.

```
select xmlextract
('//book[title=”Seven Years in Trenton”]//toupper(first-name)', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------------
JOE
```

normalize-space

Description

Makes two changes when it returns its argument value:

- It removes leading and trailing white-space characters.
- It replaces all substrings of two or more white-space characters that are not leading characters with a single white-space character.

Syntax

normalize-space(*string-parameter*)

Examples

This example applies normalize-space to a parameter that includes leading and trailing spaces, and embedded newline and tab characters:

```
select xmlextract
('normalize-space(" Normalize space example. ")', text_doc)
from sample_docs where name_doc='bookstore'
------------------------
Normalize space example.
```

normalize-space and tolower or toupper are useful in XPath predicates, when you are testing values whose use of white space and case is not known. The following predicate is unaffected by the case and whitespace usage in the title elements:

```
select xmlextract
('//magazine[normalize-space(tolower(title)="tracking trenton")]//price',
text_doc)
from sample_docs where name_doc='bookstore'
--------------------------
<price>55</price>
```

concat

Description

concat returns the string concatenation of the argument values. It has zero or more parameters.

Syntax

concat(*string-parameter* [,*string-parameter*]...)

Example

concat can return multiple elements in a single call of xmlextract. For example, the following query returns both first-name and last-name elements:

```
select xmlextract('//author/concat(first-name, last-name)', text_doc)
from sample_dcs where name_doc='bookstore'
-----------------------------------------
JoeBobMaryBobToniBob
```

You can also use concat to format and punctuate results. For example:

```
select xmlextract
('//author/concat(",first(",first-name, ")-last(",last-name, ") ")' ,
text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------------------
first(Joe)-last(Bob) first(Mary)-last(Bob) first(Toni)-last(Bob)
```

Parenthesized expressions

Adaptive Server supports parenthesized expressions. This section describes the general syntax of parenthesized expressions in XPath. The following sections describe how to use parentheses with subscripts and unions.

Parentheses and subscripts

Subscripts apply to the expression that immediately precedes them. Use parentheses to group expressions in a path. The examples in this section illustrate the use of parentheses with subscripts.

The following general example, which does not use subscripts, returns all titles in the book element.

```
select xmlextract('/bookstore/book/title', text_doc)
from sample_docs where name_doc='bookstore'
--------------------------------
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Tracking Trenton</title>
<title>Treanton Today, Trenton Tomorrow</title>
<title>Whos Who in Trenton</title>
```

To list only the first title, you can use the “[1]” subscript, and enter this query:

```
select xmlextract
('/bookstore/book/title[1]', text_doc)
from sample_docs where name_doc='bookstore'
------------------------------------------------
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Tracking Trenton</title>
<title>Treanton Today, Trenton Tomorrow</title>
<title>Whos Who in Trenton</title>
```

However, the above query does not return the first title in the bookstore. It returns the first title in each book. Similarly, the following query, which uses the “[2]” subscript, returns the second title of each book, not the second title in the bookstore. Because no book has more than one title, the result is empty.

```
select xmlextract
('/bookstore/book/title[2]', text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------
NULL
```

These queries return the *i*th title in the book, rather than in the bookstore, because the subscript operation (and predicates in general) applies to the immediately preceding item. To return the second title in the overall bookstore, rather than in the book, use parentheses around the element to which the subscript applies. For example:

```
select smlextract
('(/bookstore/booktitle)[2]', text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------------------
<title>History of Trenton</title>
```

You can group any path with parentheses. For example:

```
select xmlextract('(//title)[2]', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------------------
<title>History of Trenton</title>
```

Parentheses and unions

You can also use parentheses to group operations within a step. For example, the following query returns all book titles in the bookstore.

```
select xmlextract('/bookstore/book/title', text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------------
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>
<title>Who's Who in Trenton</title>
```

The above query returns only book titles. To return magazine titles, change the query to:

```
select xmlextract('/bookstore/magazine/title', text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------------
<title>Tracking Trenton</title>
```

To return the titles of all items in the bookstore, you could change the query as follows:

```
select xmlextract('/bookstore/*/title', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Tracking Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>
<title>Whos Who in Trenton</title>
```

If the bookstore contains elements other than books and magazines—such as calendars and newspapers—you can query only for book and magazine titles by using the union (vertical bar) operator, and parenthesizing it in the query path. For example:

```
select xmlextract('/bookstore/(book|magazine)/title', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Tracking Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>
<title>Whos Who in Trenton</title>
```

CHAPTER 4 XML Mapping Functions

This chapter describes the XML mapping functions in detail, and provides examples for them.

for xml clause

Specifies a SQL select statement that returns an XML representation of the result set.

Syntax

```
select ::=
    select [ all | distinct ] select_list
    [into_clause ]
    [where_clause ]
    [group_by_clause ]
    [having_clause ]
    [order_by_clause ]
    [compute_clause ]
    [read_only_clause ]
    [isolation_clause ]
    [browse_clause ]
    [plan_clause]
    [for_xml_clause]
for_xml_clause ::=
    for xml [schema | all] [option option_string] [returns_clause]
option_string ::= basic_string_expression
returns_clause ::=
    returns { char [(integer)] | varchar [(integer)]
```

|*unichar* [(*integer*)] | *univarchar* [(*integer*)]
|*text* | *unitext* | *java.lang.String*}

Note See “option_strings: general format” on page 39 for more information about option strings.

Note See Chapter 6, “XML Support for I18N” on page 113 for more information on using for xml with I18N data.

Description

- The for xml clause is a new clause in SQL select statements. The syntax shown above for select includes all of the clauses, including the for xml clause.
- The syntax and description of the other select statement clauses are in *Sybase Adaptive Server Reference Manual, Volume 2: “Commands.”*
- The for xml clause supports the java.lang.string datatype, represented as string. Any other Java type is represented as objectID.
- **Note** For a description of for xml schema and for xml all, see “for xml schema and for xml all” on page 67.

  The variants of the for xml clause are as follows:

  a If a select statement specifies a for xml clause, refer to the select statement itself as basic select, and the select statement with a for xml select as for xml select. For example, in the statement

  ```
  select 1, 2 for xml
  ```

  the basic select is select 1, 2, and the for xml select is select 1, 2 for xml.

  b A for xml schema select command or subquery has a *for_xml_clause* that specifies schema.

  c A for xml all select command or subquery has a *for_xml_clause* that specifies all.
- A for xml select statement cannot include an into_clause, compute_clause, read_only_clause, isolation_clause, browse_clause, or plan_clause.
- for xml select cannot be specified in the commands create view, declare cursor, subquery, or execute command.

- for xml select cannot be joined in a union, but it can contain unions. For instance, this statement is allowed:

  ```
  select * from T
  union
  select * from U
  for xml
  ```

  But this statement is not allowed:

  ```
  select * from T for xml
  union
  select * from U
  ```

- The value of for xml select is an XML representation of the result of the basic select statement. The format of that XML document is the SQLX format described in Chapter 5, "XML Mappings."

- The returns clause specifies the datatype of the XML document generated by a for xml query or subquery. If no datatype is specified by the returns clause, the default is text.

- The result set that a for xml select statement returns depends on the *incremental* option:

  - *incremental* = *no* returns a result set containing a single row and a single column. The value of that column is the SQLX-XML representation of the result of the basic select statement. This is the default option.

  - *incremental* = *yes* returns a result set containing a row for each row of the basic select statement. If the root option specifies *yes* (the default option), an initial row specifies the opening XML root element, and a final row specifies the closing XML root element.

    For example, these select statements return two, one, two, and four rows, respectively:

    ```
    select 11, 12 union select 21, 22
    select 11, 12 union select 21, 22 for xml
    select 11, 12 union select 21, 22
         for xml option "incremental=yes root=no"
    select 11, 12 union select 21, 22
         for xml option "incremental=yes root=yes"
    ```

- The date and time fields in a datetime value in the results of a for xml query are separated by the delimiter 'T' (letter T) as now specified in the ANSI SQL-XML standard. Without this format, validation fails with standard XML parsers.

For example, if you execute this query in Adaptive Server 12.5.2, the results are:

```
select getdate() for xml
```

```
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
< row >
< C1 > 2008-05-30 11:42:19 < /C1 >
< /row >
< /resultset >
```

But in Adaptive Server 15.0.2, the results from the same query are:

```
select getdate() for xml
```

```
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
< row >
< C1 > 2008-05-30T11:41:42 < /C1 >
< /row >
< /resultset >
```

Options

The general format of the *option_string* is specified in “option_strings: general format” on page 39. The options for the for xml clause are specified in “SQLX options” on page 87.

Exceptions

Any SQL exception raised during execution of the basic select statement is raised by the for xml l select. For example, both of the following statements raise a zero divide exception:

```
select 1/0
select 1/0 for xml
```

Example

The for xml clause:

```
select pub_id, pub_name
from pubs2.dbo.publishers
for xml
----------------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
   <pub_id>0736</pub_id>
   <pub_name>NewAgeBooks</pub_name>
</row>

<row>
   <pub_id>0877</pub_id>
   <pub_name>Binnet & Hardley</pub_name>
```

```
</row>

<row>
   <pub_id>1389</pub_id>
   <pub_name>Algodata Infosystems</pub_name>
</row>

</resultset>
```

for xml subqueries

In Transact-SQL, an expression subquery is a parenthesized subquery. It has a single column, the value of which is the expression subquery result, and must return a single row. You can use an expression subquery almost anywhere you can use an expression. For more information about subqueries, see the *Transact-SQL® User's Guide*.

The for xml subqueries feature allows you to use any subquery containing a for xml clause as an expression subquery.

Syntax

```
subquery ::= select [all | distinct ] select_list
   (select select_list
   [from table_reference [, table_reference]... ]
   [where search_conditions]
   [group by aggregate_free_expression [aggregate_free_expression]...]
   [having search_conditions]
   [for_xml_clause])
for_xml_clause:: = See "for xml schema and for xml all" on page 64
table_reference::= table_view_name |ANSI_join | derived_table
table_view_name::= See SELECT in Vol. 2, "Commands, in the
   "Reference Manual"
ANSI_join::= See SELECT in Vol. 2, "Commands," in the "Reference
   Manual"
derived_table::= (subquery) as table_name
```

Description

- A select command containing a for xml clause generates an XML document that represents the results of the select statement, and returns that XML document as a result set, with a single row and a single column. You can access that result set using normal techniques for processing result sets.
- For a general description of the for xml clause and its *option_string*, see "for xml clause" on page 61. For a description of extensions to the for xml clause that support the SCHEMA keyword and the return clause, see "for xml schema and for xml all" on page 67.
- A for xml subquery is a subquery that contains a for xml clause.

- You can use a for xml subquery as an expression subquery, though there are some differences between them; for example, the following restrictions apply to ordinary expression subqueries, but not to for xml subqueries:
  - No multiple items in the select list
  - No text and image columns in the select list
  - No group by or having clauses
- You cannot specify a for xml subquery within a for xml select or within another for xml subquery.
- You cannot use a for xml subquery in these commands:
  - for xml select
  - create view
  - declare cursor
  - select into
  - as a quantified predicate subquery, such as any/all, in/not in, exists/not exists
- A for xml subquery cannot be a correlated subquery. For more information on correlated subqueries, see the *Transact-SQL User's Guide*.
- A for xml clause that returns a text or unitext datatype cannot be used in a nested scalar subquery.
- The datatype of a for xml subquery is specified by the returns clause of the *for_xml_clause*. If a returns clause specifies no datatype, the default datatype is text.

Exceptions

- Exceptions are the same as those specified for the *for_xml_clause*.
- If the returns clause specifies a datatype to which you cannot convert the result of the subquery, an exception is raised: Result cannot be converted to the specified datatype.

Examples

Example 1
A for xml subquery returns the XML document as a string value, which you can assign to a string column or variable, or pass as an argument to a stored procedure or built-in function. For example:

```
declare @doc varchar(16384)
set @doc = (select * from systypes for xml returns varchar(16384))
select @doc
--------------
```

Example 2 To pass the result of a for xml subquery as a string argument, enter:

```
select xmlextract('//row[usertype = 18]',
     (select * from systypes for xml))
------------
```

Example 3 To specify a for xml subquery as a value in insert or update:

```
create table docs_xml(id integer, doc_xml text)
insert into docs_xml
   select(1, (select * from systypes for xml)
--------

update docs_xml
set doc_xml = (select * from sysobjects for xml)
where id = 1
------------
```

for xml schema and *for xml all*

This section describes additional forms of the for xml clause. You can generate an XML schema, an XML schema and XML DTD, or the XML data document.

Description

- The select statement or subquery with a for xml schema clause produces an XML document, which describes the same SQLX XML result set that would be generated by the select statement if it contained the for xml clause without the schema predicate.
- The result of this for xml subquery is an xml value:

  ```
  (subquery for xml schema option option_string)
  ```

  This xml value is the same as the java.lang.String value result of the following query:

  ```
  forxmlj('subquery', option_string)
  ```

- A select statement or subquery with a for xml all clause produces an XML document that contains the SQLX result set, the XML schema, and the XML DTD that describes that result set. These are contained in a single XML document with the following elements:
  - <*multiple-results*>—The root element
  - <*multiple-results-item type="result-set*">—an element containing:

    <*multiple-results-item-dtd*>—the DTD for the result set

    <*multiple-result-item-schema*>—the XML schema for the result set

<*multiple-result-item-data*>—the result set

Options

The general format of the *option_string* is specified in “option_strings: general format” on page 39. The options for the for xml clause are specified in “SQLX options” on page 87.

Exceptions

The exceptions to extensions are the same as those specified in “SQLX options” on page 87.

Examples—Usage

These examples show uses of for xml schema and for xml all.

Example 1 In this example, a for xml all subquery returns

- the XML schema
- the XML schema and XML DTD
- the result set as an XML document

These are all returned in a string value, which you can either assign to a string column or variable, or pass as a string argument to a stored procedure or function.

```
declare @doc varchar(16384)
set @doc = (select * from systypes for xml all returns varchar(16384))
select @doc
-----------
```

Example 2 This example passes the result of a for xml schema subquery as a string argument:

```
select xmlextract('//row[usertype=18]'
   (select * from systypes for xml all))
---------
```

Example 3 This example specifies a for xml all subquery as a value in an insert or update command:

```
create table docs_xml(id integer, doc_xml xml)
insert into docs_xml
   values(1,(select * from sysobjects for xml all)
where id=1
```

Examples—results

This set of examples shows the results generated by the commands in the examples above.

Example 1 This example shows a basic select for xml statement result.

```
select "a", 1 for xml
-----------
```

```
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>a</C1>
    <C2>1</C2>
  </row>

</resultset>

(1 row affected)
```

Example 2 This examples shows for xml schema, returning the XML schema that describes the result set in Example 1.

```
select "a", 1 for xml schema
--------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:sqlxml="http://www.iso-standards.org/mra/9075/sqlx">

<xsd:import namespace="http://www.w3.org/20001/XMLSchema"
 schemaLocation="http://www.iso-standards.org/mra/9075/sqlx.xsd"/>

<xsd:complexType name="RowType.resultset"
  <xsd:sequence>
   <xsd:element name="C1" type="VARCHAR_1"/>
   <xsd:element name="C2" type="INTEGER"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TableType.resultset"
  <xsd: sequence>
   <xsd:element name="row" type="RowType.resultset"
    minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType

<xsd:simpleType name="VARCHAR_1">
  <xsd:restriction base="xsd:string".
   <xsd:length value="1"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="INTEGER">
  <xsd:restriction base="xsd:integer">
   <xsd:maxInclusive value="2147483647"/>
   <xsd:minInclusive value="-2147483648"/>
  </xsd:restriction>
```

```
</xsd:simpleType>

<xsd:element name+"resultset" type="TableType.resultset"/>
</xsd:schema>

(1 row affected)
```

Example 3 This example of using for xml all returns the schema, DTD, and data for the result set.

```
select 'a', 1 for xml all
-----------
<multiple results>

<multiple-results-item type="result-set">
<multiple-results-item-dtd>

<!DOCTYPE resultset [
<!ELEMENT resultset(row*)>
<!ELEMENT row (C1,C2)>
<!ELEMENT C1(#PCDATA)>
<!ELEMENT C2(#PCDATA)>

]>

</multiple-results-item-dtd>

</multiple-results-item-schema>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmln:sqlxml="http://www.iso-standards.org/mra/9075/sqlx">

 <xsd:import namespace="http://2=www.w3.org/2001/XMLSchema"
  schemaLocation="http://www.iso-standards.org/mra/9075/sqlx.xsd"/>

<xsd:complexType name="RowType.resultset">
 <xsd:sequence>
  <xsd:element name="C1"type="VARCHAR_1" />
  <xsd:element name="C2"type="INTEGER" />
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TableType.resultset">
 <xsd:sequence>
  <xsd:element name="row" type="RowType.resultset"
    minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
```

```
</xsd:complexType>

<xsd:simpleType name="VARCHAR_1">
 <xsd:restriction base="xsd:string">
  <xsd:length value="1"/>
 </xsd:restriction>
</xsd:simpleType

<xsd:element name="resultset" type="TableType.resultset"/>

</xsd:schema>

</multiple-results-item-data>

<multiple-results-item-data>

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

 <row>
  <C1>a</C1>
  <C2>1</C2>
 </row>

</resultset>

</multiple-results-item-data>

</multiple-results-item>

</multiple-results>

(1 row affected)
```

forxmlj, forxmldtdj, forxmlschemaj, forxmlallj

Note The functions in this section are Java-based, and you must install them in your server before you can use them. For instructions see Appendix D, “The Java-Based XQL Processor”.

The Java-based forxml functions map the result set of a SQL query to a SQLX-XML schema, result set document, or both. The SQL query is specified as a character string, containing an arbitrary SQL query expression.

forxmlj is a functional form of the mapping provided by the for xml clause of the select statement. The differences are:

- In some contexts, such as function arguments, update statement set clauses, and insert statement value lists, you can use the forxmlj function but not a select statement with for xml.
- A select statement with a for xml clause returns the result in the datatype specified in the *returns_clause*. The forxmlj function returns the result as java.lang.String.
- A select statement with a for xml clause returns either a single row or multiple rows, depending on the *incremental* option. The forxmlj function returns a single result.

Syntax

```
forxmljfunction ::=
     forxmlj(sql_query_expression, option_string)
     | forxmldtdj(sql_query_expression, option_string)
     | forxmlschemaj(sql_query_expression, option_string)
forxmlallj_procedure::=
    execute forxmlallj
       sql_query_expression, option_string
       rs_target_out, schema_target_out, dtd_target_out
sql_query_expression::=basic_string_expression
option_string::=basic_string_expression
```

Description

- A *basic_string_expression* is a *sql_query_expression* whose datatype is char, varchar, unichar, univarchar, or java.lang.String.
- If any parameter of forxmlj is null, then the result of the call is null.
- If the *sql_query_expression* is an all-blank or empty string, then the result of the call is an empty string.
- The *sql_query_expression* must contain a valid SQL select statement, which can include a from clause, where clause, group by clause, having clause, and order by clause. It cannot include an into clause, compute clause, read_only clause, isolation clause, browse clause, or plan clause.
- forxmlj evaluates the *sql_query_expression* and returns a SQLX-XML document containing the result set, formatted as a SQLX result set.
- forxmldtdj evaluates the *sql_query_expression*, and returns an XML DTD describing the SQLX-XML result set for that query.

- forxmlschemaj evaluates the *sql_query_expression*, and returns a SQLX-XML schema describing the SQL-XML result set for that query.
- The forxmlallj procedure evaluates the *sql_query_expression*, and returns a SQLX-XML result set, schema, and DTD for that query.

Note For a description of the SQLX-XML representation of SQL result sets, see Chapter 5, "XML Mappings."

Options

The general format of the option_string is specified in "option_strings: general format" on page 39. The options for the for xml clause are specified in Chapter 5, "XML Mappings."

Exceptions

Any SQL exception raised during execution of the *sql_query_expression* is raised by the forxmlj function.

Examples

The forxmlj function:

```
set stringsize 16384
select forxmlj
      ("select pub_id, pub_name
         from pubs2.dbo.publishers", "")
----------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
   <pub_id>0736</pub_id>
   <pub_name>New AgeBooks</pub_name>
</row>

<row>
   <pub_id>0877</pub_id>
   <pub_name>Binnet & Hardley</pub_name>
</row>

<row>
   <pub_id>1389</pub_id>
   <pub_name>Algodata Infosystems</pub_name>
</row>

</resultset>
```

The forxmldtdj function:

```
set stringsize 16384
select forxmldtdj
      ("select pub_id, pub_name
         from pubs2.dbo.publishers",
      "tablename=extract nullstyle=omit")
-----------
<!ELEMENT extract (row*)>
<!ELEMENT row (pub_id, pub_name?)>
<!ELEMENT pub_id (#PCDATA)>
<!ELEMENT pub_name (#PCDATA)>
```

The forxmlschemaj function:

```
set stringsize 16384
select forxmlschemaj
     ("select pub_id, pub_name
        from pubs2.dbo.publishers",
     "tablename=extract nullstyle=omit")

<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sqlxml=
   "http://www.iso-standards.org/mra/9075/sqlx">

<xsd:simpleType name="CHAR_4">
   <xsd:restriction base="xsd:string">
      <xsd:length value="4"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="VARCHAR_40">
   <xsd:restriction base="xsd:string">
      <xsd:length value="40"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="RowType.extract">
   <xsd:sequence>
       <xsd:element name="pub_id" type="CHAR_4"
          minOccurs="0" MaxOccurs="1"/>
        <xsd:element name="pub_name" type="VARCHAR_40"
          minOccurs="0" maxOccurs="1"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TableType.extract">
   <xsd:sequence>
       <xsd:element name="row" type="RowType.extract"
         minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
```

```
</xsd:complexType>

<xsd:element name="extract" type="TableType.extract"/>
</xsd:schema>
```

The forxmlallj procedure:

```
set stringsize 16384
declare @rs varchar(16384)
declare @schema varchar(16384)
declare @dtd varchar(16384)
execute forxmlallj
  "select pub_id, pub_name from pubs2.dbo.publishers",
  "name=extract  null=attribute",
  @rs out, @schema    out, @dtd out
------------
<extract
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
   <pub_id>0736</pub_id>
   <pub_name>New Age Books</pub_name>
</row>

<row>
   <pub_id>0877</pub_id>
   <pub_name>Binnet & Hardley</pub_name>
</row>

<row>
   <pub_id>1389</pub_id>
   <pub_name>Algodata Infosystems</pub_name>
</row>
</extract>

<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sqlxml=
   "http://www.iso-standards.org/mra/9075/sqlx">
<xsd:simpleType name="CHAR_4">
   <xsd:restriction base="xsd:string">
      <xsd:length value="4"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="VARCHAR_40">
   <xsd:restriction base="xsd:string">
     <xsd:length value="40"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="RowType.extract">
   <xsd:sequence>
     <xsd:element name="pub_id" type="CHAR_4"
          nullable="true" />
     <xsd:element name="pub_name" type="VARCHAR_40"
          nullable="true" />
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TableType.extract">
   <xsd:sequence>
      <xsd:element name="row" type="RowType.extract"
         minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:elementname="extract" type="TableType.extract">
</xsd:schema>
```

```
<!ELEMENT extract (row*)>
<!ELEMENT row (pub_id, pub_name)>
<!ELEMENT pub_id (#PCDATA)>
<!ELEMENT pub_name (#PCDATA)>
```

forsqlcreatej, forsqlinsertj, forsqlscriptj

Note The functions in this section are Java-based, and you must install them in your server before you can use them.

The Java-based forsql functions map SQLX-XML schema and SQLX-XML result set documents to a SQL script.

- The SQLX-XML schema and result set documents are of the form generated by the forxmlj functions.

- The forsqlschemaj function maps a SQLX-XML schema to a SQL create command, and creates a table suitable for the data described by the SQLX-XML schema.
- The forxmlinsertj function maps a SQLX-XML result set to a sequence of SQL insert commands, and re-creates the data described by the SQLX-XML result set.
- The forxmlscriptj function maps both a SQLX-XML schema and a SQLX-XML result set to a SQL create command and creates a table suitable for the data described by the SQLX-XML schema, and a sequence of SQL insert commands that re-create the data described by the SQLX-XML result set.

Syntax

sqlx_to_sql_script_function ::=
  forsqlcreatej(*sqlx_schema*, *option_string*)
  | forsqlinsertj(*sqlx_resultset*, *option_string*)
  |forsqlscriptj(*sqlx_schema*, *sqlx_resultset*, *option_string*)
sqlx_schema::=*basic_string_expression*
sqlx_resultset::=*basic_string_expression*
option_string::=*basic_string_expression*

Description

- A *basic_string_expression* is a *sql_query_expression* whose datatype is char, varchar, unichar, univarchar,or java.lang.String.
- If any parameter of forsqlcreatej, forsqlschemaj, or forsqlscriptj is null, then the result of the call is null.
- If *sqlx_schema* or *sqlx_resultset* is an all-blank or empty string, then the result of the call is an empty string.
- *sqlx_schema* must contain a valid XML document that contains a SQLX-XML schema.
- *sqlx_resultset* must contain a valid XML document that contains a SQLX-XML result set.
- forsqlcreatej generates a SQL create command to create a SQL table suitable for the data described by *sqlx_schema*.
- forsqlinsertj generates a sequence of SQL insert commands to populate a SQL table with the data of *sqlx_resultset*.

  Because this function operates on a SQLX-XML result set without a corresponding schema, the generated insert commands assume that all of the data is varchar.
- forsqlscriptj generates a SQL create and a sequence of SQL insert commands to populate a SQL table with the data of the *sqlx_resultset*.

Because this function operates on both a SQLX-XML schema and result set, create specifies the column datatypes of *sqlx_schema*, and the insert commands assume those datatypes.

- The scripts generated use quoted identifiers for all identifiers. This does not affect subsequent reference to any regular identifiers.

Options

The general format of the *option_string* is described in “option_strings: general format” on page 39.

The forsqlcreatej, forsqlinsertj, and forsqlscripj functions support the following option, described in the “Exceptions” section, below.

```
xmlerror={exception | null | message}
```

Exceptions

If the value of *sqlx_schema* or *sqlx_resultset* is not valid XML:

- If the explicit or default options specify:

  ```
  xmlerror=exception
  ```

  an exception is raised:

  ```
  invalid XML data
  ```

- If the explicit or default options specify:

  ```
  xmlerror=null
  ```

  a null value is returned.

- If the explicit or default options specify:

  ```
  xmlerror=message
  ```

  a character string containing an XML element containing the exception message is returned. This value is in the form of a SQL comment, so the returned value is valid SQL.

Examples

The forsqlcreatej function:

```
set stringsize 16384
declare @schema varchar(16384)
select @schema = forxmlschemaj(
   "select pub_id, pub_name from pubs2.dbo.publishers",
   "tablename=extract  null=attribute")
select forsqlcreatej(@schema, "")
------------
CREATE TABLE "extract"(
   "pub_id" CHAR(4) null,
   "pub_name" VARCHAR(40) null )
```

The forsqlinsertj function:

```
set stringsize 16384
declare @rs varchar(16384)
select @rs = forxmlj(
   "select pub_id, pub_name from pubs2.dbo.publishers")
select forsqlinsertj(@rs, "")
------------
   --Begin table "resultset"
insert into "resultset"
   ("pub_id", "pub_name")
    values ( '0736', 'New Age Books')
insert into "resultset"
   ("pub_id", "pub_name")
    values ( '0877', 'Binnet & Hardley')
insert into "resultset"
   ("pub_id", "pub_name")
    values ( '1389', 'Algodata Infosystems')
--End table "resultset"
```

The forsqlscriptj function:

```
set stringsize 16384
declare @rs varchar(16384)
declare @schema varchar(16384)
declare @dtd varchar(16384)
execute forxmlallj
   "select pub_id, pub_name from pubs2.dbo.publishers",
   "tablename=extract  null=attribute",
    @rs out, @schema out, @dtd out
declare @script varchar(16384)

select @script = forsqlscriptj(@schema, @rs, "")
select @script
execute ("set quoted_identifier on " + @script )
execute ("select pub_id, pub_name from extract")
execute ("drop table extract")
-----------
(return status = 0)

Return parameters:

*****Values of @rs, @schema, and @dtd omitted********
(1 row affected)
(1 row affected)

CREATE TABLE "extract"(
   "pub_id" CHAR(4) null,
    "pub_name" VARCHAR(40) null)
```

```
--Begin table "extract"
insert into "extract"
   ("pub_id", "pub_name")
    values ( '0736', 'New Age Books')
insert into "extract"
   ("pub_id", "pub_name")
    values ( '0877', 'Binnet & Hardley')
insert into "extract"
   ("pub_id", "pub_name")
    values ( '1389', 'Algodata Infosystems')
--End table "extract"

(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)

pub_id pub_name
------ -------------------
1)  New Age Books
2)  Binnet & Hardley
3)  Algodata Infosystems

(3 rows affected)
```

Using Java functions to map hierarchic XML documents and SQL data

Adaptive Server supports two client-oriented Java-based XML functions for mapping data between SQL tables or result sets and hierarchic XML documents. They are:

- ForXmlTree – maps a set of SQL tables or result sets to a tree-structured XML document.
- OpenXml – extracts repeating data from a tree-structured XML document to a SQL table.

The following sections provide sample data and an overview and examples of how you can use ForXmlTree and OpenXml. For a more detailed description, see *$SYBASE/$SYBASE_ASE/sample/XML/xml-util.{doc, pdf}*.

Sample data and its tree-structured XML representation

SQL data is stored in tables, using foreign-key and primary-key columns to provide the tree-structured relationships between tables. When such data is depicted in XML, the tree-structured relationships are commonly represented with nested elements.

For example, consider tables with the data shown in Table 4-1.

Table 4-1: Sample tables

Table data
depts(dept_id, dept_name)
emps(emp_id, emp_name, dept_id)
emp_phones(emp_id, phone_no
projects(project_id, dept_id)

The tree-structured XML representation of the data in Table 4-1 is:

```
<sample xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<depts>
<dept>
      <dept_id>D123</dept_id>
      <dept_name>Main</dept_name>
       <emps>
           <emp>
               <emp_id>E123</emp_id>
               <emp_name>Alex Allen</emp_name>
               <salary>912.34</salary>
               <phones>
                   <phone><phone_no>510.555.1987</phone_no></phone>
                   <phone><phone_no>510.555.1876</phone_no></phone>
                   <!-- other phone elements for this emp -->
               </phones>
            <!-- Other emp elements for this dept -- >
       </emps>

       <projects>
             <project>
                  <project_id>PABC</project_id>
                  <budget>598.76</budget>
             </project>
             <!-- Other project elements for this dept - ->
         </projects>
     </dept>
    <!-- other dept elements for this set of depts. -->
</depts>
```

```
</sample>
```

Using *ForXmlTree* to map SQL data to hierarchic XML

The Java-based function ForXmlTree maps a set of SQL tables or result sets to a tree-structured XML document. It is based on the for xml clause of the SQL select command, which was introduced in Adaptive Server 12.5.1.

select...for xml performs these tasks:

- Maps a single SQL result set to a single XML document.
- Generates a direct mapping of the SQL result set to XML. For example, if select returns a result set with 1000 rows, each having 20 columns, then the XML document returned by for xml has 1000 elements for the rows, each having 20 elements for the columns.

The Java-based function ForXmlTree:

- Can be invoked in the SQL server, a client command line, or a client or server Java application.
- Maps a collection of results sets to a single tree-structured XML document.
- Requires a <*forxmltree*> specification argument, which describes the desired output tree and the SQL data to be included at each node of the tree.
- Generates a for xml-style mapping of XML data at each node of the output tree-structured XML document.

As a result, you can regard the ForXmlTree capability as a two-dimensional for xml mapping. For example, the following <*forxmltree*> input for ForXmlTree generates the XML document shown in “Sample data and its tree-structured XML representation” on page 81.

```
<!-- A forxmltree spec for depts-emps-phones-projects, with aggregation -->
<forxmltree treename="sample">
<node> <!-- The node element for depts -->
   <query> select * from depts order by dept_id </query>
   <options> tablename=depts rowname=dept </options>
   <link variablename="@dept_id" columnname="dept_id" type="char(11)" />
   <node> <!-- The node element for emps, under depts -->
      <query>
           select emp_id, emp_name, salary from emps e
            where e.dept_id = @dept_id order by emp_id
      </query>
      <options> tablename=emps rowname=emp   </options>
       <link variablename="@emp_id" columnname="emp_id" type="char(6)"/>
         <node> <!-- The node element for phones, under emps -->
           <query>
            select phone_no from emp_phones ep where ep.emp_id = @emp_id
           </query>
           <options> tablename=phones rowname=phone </options>
         </node> <!-- End the node for phones -->
   </node> <!-- End the node for emps --
  <node> <!-- The node element for projects, under dept -->
      <query>
          select project_id, budget from projects p
          where p.dept_id = @dept_id order by project_id
      </query>
      <options> tablename=projects rowname=project </options>
  </node> <!-- End the node for projects -->
</node> <!-- End the node for depts -->
</forxmltree>
```

Using *OpenXml* to map hierarchic XML to SQL

The ForXmlTree function described in “Using ForXmlTree to map SQL data to hierarchic XML” on page 82 maps a collection of SQL tables or result sets to a hierarchic XML document. The OpenXml function reverses this process, and extracts the data for a SQL table from an input XML document.

OpenXml is similar to the xmlextract function, introduced in Adaptive Server 12.5.1, which extracts a specified data value from a given XML document. xmlextract specifies an XML document and a single XPath query expression. It returns the result of applying the XPath query to the XML document.

The Java-based OpenXml function:

- Can be invoked from either a client command line or a client Java application. It is not intended for use in the SQL server.
- Requires arguments that include the specified XML document and a set of options that specify the XPath query that extracts the desired output rows and the Xpath queries that extract the desired columns in each output row.

Thus, you can regard OpenXml as a two-dimensional xmlextract.

OpenXml performs either or both of these actions:

- Generates a SQL script to create and populate a SQL table with the extracted data.
- Executes that script to create the SQL tables with the extracted data.

The following examples assume that the XML document in “Sample data and its tree-structured XML representation” on page 81 is stored in *example-document.xml*.

Example 1

This example shows four client command line calls to extract the depts, emps, emp_phones, and projects tables from the XML document.

```
java jcs.xmlutil.OpenXml -i "file:example-document.xml" \
     -r "file:depts.opt" -o "depts.sql"

java jcs.xmlutil.OpenXml -i "file:example-document.xml" \
       -r "file:emps.opt" -o "emps.sql"

java jcs.xmlutil.OpenXml -i "file:example-document.xml" \
       -r "file:emp-phones.opt" -o "emp-phones.sql"

java jcs.xmlutil.OpenXml -i "file:example-document.xml" \
       -r "file:projects.opt" -o "projects.sql"
```

Example 2

This example shows the contents of the options that the command line calls in Example 1 reference. These options specify the data that the calls for OpenXml should extract, and the SQL table in which they should be stored.

```
-- Content of input file "depts.opt
"tablename='depts_ext'
rowpattern='//dept'
columns=
  '   dept_id char( 4 ) "/@dept_id"
      dept_name varchar(50) "/@dept_name" '

-- Content of input options file "emps.opt"
tablename='emps_ext'
rowpattern='//dept/emps/emp'
columns=
  '   emp_id char( 4 ) "/emp_id/text()"
      emp_name varchar(50) "/emp_name/text()"
       dept_id char(4) "/../../@dept_id"
       salary dec(7,2) "/salary/text()"

'-- Content of input options file "emp-phones.opt"
tablename='emp_phones_ext'
rowpattern='/sample/dept/emps/emp/phone'
columns=  '   emp_id char( 4 ) "/../emp_id/text()"
```

```
       phone_no varchar(20) "/@phone_no" '

--Content of input options file "projects.opt"
tablename='projects_ext'
rowpattern='//dept/projects/project'
columns=
  '   project_id char( 4 ) "/project_id/text()"
      dept_id char(4) "/../../@dept_id"
      budget dec(7,2) "/budget/text()" '
```

Example 3

This example shows the SQL script generated by the first OpenXml call. The script creates and populates a table with the extracted depts table data. Subsequent OpenXml calls, shown in Example 1, generate similar scripts for the emps, emp_phones, and projects data.

```
-- output file depts.sql

create table depts_ext
    (dept_id char( 4 ) null, dept_name varchar(50) null
)

insert into depts_ext values('D123', 'Main')

insert into depts_ext values('D234', 'Auxiliary')

insert into depts_ext values('D345', 'Repair')
```

Java SQLX mappings for multiple result set queries

The select ... for xml statement and the Java-based SQLX mapping functions map a single SQL result set to a SQLX-formatted XML document. Adaptive Server provides a Java-based SQLX mapping function, forxmlmultiplej, that maps multiple result sets of a SQL query to an XML document.

forxmlmultiplej

Description

Maps result sets of a SQL query, that can contain multiple result sets, to an XML document.

Syntax

forxmlmultiplej_function ::=
    forxmlmultiplej(*sql_query_expression*, *option_string*)

Options

See "forxmlj, forxmldtdg, forxmlschemaj, forxmlallj" in Chapter 4, "XML mapping functions," in *XML Services* for a description of *sql_query_expression* and *option_string*.

Usage

- *sql_query_expression* can return multiple result sets, and can contain SQL print commands.
- See "Multiple result sets" in *$SYBASE/$SYBASE_ASE/sample/XML/Using-SQLX-mappings.htm* for examples and a complete description of forxmlmultiplej.

CHAPTER 5 XML Mappings

The for xml clause in select statements and the forxmlj function map SQL result sets to SQLX-XML documents, using the SQLX-XML format defined by the ANSI SQLX standard. This chapter describes the SQLX-XML format and the options supported by both the for xml clause and the forxmlj function.

Note When you use isql to generate an XML document with the for xml clause, the documents you generate may be invalid, due to a leading blankspace added as a column separator by isql.

SQLX options

Note In Table 5-1, underlined words specify the default value.

Table 5-1: Options for SQLX mappings

Option name	Option value	Purpose
binary	hex | base64	Representation of binary. Applies on to forxmlj.
columnstyle	element | attribute	Representation of SQL columns
entitize	yes | no | cond	forxmlj and for xml clause
format	yes | no	Include formatting
header	yes | no | encoding Default value depends on the return type. See Chapter 6, "XML Support for I18N."	Include the XML declaration
incremental	yes | no	Return a single row or multiple rows from a select statement that specifies for xml
multipleentitize	yes | no	forxmlmultiplej and for xml clause
nullstyle	attribute | omit	Representation of nulls with *columnstyle=element*
ncr	non_ascii | non_server | no	forxmlj and for xml clause
prefix	SQL name	Base for generated names. Default value is C.
root	yes | no	Include a root element for the table name
rowname	SQL name	Name of the row element. Default value is row.
schemaloc	quoted string with a URI	*schemalocation* value
statement	yes | no	Include the SQL query
tablename	SQL name	Name of the root element. Default value is resultset.
targetns	quoted string with a URI	*targetnamespace* value (if any)
xsidecl	yes | no	forxmlj and for xml clause

SQLX option definitions

This section defines the SQLX options shown in Table 5-1.

binary={hex | base64}

This option indicates whether to represent columns whose datatype is binary, varbinary, or image with hex or base64 encoding. This choice will depend on the applications you use to process the generated document. Base64 encoding is more compact than hex encoding.

This example shows *binary=hex*, the default option.

```
select forxmlj(“select 0x012131415161718191a1b1c1d1e1f1”,
"binary=hex")
-------------------------------------------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instaXML Servicesce">
   <row>
      <C1>012131415161718191A1B1C1D1E1F1</C1>
   </row>

</resultset>
```

This example shows *binary=base64*:

```
select forxmlj(“select 0x012131415161718191a1b1c1d1e1f1”,
"binary=base64")
--------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
     <C1>ASExQVFhcYGRobHB0eHx</C1>
</row>

</resultset>
```

columnstyle=
{element | attribute}

This option indicates whether to represent SQL columns as elements or attributes of the XML “row” element.

This example shows *columnstyle=element* (the default):

```
select pub_id, pub_name from pubs2..publishers
for xml option "columnstyle=element"
--------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance">

   <row>
     <pub_id>0736</pub_id>
     <pub_name>New Age Books</pub_name>
   </row>

   <row>
     <pub_id>0877</pub_id>
     <pub_name>Binnet & Hardley</pub_name>
   </row>

   <row>
     <pub_id>1389</pub_id>
     <pub_name>Algodata Infosystems</pub_name>
   </row>

</resultset>
```

This example shows *columnstyle=attribute*:

```
select pub_id, pub_name from pubs2..publishers
for xml option  "columnstyle=attribute"
----------------------------------------------------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <row
     pub_id="0736"
     pub_name="New Age Books"
   />
   <row
     pub_id="0877"
     pub_name="Binnet & Hardley"
   />
   <row
     pub_id="1389"
     pub_name="Algodata Infosystems"
   />
</resultset>
```

entitize =
{yes | no | cond}

This option specifies whether to convert reserved XML characters (“<>”, “&”, “ ‘ ”, “ “ “) into XML entities(&lt; &apos &gt; &amp; &quote;), in string columns. Use yes or no to indicate whether you want the reserved characters entitized. cond entitizes reserved characters only if the first non-blank character in a column is not “<“. for xml assumes that string columns whose first character is “<“ are XML documents, and does not entitize them.

For example, this example entitizes all string columns:

```
select 'a<b' for xml option 'entitize=yes'
----------
<resultset>
   <row>
      <C1><a&lt;b</C1>
  </row>
</resultset>
```

This example, however, entitizes no string column:

```
select '<ab>' for xml option 'entitize=no'
-------
<resultset>
   <row>
     <C1><ab></C1>
   </row>
</resultset>
```

This example entitizes string columns that do not begin with “<“:

```
select '<ab>', 'a<b' for xml option 'entitize=cond'
---------
<resultset>
   <row>
     <C1><ab></C1>
     <C2>a&lt;b</C2>
   </row>
</resultset>
```

format={yes | no}

This option specifies whether or not to include formatting for newline and tab characters.

For example:

```
select 11, 12 union select 21, 22
for xml option "format=no"
------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row><C1>11</C1><C2>12</C2></row>
<row><C1>21</C1><C2>22</C2></row>
</resultset>
```

header=
{yes | no | encoding}

This option indicates whether or not to include an XML header line in the generated SQLX-XML documents. The XML header line is as follows:

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

Include such a header line if you use the generated SQLX-XML documents as standalone XML documents. Omit the header line if you combine the generated documents with other XML.

For a description of the encoding option, see “XML Support for I18N” on page 113.

For example:

```
select 1,2 for xml option "header=yes"
-------------------------------------------
<?xml version="1.0" ?>
<resultset xmlns:xsi="http://www.w3.org/2001
    /XMLSchema-instance">
<row>
      <C1>1</C1>
      <C2>2</C2>
   </row>
</resultset>
```

incremental={yes | no}

This option applies only to the for xml clause, not to the forxml function. It specifies which of the following a select statement with a for xml clause returns:

- *incremental=no* – returns a single row with a single column of datatype text, containing the complete SQLX-XML document for the result of the select statement. *incremental=no* is the default option.
- *incremental=yes* – returns a separate row for each row of the result of the select statement, with a single column of datatype text that contains the XML element for that row.
  - If the *root* option is *yes* (the default), the *incremental=yes* option returns two additional rows, containing the opening and closing elements for the *tablename*.
  - If the *root* option is *no,* the *tablename* option (explicit or default) is ignored. There are no two additional rows.

    For example, the following three select statements will return one row, two rows, and four rows, respectively.

    ```
    select 11, 12 union select 21, 22
    for xml option "incremental=no"

    select 11, 12 union select 21, 22
    for xml option "incremental=no root=no"

    select 11, 12 union select 21, 22
    for xml option "incremental=no root=yes"
    ```

multipleentitize=
 {yes | no}

This option applies to for xml all. See the option “Entitize = yes | no” for a discussion of entitization.

ncr=
 {no | non_ascii |
 non_server}

See “Numeric character representation” on page 114.

nullstyle=
{attribute | omit}

This option indicates which of the alternative SQLX representations of nulls to use when the columnstyle is specified or defaults to *columnstyle=element*. The nullstyle option is not relevant when *columnstyle=attribute* is specified.

The *nullstyle=omit* option (the default option) specifies that null columns should be omitted from the row that contains them. The *nullstyle=attribute* option indicates that null columns should included as empty elements with the *xsi:nill=true* attribute.

This example shows the *nullstyle=omit* option, which is also the default:

```
select 11, null union select null, 22
for xml option "nullstyle=omit"
--------------------------------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <row>
        <C1>11</C1>
     </row>
     <row>
         <C2>22</C2>
     </row>

</resultset>
```

This example shows *nullstyle=attribute*:

```
select 11, null union select null, 22
for xml option "nullstyle=attribute"
------------------------------------------------------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <row>
      <C1>11</C1>
       <C2 xsi:nil="true"/>
   </row>
   <row>
       <C1 xsi:nil="true"/>
       <C2>22</C2>
   </row>
</resultset>
```

root= {yes | no}

This option specifies whether the SQLX-XML result set should include a root element for the tablename. The default is *root=yes*. If *root=no*, then the tablename option is ignored.

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

   <row>
      <C1>11</C1>
      <C2>12</C2>
  </row>

   <row>
      <C1>21</C1>
      <C2>22</C2>
   </row>

</resultset>


select 11, 12 union select 21, 22
for xml option "root=no"
-----------------------------------------------

    <row>
      <C1>11</C1>
      <C2>12</C2>
   </row>

   <row>
      <C1>21</C1>
      <C2>22</C2>
   </row>

select forxmlj("select 11, 12 union select 21, 22","root=no")
```

rowname=sql_name

This option specifies a name for the “row” element. The default *rowname* is “row”.

The *rowname* option is a SQL name, which can be a regular identifier or delimited identifier. Delimited identifiers are mapped to XML names as described in “Mapping SQL names to XML names” on page 101.

This example shows *rowname=RowElement*:

```
select 11, 12 union select 21, 22
forxml option "rowname=RowElement"
-------------------------------------------
```

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

   <RowElement>
      <C1>11</C1>
      <C2>12</C2>
   </RowElement>

   <RowElement>
      <C1>21</C1>
      <C2>22</C2>
   </RowElement>

</resultset>
```

schemaloc=uri

This option specifies a URI to be included as the *xsi:SchemaLocation* or *xsi:noNamespaceSchemaLocation* attribute in the generated SQLX-XML document. This option defaults to the empty string, which indicates that the schema location attribute should be omitted.

The schema location attribute acts as a hint to schema-enabled XML parsers. Specify this option for a SQLX-XML result set if you know the URI at which you will store the corresponding SQLX-XML schema.

If the *schemaloc* option is specified without the *targetns* option, then the *schemaloc* is placed in the *xsi:noNamespaceSchemaLocation* attribute, as in the following example:

```
select 1,2
for xml option "schemaloc='http://thiscompany.com/schemalib' "
------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation=
     "http://thiscompany.com/schemalib">
   <row>
     <C1>1</C1>
     <C2>2</C2>
   </row>
</resultset>
```

If the *schemaloc* option is specified with the *targetns* option, the *schemaloc* is placed in the *xsi:schemaLocation* attribute, as in the following example:

```
select 1,2
for xml option "schemaloc='http://thiscompany.com/schemalib'
        targetns='http://thiscompany.com/samples'"
-----------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
```

```
        /XMLSchema-instance"
   xsi:schemaLocation="http://thiscompany.com/schemalib"
xmlns="http://thiscompany.com/samples">

   <row>
      <C1>1</C1>
      <C2>2</C2>
   </row>

</resultset>
```

statement={yes | no}

This option specifies whether or not to include a statement attribute in the root element. If root=no is specified, the statement option is ignored.

```
select name_doc from sample_doc
where name_doc like "book%"
for xml option "statement=yes"
--------------------------------------------------
<resultset statement="select name_doc
  from sample_docs where name_doc like &quot;book%&quot;"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <row>
      <name_doc>bookstore</name_doc>
   </row>
</resultset>
```

tablename=sql_name

This option specifies a name for the result set. The default *tablename* is “resultset”.

The *tablename* option is a SQL name, which can be a regular identifier or delimited identifier. Delimited identifiers are mapped to XML names as described in “Mapping SQL names to XML names” on page 101.

This example shows *tablename=SampleTable*.

```
select 11, 12 union select 21, 22
for xml option "tablename=SampleTable"
--------------------------------------------------
<SampleTable xmlns:xsi="http://www.w3.org/2001
           /XMLSchema-instance">

   <row>
      <C1>11</C1>
      <C2>12</C2>
   </row>

   <row>
      <C1>21</C1>
```

```
                <C2>22</C2>
             </row>

          </SampleTable>
```

targetns=uri

This option specifies a URI to be included as the *xmlns* attribute in the generated SQLX-XML document. This option defaults to the empty string, which indicates that the *xmlns* attribute should be omitted. See the schemaloc attribute for a description of the interaction between the *schemaloc* and *targetns* attributes.

```
select 1,2
for xml
option "targetns='http://thiscompany.com/samples'"
---------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://thiscompany.com/samples">
   <row>
      <C1>1</C1>
      <C2>2</C2>
   </row>

</resultset>
```

xsidecl={yes | no}

This option allows you to specify whether to declare the XML xsi attribute.

For example:

```
select 1 for xml option 'xsidecl=yes'
--------
  <resultset
    xmlns:xsi="http://www.w3.org/2001/XMLScainstance">
     <row>
         <C1>1</C1>
     </row>
  </resultset>


select 1 for xml option 'xsidecl=no'
-------
<resultset>

   <row>

      <C1>1</C1>

   </row>
```

Use the xsi attribute for null values in nullstyle=attribute:

```
select null for xml
     option 'nullstyle=attribute xmldecl=yes'
```

```
   If you specify xsidecl=no or <resultset
     xmlns:xsi="http://www.w3.org/2001
     /XMLSchema-instance">
        <row>
            <C1 xsi:nil="true"/>
        </row>
   </resultset>
```

If you specify either nullstyle=element or nullstyle=attribute, and you plan to embed the resulting XML document in a larger XML document already containing a declaration of the xsi attribute, you can specify xsidecl=no.

SQLX data mapping

This section describes the SQLX-XML format used by the documents generated by both the for xml clause in select statements and by the forxmlj function. The SQLX-XML format is specified by the ANSI SQLX standard.

Mapping duplicate column names and unnamed columns

The following query returns two columns with the same name, and three columns with no name:

```
select t1.title_id, t2.title_id, t2.advance-t1.advance,
t1.price*t1.total_sales, t2.price*t2.total_sales
from pubs2..titles t1, pubs2..titles t2
where t1.price=t2.price and t2.advance-t1.advance>3000
title_id title_id
------   ------    --------   ---------   ---------

BU2075   MC3021    4,875.00   55,978.78   66,515.54
MC2222   BU1032    5,000.00   40,619.68   81,859.05
MC2222   BU7832    5,000.00    40,619.68   81,859.05
```

When this data is mapped to XML, the columns become elements or attributes (depending on the *columnstyle* option), and such elements and attributes must have unique names. The generated XML therefore adds integer suffixes to duplicate column names, and generates unique suffixed names for unnamed columns. For example (using the above query):

```
select t1.title_id, t2.title_id, t2.advance-t1.advance,
t1.price*t1.total_sales, t2.price*t2.total_sales
from pubs2..titles t1, pubs2..titles t2
where t1.price=t2.price and t2.advance-t1.advance>3000
for xml
----------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
         /XMLSchema-instance">

   <row
      <title_id1>BU2075</title_id1>
      <title_id2>MC3021</title_id2>
      <C1>4875.00</C1>
      <C2>55978.78</C2>
      <C3>66515.54</C3>
   </row>

   <row>
      <title_id1>MC2222</title_id1>
      <title_id2>BU1032</title_id2>
      <C1>5000.00</C1>
      <C2>40619.68</C2>
      <C3>81859.05</C3>
   </row>

   <row>
      <title_id1>MC2222</title_id1>
      <title_id2>BU7832</title_id2>
      <C1>5000.00</C1>
      <C2>40619.68</C2>
      <C3>81859.05</C3>
   </row>

</resultset>
```

If the name XML generates for an unnamed column corresponds to an existing column name, that generated name is skipped. In the following example, the last of the unnamed columns has the explicit column name “C1”, so “C1” is not used as a generated column name:

```
select t1.title_id, t2.title_id, t2.advance-t1.advance,
t1.price*t1.total_sales,t2.price*t2.total_sales as C1
from pubs2..titles t1, pubs2..titles t2
where t1.price=t2.price and t2.advance-t1.advance>3000
for xml
-----------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
```

```
        /XMLSchema-instance">

<row>
      <title_id1>BU2075</title_id1>
      <title_id2>MC3021</title_id2>
      <C2>4875.00</C2>
      <C3>55978.78</C3>
      <C1>66515.54</C1>
</row>

<row>
      <title_id1>MC2222</title_id1>
      <title_id2>BU1032</title_id2>
      <C2>5000.00</C2>
      <C3>40619.68</C3>
      <C1>81859.05</C1>
</row>

<row>
      <title_id1>MC2222</title_id1>
      <title_id2>BU7832</title_id2>
      <C2>5000.00</C2>
      <C3>40619.68</C3>
      <C1>81859.05</C1>
</row>

</resultset>
```

In the previous examples, the names generated for unnamed columns have the form “C1”, “C2”, and so on. These names consist of the base name “C” and an integer suffix. You can specify an alternative base name with the *prefix* option.

This example shows *prefix=’column_’*:

```
select t1.title_id, t2.title_id, t2.advance-t1.advance,
t1.price*t1.total_sales, t2.price*t2.total_sales
from pubs2..titles t1, pubs2..titles t2
where t1.price=t2.price and t2.advance-t1.advance>3000
for xml option "prefix=column_"
---------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
          /XMLSchema-instance">
   <row>
      <title_id1>BU2075</title_id1>
      <title_id2>MC3021</title_id2>
      <column_1>4875.00</column_1>
      <column_2>55978.78</column_2>
      <column_3>66515.54</column_3>
   </row>

   <row>
      <title_id1>MC2222</title_id1>
      <title_id2>BU1032</title_id2>
      <column_1>5000.00</column_1>
      <column_2>40619.68</column_2>
      <column_3>81859.05</column_3>
   </row>

   <row>
      <title_id1>MC2222</title_id1>
      <title_id2>BU7832</title_id2>
      <column_1>5000.00</column_1>
      <column_2>40619.68</column_2>
      <column_3>81859.05</column_3>
   </row>

</resultset>
```

Mapping SQL names to XML names

The SQLX representation of SQL tables and result sets uses the SQL names as XML element and attribute names. However, SQL names can include various characters that are not valid in XML names. In particular, SQL names include “delimited” identifiers, which are names enclosed in quotes. Delimited identifiers can include arbitrary characters, such as spaces and punctuation. For example:

```
"salary + bonus: "
```

is a valid SQL delimited identifier. The SQLX standard therefore specifies mappings of such characters to valid XML name characters.

The objectives of the SQLX name mappings are:

- To handle all possible SQL identifiers
- To make sure there is an inverse mapping that can regenerate the original identifier

The SQLX name mapping is based on the Unicode representation of characters. The basic convention of the SQLX name mapping is that an invalid character whose Unicode representation is:

```
U+nnnn
```

is replaced with a string of characters of the form:

```
_xnnnn_
```

The SQLX mapping of an invalid name character prefixes the 4 hex digits of the Unicode representation with:

```
_x
```

and suffixes them with an underscore.

For example, consider the following SQL result set:

```
set quoted_identifier on
select 1 as "a + b < c & d", 2 as "<a xsi:nill=""true"">"
----------------------

a + b < c & d <a xsi:nill="true">
------------- --------------------
             1                   2
```

The select list in this example specifies values that are constants (1 and 2), and specifies column names for those values using as clauses. Those column names are delimited identifiers, which contain characters that are not valid in XML names.

The SQLX mapping of that result set looks like this:

```
set quoted_identifier on
select 1 as "a + b < c & d", 2 as "<a xsi:nill=""true"">"
for xml
------------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
        /XMLSchema-instance">

<row>
<a_x0020__x002B__x0020_b_x0020__x003C__x0020_c_x0020__x0026__x0020_d_x0020_>
1
</a_x0020__x002B__x0020_b_x0020__x003C__x0020_c_x0020__x0026__x0020_d_x0020_>
<_x003C_a_x0020_xsi_x003A_nill_x003D__x0022_true_x0022__x003E_>
2
</_x003C_a_x0020_xsi_x003A_nill_x003D__x0022_true_x0022__x003E_></row>

</resultset>
```

The resulting SQLX result set is not easily readable, but the SQLX mappings are intended for use mainly by applications.

The *_xnnnn_* convention handles most SQLX name-mapping considerations.

One further requirement, however, is that XML names cannot begin with the letters “XML”, in any combination of uppercase or lowercase letters. The SQLX name-mapping therefore specifies that the leading “x” or “X” in such names is replaced by the value *_xnnnn_*. The “M” and “L” (in either upper or lower case) are unchanged, since substituting the initial “X” alone masks the phrase “XML”.

For example:

```
select 1 as x, 2 as X, 3 as X99, 4 as xML, 5 as XmLdoc
forxml
------------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
        /XMLSchema-instance">

   <row>
     <x>1</x>
     <X>2</X>
     <X99>3</X99>
     <_x0078_ML>4</_x0078_ML>
     <_x0058_mLdoc>5</_x0058_mLdoc>
   </row>

</resultset>
```

The requirements in mapping SQL names to XML names also apply to the SQL names specified in the *tablename*, *rowname*, and *prefix* options. For example:

```
select 11, 12 union select 21, 22
for xml option "tablename='table @ start' rowname=' row & columns '
       prefix='C '"
----------------------------------------------------
<table_x0020__x0040__x0020_start xmlns:xsi="http://www.w3.org/2001
          /XMLSchema-instance">

<_x0020_row_x0020__x0026__x0020_columns_x0020_>
    <C_x0020_1>11</C_x0020_1>
    <C_x0020_2>12</C_x0020_2>
</_x0020_row_x0020__x0026__x0020_columns_x0020_>

<_x0020_row_x0020__x0026__x0020_columns_x0020_>
    <C_x0020_1>21</C_x0020_1>
    <C_x0020_2>22</C_x0020_2>
</_x0020_row_x0020__x0026__x0020_columns_x0020_>

</table_x0020__x0040__x0020_start>
```

Mapping SQL values to XML values

The SQLX representation of SQL result sets maps the values of columns to the values of the XML attributes or elements that represent the columns.

Numeric values

Numeric datatypes are represented as character string literals in the SQLX mapping. For example:

```
select 1, 2.345, 67e8 for xml
-----------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
       /XMLSchema-instance">

  <row>
      <C1>1</C1>
      <C2>2.345</C2>
      <C3>6.7E9</C3>
  </row>

</resultset>
```

Character values

Character values contained in char, varchar, or text columns require additional processing. Character values in SQL data can contain characters with special significance in XML: the quote ("), apostrophe ('), less-than (<), greater-than (>), and ampersand (&) characters. When SQL character values are represented as XML attribute or element values, they must be replaced by the XML entities that represent them: @quot; &apos;, &lt;, &gt;, and &amp;.

The following example shows a SQL character value containing XML markup characters. The character literal in the SQL select command doubles the apostrophe, using the SQL convention governing embedded quotes and apostrophes.

```
select '<name>"Baker''s"</name>'
----------------------
<name>"Baker's"</name>
```

The following example shows SQLX mapping of that character value, with the XML markup characters replaced by their XML entity representations. The character literal argument in the forxmlj function doubles the embedded quotes.

```
select '<name>"Baker''s"</name>' for xml
-------------------------------------------------
```

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

<row>
<C1>&lt;name&gt;&quot;Baker&apos;s&quot;&lt;/name&gt;<
      /C1>
</row>

</resultset>
```

Binary values

Binary values contained in binary, varbinary, or image columns are represented in either hex or base64 encoding, depending on the option *binary={hex|base64}*. The base64 encoding is more compact. The choice between the two representations depends on the applications that process the XML data.

See the examples in “SQLX options” on page 87.

SQLX schema mapping

The forxmlschemaj function and the forxmlallj functions generate an XML schema that describes the SQLX-XML document for a specified result set. This section provides a general overview of such generated XML schemas. These XML schemas are generally used only by XML tools, so you need not understand each line in detail.

Overview

The following SQL result set has 5 columns, whose datatypes are respectively varchar(3), numeric(3,1), varbinary(2), numeric(3,1), and numeric(3,2).

```
select 'abc', 12.3, 0x00, 45.6, 7.89
--- ------ ---- ------ ------
abc   12.3 0x00   45.6   7.89
```

The SQLX-XML result set for this data is:

```
select forxmlj("select 'abc', 12.3, 0x00, 45.6, 7.89", "")
```

```
--------------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
       /XMLSchema-instance">
   <row>
      <C1>abc</C1>
      <C2>12.3</C2>
      <C3>00</C3>
      <C4>45.6</C4>
      <C5>7.89</C5>
   </row>

</resultset>
```

The SQLX-XML schema describing this document is:

```
select forxmlschemaj("select 'abc', 12.3, 0x00, 45.6, 7.89", "")
-------------------------------------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sqlxml="http://www.iso-standards.org/mra/9075/sqlx">

   <xsd:import namespace="http://www.w3.org/2001/XMLSchema"
      schemaLocation="http://www.iso-standards.org/mra/9075/sqlx.xsd" />

   <xsd:complexType name="RowType.resultset">
     <xsd:sequence>
        <xsd:element name="C1" type="VARCHAR_3" />
        <xsd:element name="C2" type="NUMERIC_3_1" />
        <xsd:element name="C3" type="VARBINARY_2" />
        <xsd:element name="C4" type="NUMERIC_3_1" />
        <xsd:element name="C5" type="NUMERIC_3_2" />
     </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="TableType.resultset">
     <xsd:sequence>
       <xsd:element name="row" type="RowType.resultset"
           minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
   </xsd:complexType>

   <xsd:simpleType name="VARCHAR_3">
     <xsd:restriction base="xsd:string">
         <xsd:length value="3"/>
     </xsd:restriction>
   </xsd:simpleType>
   <xsd:simpleType name="NUMERIC_3_1">
     <xsd:restriction base="xsd:decimal">
        <xsd:totalDigits value="3"/>
        <xsd:fractionDigits value="1"/>
     </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="VARBINARY_2">
     <xsd:restriction base="xsd:hexBinary">
        <xsd:length value="2"/>
     </xsd:restriction>
     </xsd:simpleType>

   <xsd:simpleType name="NUMERIC_3_2">
     <xsd:restriction base="xsd:decimal">
        <xsd:totalDigits value="3"/>
        <xsd:fractionDigits value="2"/>
     </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="resultset" type="TableType.resultset"/>

</xsd:schema>
```

This XML schema has five components:

- In the last part of this sample XML schema are three *xsd:simpleType*elements, which declare simple XML types for the four distinct datatypes in the XML document. These *simpleType* declarations specify the XML base type for each type, and specify *xsd:restriction* elements that define the length characteristics of the SQL data. Each *simpleType* declarations has an XML name: VARCHAR_3, NUMERIC_3_1, VARBINARY_2, and NUMERIC_3_2.
- The XML schema contains a separate *xsd:simpleType* for each distinct attribute combination of SQL datatype, length, and precision. For instance, there are separate types for NUMERIC_3_1 and NUMERIC_3_2. However, there is only one *xsd:simpleType* declaration for NUMERIC_3_1, even though there are two columns with that type. The element declarations for those columns both reference the same simple type name, NUMERIC_3_1.
- The first part of the example XML schema is an *xsd:complexType* for the row type, which defines an element for each column. Each of those element declarations specifies the datatype of the element with the simple type name described above.

- The middle part of the example XML schema is an *xsd:complexType* for the result set, declaring it to be a sequence of row elements whose type is the previously defined row type.
- Finally, the very last line of the example XML schema declares the root element of the result set document.

Option: *columnstyle=element*

The format of a generated XML schema for *columnstyle=element* specifies the columns as XML *elements* of the rowtype declaration. For example:

```
select forxmlschemaj("select 1,2", "columnstyle=element")
-----------------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sqlxml="http://www.iso-standards.org/mra/9075/sqlx">
<xsd:import namespace="http://www.w3.org/2001/XMLSchema"
   schemaLocation="http://www.iso-standards.org/mra/9075/sqlx.xsd" />

   <xsd:complexType name="RowType.resultset">
     <xsd:sequence>
        <xsd:element name="C1" type="INTEGER" />
        <xsd:element name="C2" type="INTEGER" />
     </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="TableType.resultset">
      <xsd:sequence>
         <xsd:element name="row" type="RowType.resultset"
             minOccurs="0" maxOccurs="unbounded"/>

   </xsd:sequence>
   </xsd:complexType>

   <xsd:simpleType name="INTEGER">
      <xsd:restriction base="xsd:integer">
          <xsd:maxInclusive value="2147483647"/>
           <xsd:minInclusive value="-2147483648"/>
      </xsd:restriction>
  </xsd:simpleType>

  <xsd:element name="resultset" type="TableType.resultset"/>

</xsd:schema>
```

Option: *columnstyle=attribute*

The format of a generated XML schema for *columnstyle=attribute* is similar to the XML schema for *columnstyle=element*. The only difference is that the columns are specified as XML *attributes* of the rowtype declaration. For example:

```
select forxmlschemaj("select 1,2", "columnstyle=attribute")
-----------------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:sqlxml="http://www.iso-standards.org/mra/9075/sqlx">

<xsd:import namespace="http://www.w3.org/2001/XMLSchema"
   schemaLocation="http://www.iso-standards.org/mra/9075/sqlx.xsd" />

   <xsd:complexType name="RowType.resultset">

      <xsd:attribute name="C1" type="INTEGER" use="required"/>
      <xsd:attribute name="C2" type="INTEGER" use="required"/>

  </xsd:complexType>

   <xsd:complexType name="TableType.resultset">
   <xsd:sequence>
      <xsd:element name="row" type="RowType.resultset"
          minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:simpleType name="INTEGER">
     <xsd:restriction base="xsd:integer">
        <xsd:maxInclusive value="2147483647"/>
         <xsd:minInclusive value="-2147483648"/>
     </xsd:restriction>
  </xsd:simpleType>

  <xsd:element name="resultset" type="TableType.resultset"/>

</xsd:schema>
```

Option: nullstyle=omit

The format of a generated XML schema for *nullstyle=omit* specifies the *minOccurs="0"* and *maxOccurs="1"* attribute in each nullable columns declaration. For example:

```
select forxmlschemaj("select 1,null", "nullstyle=omit")
-----------------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:sqlxml="http://www.iso-standards.org/mra/9075/sqlx">

 <xsd:import namespace="http://www.w3.org/2001/XMLSchema"
 schemaLocation="http://www.iso-standards.org/mra/9075/sqlx.xsd" />

   <xsd:complexType name="RowType.resultset">
      <xsd:sequence>
         <xsd:element name="C1" type="INTEGER" />
         <xsd:element name="C2" type="INTEGER"
             minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="TableType.resultset">
      <xsd:sequence>
          <xsd:element name="row" type="RowType.resultset"
           minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:simpleType name="INTEGER">
      <xsd:restriction base="xsd:integer">
         <xsd:maxInclusive value="2147483647"/>
          <xsd:minInclusive value="-2147483648"/>
       </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="resultset" type="TableType.resultset"/>

</xsd:schema>
```

Option: *nullstyle=attribute*

The format of a generated XML schema for *nullstyle=attribute* specifies the *nullable="true"* attribute in each nullable columns declaration. For example:

```
select forxmlschemaj("select 1,null", "nullstyle=attribute"
-----------------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sqlxml="http://www.iso-standards.org/mra/9075/sqlx">

  <xsd:import namespace="http://www.w3.org/2001/XMLSchema"
```

```
   schemaLocation="http://www.iso-standards.org/mra/9075/sqlx.xsd" />

<xsd:complexType name="RowType.resultset">
      <xsd:sequence>
         <xsd:element name="C1" type="INTEGER" />
         <xsd:element name="C2" type="INTEGER" nullable="true"/>
      </xsd:sequence>
   </xsd:complexType><

   <xsd:complexType name="TableType.resultset">
      <xsd:sequence>
         <xsd:element name="row" type="RowType.resultset"
            minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:simpleType name="INTEGER">
      <xsd:restriction base="xsd:integer
         <xsd:maxInclusive value="2147483647"/>
          <xsd:minInclusive value="-2147483648"/>
       </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="resultset" type="TableType.resultset"/>

</xsd:schema>
```

CHAPTER 6 XML Support for I18N

This chapter discusses the extension of XML Services to support non-ASCII data. This is necessary both to support XML standards that specify a Unicode base and to support XML-based applications across multiple languages.

In this document the term “I18N” stands for internationalization, which begins with “I” and 18 characters later ends in “n.” This term refers to support for Unicode and other characters beyond the ASCII set.

Overview

The I18N extensions fall into three categories:

- I18N support in the for xml clause, to generate documents that contain non-ASCII data.
- I18N in xmlparse, to store documents containing non-ASCII data.
- I18N in xmlextract and xmltest, to process XML documents and queries containing non-ASCII data.

Unicode datatypes

The following terms refer to categories of datatypes used for Unicode:

- “String datatypes” refers to char, varchar, text, and java.lang.String.

- "Unicode datatypes" refers to unichar, univarchar, unitext, and java.lang.String.
- "String/Unicode columns" refers to columns whose datatypes are "string datatypes" or "Unicode datatypes."

Surrogate pairs

"Surrogate pairs" refers to the pair of 16-bit values that Unicode uses to represent any character that may require more than 16 bits.

Most characters are represented within the range [0x20, 0xFFFF], and can be represented with a single 16-bit value. A surrogate pair is a pair of 16 bit values that represent a character in the range [0x010000..0x10FFFF]. See "Example 7" on page 120 for more details.

Numeric character representation

Numeric Character Representation (NCR) is a technique for representing arbitrary characters in ASCII hexidecimal notation in XML documents. For example, the NCR representation of the Euro sign "€" is "&#x20AC;". This notation is similar to the SQL hexidecimal character notation, u&'\20ac'.

Client-server conversions

Unicode data in the server can be:

- UTF-16 data, stored in unichar, univarchar, unitext, and *java.lang.String*.
- UTF-8 data, stored in char, varchar, and text, when the server character set is UTF-8.

Transferring data between client and server Any one of the three following techniques transfers univarchar or unitext data between client and server:

- Use CTLIB, or, or BCP. Transfer the data as a bit string. The client data is UTF-16, and byte order is adjusted for client-server differences.
- Use ISQL or BCP. Specify "-J UTF-8". The data is converted between the client UTF-8 and the server UTF-16.

- Use Java. Specify the client character set (whether source or target) in data transfers. You can specify UTF-8, UTF-16BE, UTF-16L, UTF-16LE, UTF-16 (with BOM), US-ASCII, or another client character set.

Note If you want to store Unicode XML documents through JDBC, you must mention the connection property 'DISABLE_UNICODE_SENDING', a “false” property that allows you to send Unicode data from the JDBC connection to Adaptive Server.

Techniques for specifying the character set of client files, whether input or output, in client Java applications appear in Java applications in the following sample directory.

```
    $SYBASE/$SYBASE_ASE/sample/JavaXml/JavaXml.zip
```

This directory also supplies the documents *Using-SQLX-mappings*, section *Unicode and SQLX result set documents*.

Character sets and XML data

If you store an XML document in a string column or variable, XML Services assumes that document to be in the server character set. If you store it in a Unicode column or variable, XML Services assumes it to be UTF-16. Any encoding clause in the XML document is ignored

I18N in *for xml*

This section discusses extending the for xml clause to handle non-ASCII data.

You can specify Unicode columns and string columns containing non-ASCII characters in the *select_list* of the for xml clause.

The default datatype in the returns clause is text.

The resulting XML document is generated internally as a Unicode string and converted, if necessary, to the datatype of the returns clause.

For detailed documentation of this clause, see “for xml clause” on page 61.

Option strings

The option string of a for xml clause can specify a u& form of literal, and then contain the SQL notation for characters. Then you can specify Unicode characters for the rowname, tablename, and prefix options. For example, enter:

```
select * from T
for xml
options u&'tablename = \0415\0416 rowname =
           \+01d6d prefix = \0622'
```

If a specified tablename, rowname, or prefix option contains characters that are not valid in simple identifiers, you must specify the option as a quoted identifier. For example, enter:

```
select * from T
for xml
optons u&'tablename = "chars\0415 and \0416"
     rowname = "\+01d6d1 & \+01d160"
     prefix = "\0622-"'
```

Numeric Character Representation in *for xml*

The *option_string* of a select for xml statement includes an ncr option that specifies the representation of string and Unicode columns:

```
ncr = {no | non_ascii | non_server}
```

- *ncr = no* specifies that string and Unicode columns are represented as plain values. These plain values are entitized or not entitized according to the entitize option.
- *ncr = non_ascii* and *ncr = non_server* specify that string and Unicode columns that are, respectively, non-ASCII or not members of the default server character set are represented as NCRs. Any characters not converted to NCRs are either entitized or not, according to the entitize option.

The default NCR option in the for xml clause is ncr = non_ascii.

The ncr option applies only to column values, not to column names or to names specified in the tablename, rowname, or prefix options. XML does not allow NCRs in element or attribute names.

header option

The header option of the for xml clause is extended with a new encoding value:

```
header = {yes | no| encoding}
```

With *header=encoding*, the header is:

```
<?xml version = "1/0" encoding = "UTF-16?">
```

Using the encoding value indicates that the XML header should be included, and that it should contain an XML encoding declaration.

The default header option is *no* if:

- The returns datatype is a Unicode datatype
- The ncr option is *non-ascii*
- The server character set is ISO1, ISO8859_15, ascii_7, or UTF-8.

Otherwise, the default header option is encoding.

Exceptions

None.

Examples

Use the example table generated by the following commands for all the examples following.

```
create table example_I18N_table (name varchar(10) null,
  uvcol univarchar(10) null)
------------------
insert into example_I18N_table values('Arabic',
  u&'\622\623\624\625\626')

insert into example_I18N_table values('Hebrew',
  u&'\5d2\5d3\5d4\5d5\5d6')

insert into example_I18N_table values('Russian',
  u&'\410\411\412\413\414')
```

The example table in Figure 6-1 has two columns:

- A varchar column indicating a language.

- A univarchar column with sample characters of that language. The sample characters consist of strings of consecutive letters.

```
select * from example_I18N_table
name        uvcol
---------------------------------
Arabic      0x06220623062406250626
Hebrew      0x05d205d305d405d505d6
Russian     0x04100411041204130414

(3 rows affected)
```

Example 1

A select command with no variables specified displays the table:

```
select * from example_I18N_table
name        uvcol
--------    ----------------------------
Arabic      0x06220623062406250626
Hebrew      0x05d205d305d405d505d6
Russian     0x04100411041204130414
3 rows affected)
```

Example 2

To generate a SQL XML document using a for xml clause, enter:

```
select * from example_I18N_table for xml
---------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <name>Arabic</name>
    <uvcol>&#x622;&#x623;&#x624;&#x625;&#x626;</uvcol>
  </row>
  <row>
    <name>Hebrew</name>
    <uvcol>&#x5d2;&#x5d3;&#x5d4;&#x5d5;&#x5d6;</uvcol>
  </row>
  <row>
   <name>Russian</name>
   <uvcol>&#x410;&#x411;&#x412;&#x413;&#x414;</uvcol>
  </row>}
</resultset>
```

Example 3

By default, the generated SQLX XML document represents the non-ASCII characters with NCRs. If you set the character set property of your browser to Unicode, the document displays the actual non-ASCII characters, respectively Arabic, Hebrew, or Russian, or any non-ASCII characters you select.

If the browser's character set property is not set to Unicode, the Arabic, Hebrew, and Russian characters appear as question marks.

Example 4

If you want the SQLX XML document to contain non-ASCII as plain characters, specify *no* in the ncr option.

```
select * from example_I18N_table for xml
   option 'ncr=no' returns unitext
-----------------------------------------------------
0x000a003c00720065007300075006c00740073006500740020078006d...etc
```

Example 5

If you retrieve the Unicode document generated in Example 3 into a client file, specifying UTF-16 or UTF-8 as the target character set, you can display it in a browser. It will then show the actual non-ASCII characters you select.

Example 6

The options *ncr=non_ascii* and *ncr=non_server* in ncr translate a character to an NCR only if it is either not ASCII or not in the default server character set. In this example, the expression concatenates ASCII string values with both the ASCII name column and the Unicode uvcol column. The result of this expression is a string that contains both ASCII and non-ASCII characters. In the generated SQLX XML document, only non-ASCII characters are translated to NCRs:

```
select name + '(' + uvcol + ')'  from example_I18N_table2>
   for xml option 'ncr=non_ascii'
--------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
   <C1>Arabic(&#x622;&#x623;&#x624;&#x625;&#x626;)</C1>
  </row>
  <row>
   <C1>Hebrew(&#x5d2;&#x5d3;&#x5d4;&#x5d5;&#x5d6;)</C1>
  </row>    <
  row>
   <C1>Russian(&#x410;&#x411;&#x412;&#x413;&#x414;)</C1>
  </row>
</resultset>
```

A browser displays the document showing the actual non-ASCII characters, respectively Arabic, Hebrew, and Russian.

Example 7

Most characters are represented by code points in the range [0x20, 0xFFFF], and can be represented with a single 16-bit value. A surrogate pair is a pair of 16 bit values that represent a character in the range [0x010000..0x10FFFF]. The first half of the pair is in the range [0xD800..0xDBFF], and the second half of the pair is in the range [0xDC00..0xDFFF]. Such a pair (H, L) represents the character computed as follows (hex arithmetic):

```
(H - 0xD800) * 400 + (L – 0xDC00)
```

For example, the character “&#x01D6D1” is a lower-case bold mathematical symbol, represented by the surrogate pair D835, DED1:

```
select convert(unitext, u&'\+1d6d1')
---------------------
0xd835ded1
```

When you specify *ncr=non_ascii* or *ncr=non_server* to generate a SQLX XML document containing non-ASCII data with surrogate pair characters, the surrogate pairs appear as single NCR characters, not as pairs:

```
select convert(unitext, u&'\+1d6d1')
for xml option 'ncr=non_ascii"
-------------------------------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>&#x1d6d1;</C1>
 </row>
</resultset>
```

I18N in *xmlparse*

xmlparse supports Unicode datatypes (unichar, univarchar, unitext, and java.lang.String) for the input XML document.

Options

xmlparse parses an XML document and returns a representation of it as an image value containing the parsed document and its internal index. This representation is called a Unicode parsed image XML. *Unicode parsed image XML* is stored in columns of image.

xmlparse converts string datatypes to Unicode. Since string datatypes are in the server character set, which is always a subset of Unicode, conversion is a change in datatype that never raises a conversion exception.

Sort ordering in *xmlparse*

For details of XML sort ordering, see "Sort ordering in XML Services" on page 122.

xmlparse uses the sort ordering specified by the sp_configure option default xml sort order, and the same ordering for XML indexes. XML stores the sort ordering name in the image generated by xmlparse, called the *parsed XML sort order* of the document.

All functions that reference a parsed XML document raise an exception when the parsed XML sort order is different from the current default XML sort order.

I18N in *xmlextract*

xmlextract applies an XML query expression to an XML document, and returns the result you select. The input document can be a string datatype, a Unicode datatype, or an image datatype containing either character data or parsed XML.

The returns clause can specify a Unicode datatype as the datatype of the value extracted.

NCR option

xmlextract supports the ncr option:

```
ncr = {non_ascii|non_server|no}
```

At runtime, the ncr option is applied if:

- The result datatype is a string or Unicode datatype, not numeric or datetime or money, for instance.
- The XPath query does not specify text().

The default ncr option is:

- If the returns datatype is a Unicode datatype, the default value is ncr=no.
- If the returns datatype is a string datatype, the default value is ncr=non_server.

Sort ordering in *xmlextract*

Sort ordering in xmlextract is discussed in "Sort ordering in XML Services" on page 122.

xmlextract uses the parsed XML sort order stored in the input XML document, not the current default sort order in the server.

Sort ordering in XML Services

***sp_configure* option** XML Services defines the sp_configure option default xml sort order, which has three distinguishing characteristics:

- It is static; you must restart Adaptive Server to execute this configuration.
- The option value is the name of a Unicode sort order. For details see the table "Default Unicode sort order," in the *System Administration Guide, Volume 1*.
- The default option value is *binary*.

xmlparse xmlparse returns a parsed representation of the argument document, including an index of the document's elements and attributes and their values. The parsed representation specifies the *default xml sort order* as it exists when the document is parsed.

xmlextract xmlextract evaluates XPath queries that compare terms, such as “//book[author='John Doe']”. xmlextract compares the current *default xml sort order* with the document’s *parsed xml sort order.* If they are different, xmlextract raises an exception.

xmlextract uses the XML sort order stored in the input XML document, not the current default sort order in the server.

Note XML Services uses a single default order, the default xml sort order. It does not use *both* default Unicode xml sort order *and* default xml sort order.

Modifying the *default xml sort order* You can modify the *default xml sort order* with sp_configure.

After you modify *default xml sort order*, you can reparse previously parsed XML documents, using the Adaptive Server update command. For update see the *Reference Manual, Vol. 2, Commands*.

```
update xmldocs
set doc = xmlparse(xmlextract('/', doc))
```

I18n in *xmlvalidate*

xmlvalidate() supports Unicode datatypes, unichar, univarchar, unitext, and java.lang.String, as well as string and image datatypes. The returns clause of xmlvalidate can specify a Unicode datatype as the datatype of the extracted value.

NCR option

xmlvalidate() supports the ncr option:

```
ncr={non_ascii | non_server | no}
```

At runtime, the ncr option applies only if the datatype of the result clause is a string or Unicode datatype. For example, the option does not apply to numeric, datetime, or money datatypes.

Default NCR option

- The default NCR option value is ncr=no if the returns datatype is a Unicode datatype: unichar, univarchar, unitext, or java.lang.String.

- The default NCR option value is ncr=non_server if the returns datatype is a string datatype: char, varchar, or text.

CHAPTER 7 *xmltable()*

This chapter describes the xmltable() function in detail.

Introduction

xmltable() extracts a sequence of multi-valued elements from an XML document, and assembles a SQL table of those elements. A single call to xmltable() replaces a T-SQL loop performing multiple calls to xmlextract on each iteration. This function is invoked as a derived table (a parenthesized subquery specified in the from clause of another SQL query). Calling xmltable() is equivalent to executing a single xmlextract expression for each row of the table generated by xmltable().

xmltable() is a generalization of xmlextract. Both functions return data extracted from an XML document that is an argument in the function. The differences are:

- xmlextract returns the data identified by a single XPath query.
- xmltable() extracts the sequence, or row pattern, of the data identified by an XPath query, and extracts from each element of that sequence the data identified by a list of other XPath queries, the column patterns. It returns all the data in a SQL table.

xmltable and *derived table* syntax

These syntax sections show the basic syntax of xmltable() and where and how to use xmltable().

xmltable

Description

Extracts data from an XML document and returns it as a SQL table.

Syntax

xmltable_expression ::= xmltable
 (*row_pattern* passing *xml_argument*
 columns *column_definitions*
 options_parameter)
row_pattern ::= *character_string_literal*
xml_argument ::= xml_expression
column_definitions ::=
 column_definition [{ , *column_definition* }]
  column_definition ::=
 ordinality_column | *regular_column*
ordinality_column ::= *column_name* datatype **for ordinality**
regular_column ::=
 column_name datatype [**default** *literal*] [**null** | **not null**]
[**path** *column_pattern*]
column_pattern ::= *character_string_literal*
options_parameter ::=[,] **option** *option_string*
options_string ::= *basic_string_expression*

Derived table syntax Returns a SQL table from within a SQL from clause.

from_clause ::= from *table_reference* [, *table_reference*]...
table_reference ::= *table_view_name* | *ANSI_join* |*derived_table*
table_view_name::=See the select command in Reference Manual
  Volume 2, "Commands".
ANSI_join::=See the select command in Reference Manual
  Volume 2, "Commands".
derived_table ::=
 (subquery) as *table_name* [(*column_name* [, *column_name*]...)
 xmltable_expression as *table_name*

Examples

xmltable as derived tables This example shows a simple xmltable() call, returning a derived table.

```
select * from xmltable('/doc/item'
   passing '<doc><item><id>1</id><name>Box</name></item>'
      +'<item><id>2</lid><name>Jar</name></item></doc>'
   columns id int path 'id',name varchar(20) path 'name') as items_table
id              name
-----          -----
1              Box
2              Jar

(2 rows affected)
```

Example 1 The syntax of derived tables requires you to specify a table name (items_table), even if you do not reference it. For example, this example is incorrect.

```
select * from xmltable ('/doc/item'
    passing '<doc><item><id>1</id><name>Box</name></item>'
       +'<item><id>2</id><name>Jar</name></item></doc>'
    columns id int path 'id', name varchar(20) path 'name')
-------------
Msg 102 Level 15, State 1:
Incorrect syntax near ')'
```

Simple document reference examples In document references, the argument following passing is the input XML document. In this example the document is specified as a character string literal.

```
select * from xmltable ('/doc/item'
    passing '<doc><item><id>1</id><name>Box</name></item>'
       +'<item><id>2</id><name>Jar</name></item></doc>'
    columns id int path 'id', name varchar(20) path 'name') as items_table

id              name
-------         ------
1               Box
2               Jar

(2 rows affected)
```

Example 2 This example shows storing the document in a T-SQL variable, and referencing that variable in the xmltable() call.

```
declare @doc varchar(16384)
set @doc='<doc><item><id>1</id></name>Box</name></item>'
       +'<item><id>2</id><name>Jar</name></item></doc>'

select * from xmltable(/doc/item' passing @doc
    columns id int path 'id', name varchar(20) path 'name') as items_table

id            name
----          -----
1             Box
2             Jar

(2 rows affected)
```

Example 3 To store the document in a table and reference it with a scalar subquery:

```
select 100 as doc_id,
'<doc><item><id>1</id><name>Box</name></item><item><id>2</id>
    <name>Jar</name></item></doc>'as doc
into #sample_docs
select* from xmltable('/doc/item'
   passing(select doc from #sample_docs where doc_id=100)
   columns id int path 'id',name varchar(20) path 'name') as items_table

id            name
-------       -----
1             Box
2             Jar

(2 rows affected)
```

Row patterns The first argument in the xmltable call, the *row-pattern* (`’/doc/item’`) is an XPath query expression whose result is a sequence of elements from the specified document. The xmltable call returns a table with one row for each element in the sequence.

Example 4 If the row pattern returns an empty sequence, the result is an empty table:

```
select * from xmltable ('//item_entry'
   passing '<doc><item><id>1</id><name>Box</name></item>'
      +'<item><id>2</id><name>Jar</name></item></doc>'
   columns id int path 'id',
      name varchar(20) path 'name') as items_table

id               name
-------         ------


(0 rows affected)
```

Example 5 The row pattern expression cannot be an XPath function:

```
select * from xmltable ('/doc/item/tolower()'
   passing '<doc><item><id>1</id><name>Box</name></item>'
      +'<item><id>2</id><name>Jar</name></item></doc>'
   columns id int path 'id',
      name varchar(20) path 'name') as items_table

id    name
---  ------
Msg 14825, Level 16, State 0:
Line1:
XPath function call must be at leaf level.
```

Column patterns The arguments following the columns keyword is the list of column definitions. Each column definition specifies a column name and datatype, as in create table, and a path, called the column pattern. The *column-pattern* is an XPath query expression that applies to an element of the sequence returned by the *row-pattern*, to extract the data for a column of the result table.

Example 6 When the data for a column is contained in an XML attribute, specify the column pattern using "@" to reference an attribute. For example:

```
select * from xmltable ('/doc/item'
   passing '<doc><item id>"1"><name>Box</name></item>'
      +'<item id="2">/id><name><Jar</name></item></doc>'
   columns id int path '@id', name varchar(20)) as items_table

id              name
--------------------
1               Box
2               Jar

(2 rows affected)
```

Default column patterns A *column-pattern* is commonly the same as the specified *column_name*, for example name. In this case, omitting the column-pattern results in defaulting to the *column_name*:

```
select * from xmltable ('/doc/item'
   passing '<doc><item><id>1</id><name>Box</name></item>'
      +'<item><id>2</id><name>Jar</name></item></doc>'
   columns id int, name varchar(20)) as items_table

id              name
--------------------
1               Box
2               Jar

(2 rows affected)
```

Example 7 If you want a column pattern to default to the column name, in a column whose value is in an XML attribute, use a quoted identifier. You must then quote such identifiers when you reference them in the results:

```
set quoted_identifier on
select "@id", name from xmltable ('/doc/item'
   passing '<doc><item id="1"><name>Box</name></item>'
      +'<item id="2"><name>Jar</name></item></doc>'
   columns "@id" int, name varchar(20)) as items_table

@id              name
```

```
--------------------
1               Box
2               Jar

(2 rows affected)
```

Example 8 You can also use quoted identifiers to specify column names as default column patterns, using column names that are more complex XPath expressions. For example:

```
set quoted_identifier on
select "@id", "name/short", "name/full" from xmltable ('/doc/item'
    passing '<doc><item id="1"><name><short>Box</short>
           <full>Box, packing, moisture resistant, plain</full>
           </name></item>'
    +'<item id="2"><name><short>Jar</short>
           <full>Jar, lidded, heavy duty</full>
           </name></item></doc>'
    columns "@id" int, "name/short" varchar(20), "name/full" varchar(50))
    as items_table

@id              name/short              name/full
--------------------
1               Box                     Box, packing, moisture resistant,
plain
2               Jar                      Jar, lidded, heavy duty

(2 rows affected)
```

Implicit text() This example demonstrates the function text(), which is generally implicit in the column pattern. text() removes XML element tags. For example, this XPath query returns the selected element with the XML markup:

```
1> declare @doc varchar(16384)
2> set @doc= ’<doc><item><id>1</id></name>Box</name></item>’
             +<’item><id>2</id><name>Jar</name></item></doc>’
3> select xmlextract(’/doc/item[2]/name’, @doc)
------------
<name>Jar</name>
```

Example 9

Adding text() to the XPath query removes the XML tags:

```
1> declare @doc varchar(16384)
2> set @doc= ’<doc><item><id>1</id></name>Box</name></item>’
             +<’item><id>2</id><name>Jar</name></item></doc>’
```

```
3> select xmlextract(’/doc/item[2]/name/text()’, @doc)
------------
Jar
```

Example 10 text() is implicit in most column patterns. This example does not specify text() in the column pattern for either the id or name column:

```
select * from xmltable ('/doc/item'
   passing '<doc><item><id>1</id><name>Box</name></item>'
      +'<item><id>2</id><name>Jar</name></item></doc>'
   columns id int path 'id', name varchar(20) path 'name') as items_table

id             name
---            -------
1              Box
2              Jar

(2 rows affected
```

)

Datatype conversion You can derive column values in datatype conversions by applying an implicit SQL convert statement to the data extracted from the column pattern. For example:

```
select * from xmltable ('/emps/emp'
   passing '<emps>
<emp><id>1</id><salary>123.45</salary><hired>1/2/2003</hired></emp'
+'<emp><id>2</id><salary>234.56</salary><hired>2/3/2004</hired></emp>'
      +</emps>'
   columns id int path 'id', salary dec(5,2), hired date)
as items_table

id         salary        hired
-------    ---------     --------
1          123.45        Jan 2, 2003
2          234.56        Feb 3, 2004
(2 rows affected)
```

Example 11 The extracted XML data for the column must be convertible to the column datatype, or an exception is raised:

```
select * from xmltable ('/emps/emp'
   passing '<emps>
+<emp><id>1</id><salary>123.45</salary><hired>1/2/2003</hired></emp'
+'<emp><id>2</id><salary>234.56 C$</salary><hired>2/3/2004</hired></emp>'
+</emps>'
columns id int path 'id', salary dec(5,2), hired date)
as items_table
----------------------
Msg 14841, Level 16, State 3:
Line 1:
XMLTABLE:Failed to convert column pattern result to DECML for column 1.
```

Example 12 To handle XML data whose format is not suitable for a SQL convert function, extract the data to a string column (varchar, text, image, java.lang.String)

```
select * from xmltable ('/emps/emp'
   passing '<emps>
+<emp><id>1</id><salary>123.45</salary><hired>1/2/2003</hired></emp'
+'<emp><id>2</id><salary>234.56 </salary><hired>2/3/2004</hired></emp>'
+</emps>'
columns id int, salary varchar(20), hired date)
as items_table

id             salary         hired
---------------------------------
1              123.45        Jan 2, 2003
2              234.56       Feb 3, 2004
(2 rows affected)
```

Ordinality columns The order of elements in XML documents can be significant.

Elements are sometimes ordered by the value of contained elements. In this example, the <*item*> elements are ordered by the value of the contained <*id*> elements.

```
<doc>
   <item><id>1<name>Box</name></item>'
   <item><id>2<name>Jar</name></item>'
</doc>
```

You can also order elements in an arbitrary but significant manner. In the following example, the order of the <*item*> elements is based on no values, but may reflect a priority ordering: first in, first out. Such an ordering can be significant for the application of the data.

```
<doc>
   <item><id>25<name>Box</name></item>'
   <item><id>15<name>Jar</name></item>'
</doc>
```

Example 13 You can use an *ordinality_column* in xmltable to record the ordering of elements in the input XML document:

```
declare @doc varchar(16384)
set @doc = '<doc><item><id>25<name>Box</name></item>'
    +'<item><id>15</id><name>Jar</name></item></doc'
select * from xmltable('/doc/item' passing @doc
    columns item_order int for ordinality,
       id int path 'id',
       name varchar(20) path 'name') as items_table
order by item_order

item_order         id            name
----------         ---           ----
1                  25            Box
2                  15            Jar
(2 rows affected)
-----------
```

Without the for ordinality clause and the item_order column, there is nothing in the id and name columns that indicates that the row of id 25 precedes the row of id 15. The for ordinality clause provides a way to make sure that the ordering of the output SQL rows is the same as the ordering of the elements in the input XML document.

The datatype of an ordinality column can be any fixed numeric datatype: int, tinyint, bigint, numeric, or decimal. numeric and decimal must have a scale of 0. An ordinality column cannot be real or float.

Null values If a column pattern returns an empty result, the action taken depends on the `default` and `{null | not null}` clauses.

Example 14 This example omits the <*name*> element from the second <*item*>. The name column allows names by default.

```
select * from xmltable ('//item'
    passing '<doc><item><id>1</id><name>Box</name></item>'
    +'<item><id>2</id></item></doc>'
columns id int path 'id', name varchar(20), path 'name')
as items_table
-------------------------
id            name
------------------
1             Box
2             NULL

(2 rows affected)
```

Example 15 This example omits the <*name*> element from the second <*item*>, and specifies not null for the name column:

```
select * from xmltable ('/doc/item'
   passing '<doc><item><id>1</id><name>Box</name></item>'
   +'<item><id>2</id></item></doc>'
columns id int path 'id', name varchar(20) not null path 'name')
as items_table
-------------------------
Msg 14847, Level 16, State 1:
Line 1:
XMLTABLE column 0, does not allow null values.
```

Example 16 This example adds a default clause to the name column, and omits the <*name*> elements from the second <*item*>.

```
select * from xmltable ('/doc/item'
   passing '<doc><item><id>1</id><name>Box</name></item>'
   +'<item><id>2</id></item></doc>'
columns id int path 'id' name varchar(20) default '***' path 'name')
as items_table

id             name
------------------
1              Box
2              ***
(2 rows affected)
```

Context of *xmltable* calls These examples show SQL commands in which you can use an xmltable call in a derived table expression.

Example 17 select – You can use xmltable() in a simple select statement:

```
select * from xmltable ('/doc/item'
   passing '<doc><item><id>1</id><name>Box</name></item>'
   +'<item><id>2</id><name>Jar</name></item></doc>'
columns id int path 'id'
   name varchar(20) path 'name')as items_table

id             name
--             ----
1              Box
2              Jar
(2 rows affected)
```

Example 18 View definition – Specify select using xmltable in a view definition.This example stores a document in a table and references that stored document in a create view statement, using xmltable to extract data from the table:

```
select 100 as doc_id,
'<doc><item><id>1</id><name>Box</name></item>'
```

```
    +'<item><id>2</id><name>Jar</name></item></doc>' as doc
into sample_docs
create view items_table as
    select * from xmltable('/doc/item'
        passing (select doc from sample_docs where doc_id=100)
        columns id int path 'id'
        name varchar(20) path 'name')as xml_extract

id            name
------------------
1             Box
2             Jar
(2 rows affected)
```

Example 19 Cursor declaration – Specify select using xmltable to declare a cursor:

```
declare cursor C for
select * from xmltable ('/doc/item'
    passing (select doc from sample_docs where id=100)
    columns id int path 'id'
    name varchar(20) path 'name')as items_table
declare @idvar int
declare @namevar varchar(20)
open C
while @@sqlstatus=0
begin
fetch C into @idvar, @namevar
print 'ID "%1!" NAME"%2!"', @idvar, @namevar
end
-------------------------
ID "1" NAME "Box"
ID "2" NAME "Jar"

(2 rows affected)
```

In applications that require multiple actions for each generated row, such as executing update, insert, or delete from other tables, based on the data in each generated row, process an xmltable result with a cursor loop. Alternatively, you can store the xmltable result in a temporary table and process that table with a cursor loop.

Example 20 select into – Specify select using xmltable in select into:

```
select * into #extracted_table
from xmltable('/doc/item'
    passing (select doc from sample_docs where doc_id=100
    columns id int path 'id'
```

```
      name varchar(20) path 'name') as items_table

select * from #extracted_table

id            name
----          ------------
1             Box
2             Jar
```

Example 21 insert – Specify select using xmltable in an insert command:

```
create table #extracted_data (idcol int, namecol varchar(20))
insert into #extracted_data
select * into #extracted_table from xmltable('/doc/item'
   passing (select doc from sample_docs where doc_id=100
   columns id int path 'id', name varchar(20) path 'name')as items_table
select * from extracted_data

id             name
-----          ---------
1              Box
2              Jar
(2 rows affected)
```

Example 22 Scalar subquery – Specify select using xmltable in a scalar subquery. xmltable returns a SQL table, so the scalar subquery must perform either an aggregation or a selection to return a single row and column for the scalar subquery result.

```
declare @idvar int
set @idvar = 2
select @idvar,
(select name from xmltable ('/doc/item'
   passing(select doc from sample_docs where doc_id=100
   columns id int path 'id',name varchar(20) path 'name') as item_table
where items_table.id=@idvar)
----------------------------
2                      Jar
(1 rows affected)
```

Example 23 Joins – Join an xmltable result with other tables, using either comma-list joins or outer joins:

```
create table prices (id int, price decimal (5,2))
insert into prices values(1,123.45)
insert into prices values (2,234.56)
select prices.id,extracted_table.name, prices.price
from prices,(select * from xmltable('/doc/item'
    passing (select doc from sample_docs where doc_id=100
    columns id int path 'id', name varchar(20) path 'name')as a) as
              extracted_table
where prices.id=extracted_table.id

id       name        price
-----          ---------
1              Box   123.45
2              Jar   234.56
(2 rows affected)
```

Processing tables of documents You can apply xmltable() to the XML document in each row of a table of XML documents. For example, the next example creates a table containing two columns:

- The pub_id of one of the three publishers in the pubs2_publishers table.
- An XML document containing the title and price of each document published by that publisher. To reduce the size of the example table only titles whose price is greater than $15.00 are included:

```
create table high_priced_titles
(pub_id char(4), titles varchar (1000))
insert into high_priced_titles
select p.pub_id,
    (select title_id, price from pubs2..titles t,pubs2..publishers p
       where price> 15 and t.pub_id=p.pub_id
       for xml
       option 'tablename=expensive_titles, rowname=title')
          returns varchar(1000))as titles
from pubs2..publishers p
select * from high_priced_titles
--------------
pub_id    titles
------    -------------------------------------------------
0736             <expensive_titles>
    <title> <title_id>PS3333</title_id> <price>19.99</price></title>
                </expensive_titles>

0877             <expensive_titles>
    <title> <title_id>MC2222</title_id> <price>19.99</price></title>
    <title> <title_id>PS1372</title_id> <price>21.59</price></title>
    <title> <title_id>TC3218</title_id> <price>20.95</price></title>
                </expensive_titles>

01389             <expensive_titles>
    <title> <title_id>BU1032</title_id> <price>19.99</price></title>
    <title> <title_id>BU7832</title_id> <price>19.99</price></title>
    <title> <title_id>PC1035</title_id> <price>22.95</price></title>
    <title> <title_id>PC8888</title_id> <price>20.00</price></title>
                 </expensive_titles>
(3 rows affected)
```

Example 24 Use xmltable in a scalar subquery to process the XML document in each row, as a SQL table. For example, list the maximum title price for each publisher:

```
select pub_id
(select max(price)
 from xmltable('//title'passing hpt.titles
       columns title_id char(4), price money)
       as extracted_titles, high_priced_titles hpt) as max_price
 from high_priced_titles hpt
--------------
pubid         max_price
-----------------------
0736          19.99
0877          21.59
1389          22.95
```

This high_priced_titles table is essentially hierarchic: each row is an intermediate node, which contains, in its title column, a leaf node for each *title* element in the XML document. high_priced_titles has three rows.

You can flatten that hierarchy, producing a table with a row for each *title* element. To flatten the data in the titles column and produce a table, high_priced_titles_flattened, which has eight rows (one for each of the *titles*/*title* elements), use one of the following solutions.

Solution 1 You can produce high_priced_titles_flattened by using a loop that processes high_priced_titles, and applies xmltable to the titles document in each row. In the example below, notice the from clause:

```
from(select @pub_id_var)as ppp,
   xmltable('//title' passing @titles_var
      columns title_id char(6),price money)as ttt
```

The variables *@pub_id_var* and *@titles_var* are the pub_id and titles columns from the current row of high_priced_titles. The from clause joins two derived tables:

- `(select @pub_id_var) as ppp`

  This is a table with one row and one column, containing the pub_id.

- `xmltable(...) as ttt`

  This generates a table with a row for each *title* element in the *titles* document of the current high_priced_titles row.

To flatten the hierarchy, join these two derived tables, which appends the pub_id column to each row generated from the titles column:

```
create table high_priced_titles_flattened_1
(pub_id char(4), title_id(char(6), price money)

declare C cursor for select * from high_priced_titles
declare @pub_id_var char(4)
declare @titles_var char(1000)
open C

while @@sqlstatus =0
begin
fetch C into @pub_id_var, @titles_var

insert into high_priced_titles_flattened_1
select *
from (select @pub_id_var) as ppp,(col1),
    xmltable('//title' passing @titles_var
    columns title_id char (6), price money) as ttt
end
select * from high_priced_titles_flattened_1

pub_id    title_id          price
------    -------------------------------------------------
0736             PS3333               19.99
0877             MC2222               19.99
0877             PS1372               21.59
0877             TC3218               20.95
1389             BU1032               20.95
1389             BU7832               19.99
1389             PC1035               19.99
1389             PC8888               20.00
```

Solution 2 You can also generate the high_priced_titles table using a special join.

This example joins two tables: high_priced_titles as hpt, and the table generated by xmltable. The passing argument of xmltable references the preceding hpt table. Normally, it is illegal to reference a table in a from clause, in a derived table expression within the same from clause. However, xmltable is allowed to reference other tables in the same from clause, as long as these tables *precede* the xmltable call in the same from clause.

```
select hpt.pub_id, extracted_titles.*
into high_priced_titles_flattened_3
from high_priced_titles as hpt,
      xmltable('//title'
      passing htp.titles,
      columns
         title_id char(6)
         price money)as extracted_titles
pub_id           title_id             price
---------        --------------       ---------
0736             PS3333               19.99
0877             MC2222               19.99
0877             PS1372               21.59
0877             TC3218               20.95
1389             BU1032               20.95
1389             BU7832               19.99
1389             PC1035               19.99
1389             PC8888               20.00
```

Usage

- xmltable is a built-in, table-valued function.
- The result type of an xmltable expression is a SQL table, whose column names and their datatypes are specified by *column_definitions*.
- These keywords are associated with xmltable:
  - Reserved: for, option
  - Not reserved: columns, ordinality, passing, path, xmltable
- The expressions in the arguments of an xmltable call can reference the column names of preceding tables in the from clause containing the xmltable call. Only tables that precede the xmltable call can be referenced. Such a reference, to a column of a preceding table in the same from clause, is called a *lateral reference*. For example:

```
select * from T1, xmltable(...passing T1.C1...)
       as XT2, xmltable(...passing XT2.C2...)as XT3
```

  The reference to T1.C1 in the first xmltable call is a lateral reference to column C1 of table T1. The reference to XT2.C2 in the second xmltable call is a lateral reference to column C2 of the table generated by the first xmltable call.

- You cannot use xmltable in the from clause of an update or delete statement. For example, the following statement fails:

```
update T set T.C=...
```

```
from T,xmltable(...)
where...
```

- You cannot update the SQL table returned by an xmltable expression.
- Datatypes in *regular_columns* can be any SQL datatype.
- The literal following a default in a *regular_column* must be assignable to the datatype of the column.
- There can be no more than one *ordinality_column*; the datatype specified for this variable must be integer, smallint, tiny int, decimal, or numeric. decimal and numeric must have a scale of zero.
- An *ordinality_column*, if one exists, is not nullable. The nullable property of other columns is specified by the `{null | not null}` clause. The default is null.

Note This default is different from the default value of create table.

- The current setting of set quoted_identifier applies to the clauses of an xmltable expression. For example,
  - If set quoted_identifier is on, column names can be quoted identifiers, and string literals in *row_pattern*, *column_pattern,* and **default** literals must be surrounded with single quotation marks.
  - If set quoted_identifier is off, column names cannot be quoted identifiers, and string literals in *row_pattern*, *column_pattern*, and **default** literals can be surrounded with either single or double quotation marks.
- The general format of the option_string is described in the section "option_strings: general format,".

xmltable row and column patterns xmltable row and column patterns are allowed to be only simple paths. Simple paths in XPath consist only of forward traversals using '/' and element/attribute names.

- If the *row_pattern* does not begin at the root level of the document specified by *xml_argument*, an exception is raised. *The row pattern must begin at the root of the XML document.*
- If the *row_pattern* specifies an XML function, an exception is raised. *The row pattern must not specify an XML function.*

- If a column_definition does not specify a path, then the default column_pattern is the column_name of the column definition. This default is subject to the case sensitivity of the server. For example, consider this statement:

  ```
  select * from xmltable(...columns name
  varchar(30),...)
  ```

  If the server is case-insensitive, this is equivalent to the following:

  ```
  select * from xmltable(...columns name varchar(30)
  path 'name',...)
  ```

  If the server is case sensitive, the first statement is equivalent to:

  ```
  select * from xmltable
  (...columns name varchar(30)path 'NAME',...)
  ```

Generating the rows of the result table

The result value of an xmltable expression is a T-SQL table RT, defined as follows:

- RT has a row for each element in the XML sequence that results from applying the *row_pattern* to the *xml_argument*.
- The rows of RT have a column for each *column_definition*, with the *column_name* and datatype specified in the *column_definition*.
- If a *column_definition* is a *ordinality_column*, its value for the Nth row is the integer N.
- If a *column_definition* is a *regular_column*, its value for the Nth row corresponds to the following:
  - Let XVAL be the result of applying this XPath expression to the *xml_argument*:

    ```
    (row_pattern[N])/column_pattern/text()
    ```

  - If XVAL is empty, and the *column_definition* contains a default clause, the value of the column is that default value.

    If XVAL is empty and the *column_definition* specifies not null, an exception is raised.

    Otherwise, the value of the column is the null value.
  - If XVAL is not empty, and the datatype of the column is char, varchar, text, unitext, unichar, univarchar, or java.lang.String, de-entitize XVAL.
  - The value of the column is the result of:

```
convert(datatype,XVAL)
```

See also

For a sample application using xmltable, see Appendix F, “Sample Application for xmltable(),”in *XML Services*.

APPENDIX A

The *sample_docs* Example Table

The descriptions of the XML query functions reference an example table named sample_docs. This chapter shows you how to create and populate that table.

The sample_docs table has three columns and three rows.

sample_docs table columns and rows

This section shows the structure of the sample_docs table.

Sample_docs table columns

The *sample_docs* table has three columns:

- name_doc
- text_doc
- image_doc

In a specified example document, name_doc specifies an identifying name, text_doc specifies the document in a text representation, and image_doc specifies the document in a parsed XML presentation stored in an image column. The following script creates the table:

```
create table sample_docs
(name_doc varchar(100),
text_doc text null,
image_doc image null)
```

sample_docs table rows

The sample_docs table has three rows:

- An example document, “bookstore.xml”.
- An XML representation of the publishers table of the pubs2 database.
- An XML representation of (selected columns of) the titles table of the pubs2 database.

The following script inserts the example “bookstore.xml” document into a row of the sample_docs table:

```
insert into sample_docs
    (name_doc, text_doc)
    values ( "bookstore",

"<?xml version='1.0' standalone = 'no'?>
<?PI_example Process Instruction ?>
<!--example comment-->
<bookstore specialty='novel'>
<book style='autobiography'>
    <title>Seven Years in Trenton</title>
       <author>
          <first-name>Joe</first-name>
          <last-name>Bob</last-name>
           <award>Trenton Literary Review
             Honorable Mention</award>
      </author>
      <price>12</price>
</book>
<book style='textbook'>
   <title>History of Trenton</title>
      <author>
          <first-name>Mary</first-name>
           <last-name>Bob</last-name>
           <publication>Selected Short Stories of
           <first-name>Mary</first-name>
            <last-name>Bob</last-name>
           </publication>
      </author>
       <price>55</price>
</book>
<?PI_sample Process Instruction ?>
<!--sample comment-->
<magazine style='glossy' frequency='monthly'>
  <title>Tracking Trenton</title>
```

```
     <price>2.50</price>
   <subscription price='24' per='year'/>
</magazine>
<book style='novel' id='myfave'>
     <title>Trenton Today, Trenton Tomorrow</title>
     <author>
         <first-name>Toni</first-name>
         <last-name>Bob</last-name>
         <degree from='Trenton U'>B.A.</degree>
          <degree from='Harvard'>Ph.D.</degree>
          <award>Pulizer</award>
          <publication>Still in Trenton</publication>
           <publication>Trenton Forever</publication>
     </author>
     <price intl='canada' exchange='0.7'>6.50</price>
     <excerpt>
     <p>It was a dark and stormy night.</p>
     <p>But then all nights in Trenton seem dark and
         stormy to someone who has gone through what
         <emph>I</emph> have.</p>
      <definition-list>
         <term>Trenton</term>
         <definition>misery</definition>
      </definition-list>
  </excerpt>
</book>

<book style='leather' price='29.50'
xmlns:my='http://www.placeholdernamehere.com/schema/'>
  <title>Who's Who in Trenton</title>
  <author>Robert Bob</author>
</book>

</bookstore>")
```

sample_docs tables

The other two rows of the *sample_docs* table are XML representations of the publishers and titles tables of the pubs2 database. The pubs2 database is an database of example tables that is described in the *Transact-SQL User's Guide*.

The publishers and titles tables are two of the tables in this sample database.To shorten the example, the XML representation of the titles table includes only selected columns.

The following script generates the XML representations of the publishers and titles tables with the forxmlj function, which is described in “forxmlj, forxmldtdj, forxmlschemaj, forxmlallj” on page 71.

Table script (for *publishers* table)

These two insert statements add a row for the publishers table and a row for the authors table to the the sample_docs table. Each row contains a column that identifies the row (‘publishers’, ‘authors’), and a text_doc column

that provides an XML representation of the corresponding pubs2 table. You can generate the XML document by calling the Java forxmlj function.

```
insert into sample_docs (name_doc, text_doc)
    values ('publishers',
    forxmlj('select * from pubs2..publishers',
            'tablename=publishers'))

insert into sample_docs (name_doc, text_doc)
    values ('authors',
    forxmlj('select title_id, title
             type, pub_id, price,
             advance, total_sales
             from pubs2..authors',
            'tablename=authors')
```

Note This script uses the forxmlj function, which is a Java-based function that you must install before you can use. See Appendix C, “Setting up XML Services,” for instructions on installing this function.

Publishers table representation

This code sample shows the XML representation of the *publishers* table in the Pubs 2 database, generated by the script in “sample_docs tables” on page 147.

```
set stringsize 16384
```

```
select text_doc from sample_docs
where name_doc='publishers'

text_doc
-------------------------------------------
<publishers
  xmlns:xsi="http://www.w3.org/2001/XMLSchema
  instance">

<row>
   <pub_id>0736</pub_id>
   <pub_name>New Age Books</pub_name>
   <city>Boston</city
   <state>MA</state>
</row>

<row>
   <pub_id>0877</pub_id>
   <pub_name>Binnet & Hardley</pub_name>
   <city>Washington</city>
   <state>DC</state>
</row>

<row>
   <pub_id>1389</pub_id>
   <pub_name>Algodata Infosystems</pub_name>
   <city>Berkeley</city>
   <state>CA</state>
</row>

</publishers>
(1 row affected)
```

Titles table representation

This section shows the XML representation of selected columns of the titles table.

```
set stringsize 16384
select text_doc from sample_docs
where name_doc='titles'

 text_doc
```

```
----------------------------------------------------
<titles
xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”>

  <row>
     <title_id>BU1032<title_id>
     <title>The Busy Executive’s Data Base
           Guide</title>
     <type>business</type>
     <pub_id>1389</pub_id>
     <price>19.99</price>
     <advance>5000.00</advance>
     <total_sales>4095</total_sales>
  </row>

  <row>
     <title_id>BU1111</title_id>
     <title>Cooking with Computers:
               Surreptitious Balance Sheets</title>
     <type>business </type>
     <pub_id>1389</pib_id>
     <price>11.95</price>
     <advance>5000.00</advance>
     <total_sales>3876</total_sales>
  </row>

  <row>
     <title_id>BU2075</title_id>
     <title>You Can Combat Computer Stress!</title>
     <type>business </type>
     <pub_id>0736</pub_id>
     <price>2.99</price>
     <advance>10125.00</advance>
     <total_sales>18722</total_sales>
  </row>

  <row>
     <title_id>BU7832</title_id>
     <title>Straight Talk About Computers</title>
     <type>business </type>
     <pub_id>1389</pub_id>
     <price>19.99</price>
     <advance>5000.00</advance>
     <total_sales>4095</total_sales>
  </row>
```

```
<row>
   <title_id>MC2222</title_id>
   <title>Silicon Valley Gastronomic Treats</title>
   <type>mod_cook</type>
   <pub_id>0877</pub_id>
   <price>19.99</price>
   <advance>0</advance>
   <total_sales>2032</total_sales>
</row>

<row>
   <title_id>MC3021</title_id>
   <title>The Gourmet Microwave</title>
   <type>mod_cook</type>
   <pub_id>0877</pub_id>
   <price>2.99</price>
   <advance>15000.00</advance>
   <total_sales>22246</total_sales>
</row>

<row>
   <title_id>MC3026</title_id>
   <title>The Psychology of Computer Cooking</title>
   <type>UNDECIDED</type>
   <pub_id>0877</pub_id>
</row>

<row>
   <title_id>PC1035</title_id>
   <title>But Is IT User Friendly?</title>
   <type>popular_comp</type>
   <pub_id>1389</pub_id>
   <price>22.99</price>
   <advance>7000.00</advance>
   <total_sales>8780</total_sales>
</row>

<row>
   <title_id>PC8888</title_id>
   <title>Secrets of Silicon Valley</title>
   <type>popular_comp</type>
   <pub_id>1389</pub_id>
   <price>20.00</price>
   <advance>8000.00</advance>
   <total_sales>4095</total_sales>
</row>
```

```
<row>
   <title_id>PC9999</title_id>
   <title>Net Etiquette</title>
   <type>popular_comp</type>
   <pub_id>1389</pub_id>
</row>

<row>
   <title_id>PS1372</title_id>
   <title>Computer Phobic and Non-Phobic
       Individuals: Behavior Variations</title>
   <type>psychology </type>
   <pub_id>0877</pub_id>
   <price>21.59</price>
   <advance>7000.00</advance>
   <total_sales>375</total_sales>
</row>

<row>
   <title_id>PS2091</title_id>
   <title>Is Anger the Enemy?</title>
   <type>psychology </type>
   <pub_id>0736</pub_id>
   <price>10.95</price>
   <advance>2275.00</advance>
   <total_sales>2045</total_sales>
</row>

<row>
   <title_id>PS2106</title_id>
   <title>Life Without Fear</title>
   <type>psychology </type>
   <pub_id>0736</pub_id>
   <price>7.99</price>
   <advance>6000.00</advance>
   <total_sales>111</total_sales>
</row>

<row>
   <title_id>PS3333</title_id>
   <title>Prolonged Data Deprivation:
         Four Case Studies</title>
   <type>psychology</type>
   <pub_id>0736</pub_id>
   <price>19.99</price>
   <advance>2000.00</advance>
   <total_sales>4072</total_sales>
</row>

<row>
   <title_id>PS7777</title_id>
   <title>Emotional Security:
       A New Algorithm</title>
   <type>psychology </type>
   <pub_id>0736</pub_id>
   <price>7.99</price>
   <advance>4000.00</advance>
   <total_sales>3336</total_sales>
</row>

<row>
   <title_id>TC3218</title_id>
   <title>Onions, Leeks, and Garlic:
      Cooking Secrets of the Mediterranean</title>
   <type>trad_cook </type>
   <pub_id>0877</pub_id>
   <price>20.95</price>
   <advance>7000.00</advance>
   <total_sales>375</total_sales>
</row>

<row>
   <title_id>TC4203</title_id>
   <title>Fifty Years in Buckingham
        Palace Kitchens</title>
   <type>trad_cook </type>
   <pub_id>0877</pub_id>
   <price>11.95</price>
   <advance>4000.00</advance>
   <total_sales>15096</total_sales>
</row>

<row>
   <title_id>TC7777</title_id>
   <title>Sushi, Anyone?</title>
   <type>trad_cook </type>
   <pub_id>0877</pub_id>
   <price>14.99</price>
   <advance>8000.00</advance>
   <total_sales>4095</total_sales>
</row>
```

```
</titles>

(1 row affected)
```

APPENDIX B

XML Services and External File System Access

The Adaptive Server External File System Access feature provides access to operating system files as SQL tables. This appendix describes the use of the native XML processor with the File System Access Feature. For more detailed information, see the *Adaptive Server Component Integration Services User's Guide*.

When you use the File System Access feature, you create a proxy table that maps an entire directory tree from the external file system, using Adaptive Server's Component Integration Services (CIS) feature. Then you use the built-in functions of the native XML processor on the data in the proxy table to query XML documents stored in the external file system.

With External Directory Recursive Access, you can map a proxy table to a parent directory, and to all its subordinate files and subdirectories.

Getting Started

This section explains how to set up XML Services with External File System Access capabilities.

Enabling XML services and External File System Access

- Enable XML Services,CIS, and file access, using sp_configure:

```
sp_configure "emable xml", 1
```

- Verify that the configuration parameter enable cis is set to 1:

```
sp_configure "enable cis",1
```

- Enable file access using sp_configure:

```
sp_configure "enable file access", 1
```

Character set conversions with external file systems

In general, the content columns of external files system tables are treated as image. However, special conversions are performed when a *content* column is assigned to a Unicode column, i.e. a column of datatype unichar,univarchar, unitext, or java.lang.String. Such assignment of a *content* column to a Unicode column occurs in the following contexts:

- An insert command used to insert a Unicode column from a subquery that references a *content* column.
- An update command used to update a Unicode column with a new value that references a *content* column.
- A convert function call that specifies both a target Unicode datatype and a source value that is a *content* column.

In assigning a content column to Unicode, use these rules:

- If the source document has a BOM (Byte Order Mark), to convert the source document the BOM must indicate UTF-8 ir UTF-16. If the BOM indicates UCS-4, an error is raised. UCS-4 is not supported.
- If the source document has an XML header that includes an encoding clause, but no BOM, the encoding clause must specify the server character set or UTF-8 to convert the source data. An encoding clause that specifies a character set other than the server set or UTF-8 raises an error.
- If the source document has no XML header, a header with no encoding clause, and no BOM, the processor treats the character set as UTF-8, and converts the source data.
- If an error occurs during a conversion, an error is raised, but the statement continues.

Examples

The following examples show how you can use various XML built-ins to query XML documents in the external file system.

Setting up your XML documents and creating the proxy table

These examples use two XML documents stored in the files named *bookstore.1.xml* and *bookstore.2.xml*, that you create:

```
cat bookstore.1.xml

<?xml version='1.0' standalone = 'no'?>
<!-- bookstore.1.xml example document--!>
<bookstore specialty='novel'>
<book style='autobiography'>
   <title>Seven Years in Trenton</title>
    <author>
        <first-name>Joe</first-name>
         <last-name>Bob</last-name>
         <award>Trenton Literary Review Honorable Mention</award>
        </author>
        <price>12</price>
        </book>
 </bookstore>

cat bookstore.2.xml

<?xml version='1.0' standalone = 'no'?>
<!-- bookstore.2.xml example document--!>
<bookstore specialty='novel'>
   <book style='compbook'>
      <title>Modern Database Management</title>
       <author>
          <first-name>Jeffrey</first-name>
          <last-name>Hoffer</last-name>
       </author>
       <price>112.00</price>
   </book>
</bookstore>
```

You can reference these XML documents with File System Access, using create proxy table.

The following code sample shows the use of create proxy table. The directory pathname in the at clause must reference a file system directory that Adaptive Server can both see and search. If you add an ’;R’ (indicating "Recursion") extension to the end of the pathname CIS extracts file information from every directory subordinate to the pathname.

```
create proxy_table xmlxfsTab external directory
at "/remote/nets3/bharat/xmldocs;R"
select filename from xmlxfsTab f
```

```
filename
-------------------------------------------
bookstore.1.xml
bookstore.2.xml

(2 rows affected)
```

The significant columns are filename and content. The other columns contain data for access permission and so forth. The filename column holds the file name (in this example the XML document file name) and the content column holds the actual data for that file. The datatype of the content column is image.

Example: extracting the book title from the XML documents

```
select filename, xmlextract("//book/title" , content)
from xmlxfsTab
```

```
filename
-------------------------------------------------
bookstore.1.xml
<title>Seven Years in Trenton</title>
bookstore.2.xml
<title>Modern Database Management</title>

(2 rows affected)
```

Example: importing XML documents or XML query results to an Adaptive Server table

You can transfer complete XML documents or XML query results between an File Access directory structure and either a database table or another File Access directory structure. To reference a complete XML document, use the xmlextract function with the root XPath operator ("/").

```
insert into xmldoctab select filename,xmlcol=xmlextract("/",content) into
from xmlxfsTab
------------------
(2 rows affected)
```

In this example, the datatype of the xmlxfsTab.content column is image, and the default datatype returned by the xmlextract built-in function is text. Therefore, specify the returns image clause in the xmlextract call to return the result as an image value.

To preserve a header, use xmlvalidate() instead of xmlextract():

```
insert into xmldoctab select filename,xmlvalidate(content)
from xmlxfsTab
-------------
(2 rows affected)
```

The following will create a new subdirectory, *XmlDir*:

```
insert into xmlxfsTab(filename,content)
select filename = 'XmlDir/'+filename,
    xmlextract("/",xmlcol returns image) from xmldoctab
-----------
(2 rows affected)
```

This code sample queries those XML documents from the new *XmlDir* subdirectory:

```
select filename, xmlextract("//book/title", content)
from xmlxfsTab
where filename like '%XmlDir%' and filetype = 'REG'

filename
----------------------------------
XmlDir/bookstore.1.xml
<title>Seven Years in Trenton</title>
XmlDir/bookstore.2.xml
<title>Modern Database Management</title>

(2 rows affected)
```

Example: storing parsed XML documents in the file system

You can parse the XML documents stored in the external file system and store the parsed result either in an Adaptive Server table or in the File Access system.

```
insert xmlxfsTab(filename, content)
select 'parsed'+t.filename,xmlparse(t.content) from xmlxfsTab
-----------
(2 rows affected)
```

The following code sample queries the parsed documents stored in the XFS file system.

```
select filename, xmlextract("//book/title", content)
from xmlxfsTab
where filename like 'parsed%'and filetype = 'REG'
filename
```

```
--------------------------------------------
parsedbookstore.1.xml
<title>Seven Years in Trenton</title>
parsedbookstore.2.xml
<title>Modern Database Management</title>

(2 rows affected)
```

The following code sample uses the xmlrepresentation built-in function to query only the File Access documents that are parsed XML (rather than other sorts of external files):

```
select filename, xmlextract("//book/title", content)
from xmlxfsTab
where xmlrepresentation(content) = 0
filename
---------------------------------------
parsedbookstore.1.xml
<title>Seven Years in Trenton</title>
parsedbookstore.2.xml
<title>Modern Database Management</title>

(2 rows affected)
```

Example: 'xmlerror' option capabilities with External File Access

An external (O/S) file system may contain a variety of data formats, and may contain both valid and invalid XML documents. You can use the xmlerror option of the xmlextract and xmltest functions to specify error actions for documents that are not valid XML.

For example, a File Access directory structure may contain *picture.jpg* and *nonxmldoc.txt* files along with *bookstore1.xml* and *bookstore.2.xml* files:

```
select filename from xmlxfsTab
filename
-----------------------------------------
picture.jpg
bookstore.1.xml
bookstore.2.xml
nonxmldoc.txt

(4 rows affected)
```

The following code sample shows an XML query on both XML and non-XML data:

```
select filename, xmlextract("//book/title",content)
from xmlxfsTab
--------------
Msg 14702, Level 16, State 0:
Line 1:
XMLEXTRACT(): XML parser fatal error <<An exception occurred!
Type:TranscodingException,
Message:An invalid multi-byte source text sequence was
encountered>> at line 1, offset 1.
```

Example: specifying the 'xmlerror=message’ option in xmlextract

In this example, we specify the 'xmlerror= message' option in the xmlextract call. This will return the XML query results for XML documents that are valid XML, and return an XML error message element for documents that are not valid XML.

```
select filename, xmlextract("//book/title",content
     option 'xmlerror = message') from xmlxfsTab
filename
-------------------
picture.jpg
<xml_parse_error>An exception occurred!
Type:TranscodingException,
Message:An invalid multi-byte source text sequence was
encountered</xml_parse_error>

bookstore.1.xml
<title>Seven Years in Trenton</title>

bookstore.2.xml
<title>Modern Database Management</title>
nonxmldoc.txt
<xml_parse_error>Invalid document structure</xml_parse_error>

(4 rows affected)
```

Example: parsing XML and non-XML documents with the 'xmlerror=message' option

This code sample specifies the 'xmlerror= message' option in the xmlparse call. This will store the parsed XML for XML documents that are valid XML, and store a parsed XML error message element for documents that are not valid XML.

```
insert xmlxfsTab(filename, content)
select 'ParsedDir/'+filename, xmlparse(content option
      'xmlerror = message')
from xmlxfsTab
--------------

(4 rows affected)
```

The following code sample applies the xmlextract built-in function on parsed data and gets the list of non-XML data, along with exception message information.

```
select filename, xmlextract('/xml_parse_error', content)
from xmlxfsTab
where '/xml_parse_error' xmltest content and filename like 'ParsedDir%'
----------------
Or with xmlrepresentation builtin
select filename, xmlextract('/xml_parse_error', content)
from xmlxfsTab
where xmlrepresentation(content) =  0
and '/xml_parse_error' xmltest content
filename
----------------------------------
ParsedDir/picture.jpg
<xml_parse_error>An exception occurred!
Type:TranscodingException,
Message:An invalid multi-byte source text sequence was
encountered</xml_parse_error>

ParsedDir/nonxmldoc.txt

<xml_parse_error>Invalid document structure
</xml_parse_error>

(2 rows affected)
```

Example: using the option 'xmlerror=null' for non-XML documents

The following code sample specifies the 'xmlerror = null' option with a File Access table:

```
select filename, xmlextract("//book/title", content
     option 'xmlerror = null')
from xmlxfsTab
filename
----------------------------
picture.jpg
NULL
bookstore.1.xml
<title>Seven Years in Trenton</title>

bookstore.2.xml
<title>Modern Database Management</title>
nonxmldoc.txt
NULL

(4 rows affected)
```

The following code sample selects the list of non-XML documents names with 'xmlerror = null' option.

```
select filename from xmlxfsTab
where '/' not xmltest content
     option 'xmlerror = null'
filename
-----------------------------
picture.jpg
nonxmldoc.txt

(2 rows affected)
```

APPENDIX C Setting up XML Services

This appendix provides instructions for setting up both the integrated XML processor and the Java-based processor.

Installing the Java-based SQLX mapping functions

Since the functions in Chapter 4, "XML Mapping Functions" are Java-based, you must install them in the server before you can use them. This section provides instructions for installing the Java-based functions.

Java-based XML functions

These functions must be installed in the server before you can use them:

- forxmlj
- forxmldtdj
- forxmlschemaj
- forxmlallj
- forsqlcreatej
- forsqlinsertj
- forsqlscriptj

You can find guidelines and setup scripts for installing these facilities, together with source code and JavaDoc for them, in the following directory:

$SYBASE/$SYBASE_ASE/sample

Mapping function installation

To install the Java-based SQLX mapping functions, follow the procedures outlined in this section.

Environment variables

The environmental variables in Table C-1 already exist in the server utilities.

Table C-1: Environmental variables

Variable	Value
$ISERVER	"-S" parameter for isql and installjava utilities
$INTERFACES	"-I" parameter for isql and installjava utilities
$DB	"-D" parameter for isql and installjava utilities

Installing the parser

Install the Java-based XML parser, using either the make install-xerces command in the *setup* directory referenced in the directory *$SYBASE/$SYBASE_ASE/sample*, or a client utility command such as the following:

```
installjava -f $SYBASE/$SYBASE_ASE/lib/xerces.jar\
    -j "xerces_jar"\
    -D $DB  -S $ISERVER -I $INTERFACES\
     -update  -Usa -P""
```

Note The Java-based XML parser is needed for forsqlcreatej, forsqlinsertj, and forsqlscriptj; it is not needed for forxmlj, forxmldtdj, forxmlschemaj, or forxmlallj.

Installing the mapping functions

Install the Java-based SQLX mapping classes, using either the make install-sqlx command in the *setup* directory referenced in *$SYBASE/$SYBASE_ASE/sample*, or a client utility command such as the following.

```
installjava
     -f.. /SQLX-examples/sqlx.jar -j"sqlx_jar"\
     -D $DB -S $ISERVER -I $INTERFACES
```

```
                  -update -Usa -P"
```

Creating alias names

You can create SQL alias names for the Java methods of the SQLX mapping classes, using either the make sqlx-aliases command in the *setup* directory referenced in *$SYBASE/$SYBASE_ASE/sample*, or server SQL commands such as the following:

```
create procedure forxmlallj
  (queryparmparm java.lang.String, optionparm
         java.lang.String,
     out rsout java.lang.String,
     out schemaout java.lang.String,
     out dtdout java.lang.String )
 language java parameter style java
external name "jcs.sqlx.ForXml.forXmlAll"

create function forxmlj
  (queryparm java.lang.String, optionparm
         java.lang.String)
  returns java.lang.String
  language java parameter style java
  external name "jcs.sqlx.ForXml.forXml"

create function forxmlschemaj
  (queryparm java.lang.String,optionparm
         java.lang.String)
  returns java.lang.String
  language java parameter style java
  external name "jcs.sqlx.ForXml.forXmlSchema"

create function forxmldtdj
  (queryparm java.lang.String, optionparm
        java.lang.String)
  returns java.lang.String
  language java parameter style java
  external name "jcs.sqlx.ForXml.forXmlDTD"

create function forsqlcreatej
  (schemax java.lang.String, optionparm
        java.lang.String)
  returns java.lang.String
  language java parameter style java
  external name "jcs.sqlx.SqlxCommand.forSqlCreate"
```

```
create function forsqlinsertj
  (inDoc java.lang.String, optionparm java.lang.String)
  returns java.lang.String
  language java parameter style java
  external name "jcs.sqlx.SqlxCommand.forSqlInsert"

create function forsqlscriptj
 (schemax java.lang.String, inDoc java.lang.String,
  optionparm java.lang.String)
  returns java.lang.String
  language java parameter style java
  external name "jcs.sqlx.SqlxCommand.forSqlScript"
```

APPENDIX D

The Java-Based XQL Processor

This chapter describes how you use XQL to select raw data from Adaptive Server, using the XQL language, and display the results as an XML document.

XML Services provides a Java-based XQL processor. The Java-based XQL processor implements the XQL language, which is an extension of XPath.

The Java-based XQL processor is a preliminary implementation of XPath-based XML query facilities. Its capabilities are superseded by those of the native XML processor.

You can either install the Java-based XQL processor in the server, or run it outside the server. Running it outside the server is like running any Java program on the command line.

This appendix first addresses running the Java-based XQL processor as a standalone program, outside the Adaptive Server, and then addresses running it inside the Adaptive Server.

Setting up the Java-based XQL processor

Before using the Java-based XQL processor, you must set the classpath variable, install the processor, and set the memory requirements.

Setting the CLASSPATH environment variable

To create a standalone program outside Adaptive Server, you must set your CLASSPATH environment variable to include the directories that contain *xerces.jar* and *xml.zip*. For UNIX , enter:

```
setenv CLASSPATH $SYBASE/$SYBASE_ASE/lib/xerces.jar
```

```
$SYBASE/_SYBASE_ASE/lib/xml.zip
```

For Windows, enter:

```
set CLASSPATH = D:\%SYBASE%\_SYBASE_ASE\lib\xerces.jar
D:\%SYBASE%\%SYBASE_ASE%\lib\xml.zip
```

Installing the Java-based XQL processor in Adaptive Server

This section assumes you have already enabled Java in Adaptive Server. For information on enabling Java, see Java in Adaptive Server® Enterprise.

installjava copies a JAR file into Adaptive Server and makes the Java classes in that JAR file available for use in the current database. The syntax is:

```
installjava
 -f file_name
 [-new | -update ]
 ...
```

Where:

- *file_name* is the name of the JAR file you are installing in the server.
- new informs the server this is a new file.
- update informs the server you are updating an existing JAR file.

For more information about installjava, see the Utility Guide.

To add support for XML in Adaptive Server, you must install the *xml.zip* and *xerces.jar* files. These files are located in the directories *$SYBASE/_SYBASE_ASE/lib/xml.zip* and *$SYBASE/_SYBASE_ASE./lib/xerxes.jar*

For example, to install *xml.zip*, enter:

```
installjava -Usa -P -Sserver_name -f $SYBASE/_SYBASE_ASE/lib/xml.zip
```

To install *xerces.jar*, enter:

```
installjava -Usa -P -Sserver_name -f $SYBASE/_SYBASE_ASE/lib/xerces.jar
```

Note To install *xerces.jar* in a database, you must increase the size of tempdb by 10MB.

Memory requirements for running the Java-based XQL processor inside Adaptive Server

Depending on the size of the XML data you want to reference with the Java-based XQL processor, you may need to increase memory. For a typical XML document of size 2K, Sybase recommends that you set the configuration parameters in Java Services to the values shown in Table D-1. For more information on configuration parameters, see the *Sybase Adaptive Server System Administration Guide*.

Table D-1: Java Services memory parameters

Section	Reset value
enable java	1
size of process object heap	5000
size of shared class heap	5000
size of global fixed heap	5000

Using the Java-based XQL processor

Converting a raw XML document to a parsed version

Use the parse() method to convert and parse a raw text or image XML document and store the result. Use the alter table command to convert the raw XML document. For example:

```
alter table XMLTEXT add xmldoc IMAGE null
update XMLTEXT
set xmldoc = com.sybase.xml.xql.Xql.parse(xmlcol)
```

This example converts the xmlcol column of the XMLTEXT table to parsed data and stores it in the xmldoc column.

Inserting XML documents

Use the parse() method to insert an XML document, which takes the XML document as the argument and returns sybase.aseutils.SybXmlStream.

Adaptive Server has an implicit mapping between image or text data and InputStream. You can pass image or text columns to parse() without doing any casting. The parse() UDF parses the document and returns sybase.ase.SybXmlStream, which Adaptive Server uses to write the data to the image column. Adaptive Server writes this data to image columns only, not to text columns. The following is an insert statement, where XMLDAT is a table with an image column xmldoc:

```
insert XMLDAT
values (...,
com.sybase.xml.xql.Xql.parse("<xmldoc></xmldoc>"),
     ...)
```

Updating XML documents

To update a document, delete the original data and then insert the new data. The number of updates to a document or portion of a document are infrequent compared to the number of reads. An update is similar to:

```
update XMLDAT
set xmldoc =
com.sybase.xml.xql.Xql.parse("<xmldoc></xmldoc>")
```

Deleting XML documents

Deleting an XML document is similar to deleting any text column. For example, to delete a table named XMLDAT, enter:

```
delete XMLDAT
```

Using XQL

XML Query Language (XQL) has been designed as a general-purpose query language for XML. XQL is a path-based query language for addressing and filtering the elements and text of XML documents, and is a natural extension to SPath. XQL provides a concise, understandable notation for pointing to specific elements and for searching for nodes with particular characteristics. XQL navigation is through elements in the XML tree.

The most common XQL operators include:

- Child operator, / – indicates hierarchy. The following example returns *<book>* elements that are children of *<bookstore>* elements from the xmlcol column of the xmlimage table:

  ```
  select
  com.sybase.xml.xql.Xql.query("/bookstore/book",
  xmlcol)
  from xmlimage
  ```

- Descendant operator, // – indicates that the query searches through any number of intervening levels. That is, a search using the descendant operator finds an occurrence of an element at any level of the XML structure. The following query finds all the instances of *<emph>* elements that occur in an *<excerpt>* element:

```
select com.sybase.xml.xql.Xql.query
   ("/bookstore/book/excerpt//emph",xmlcol)
from xmlimage

<xql_result>
       <emph>I</emph>
</xql_result>
```

- Equals operator, = – specifies the content of an element or the value of an attribute. The following query finds all examples where “last-name = Bob”:

```
select com.sybase.xml.xql.Xql.query
   ("/bookstore/book/author[last-name='Bob']", xmlcol)
from xmlimage

<xql_result>
       <author>
       <first-name>Joe</first-name>
       <last-name>Bob</last-name>
       <award>Trenton Literary Review Honorable Mention</award>
     </author> <author>
       <first-name>Mary</first-name>
       <last-name>Bob</last-name>
       <publication>Selected Short Stories of
       <first-name>Mary</first-name>
       <last-name>Bob</last-name></publication></author>
       <author>
       <first-name>Toni</first-name>
       <last-name>Bob</last-name>
       <degree from=Trenton U>B.A.</degree>
       <degree from=Harvard>Ph.D.</degree>
       <award>Pulizer</award>
```

```
       <publication>Still in Trenton</publication>
       <publication>Trenton Forever</publication></author>
</xql_result>
```

- Filter operator, [ ] – filters the set of nodes to its left, based on the conditions inside the brackets. This example finds any occurrences of authors whose first name is Mary that are listed in a book element:

```
select com.sybase.xml.xql.Xql.query
    ("/bookstore/book[author/first-name = 'Mary']", xmlcol)
from xmlimage
<xql_result>
       <book style=textbook>
       <title>History of Trenton</title>
       <author>
       <first-name>Mary</first-name>
       <last-name>Bob</last-name>
       <publication>Selected Short Stories of
       <first-name>Mary</first-name>
       <last-name>Bob</last-name></publication></author>
<price>55</price></book>
```

- Subscript operator, [*index_ordinal*] – finds a specific instance of an element. This example finds the second book listed in the XML document. Remember that XQL is zero-based, so it begins numbering at 0:

```
select com.sybase.xml.xql.Xql.query("/bookstore/book[1]", xmlcol)
from xmlimage
Query  returned true and the  result is
<xql_result>
              <book style=textbook>
              <title>History of Trenton</title>
              <author>
              <first-name>Mary</first-name>
              <last-name>Bob</last-name>
              <publication>Selected Short Stories of
              <first-name>Mary</first-name>
              <last-name>Bob</last-name></publication></author>
              <price>55</price></book>
</xql_result>
```

- Boolean expressions – you can use Boolean expressions within filter operators. For example, this query returns all <*authors*> elements that contain at least one <*degree*> and one <*award*>.

```
select com.sybase.xml.xql.Xql.query
("/bookstore/book/author[degree and award]", xmlcol)
from xmlimage
```

```
<xql_result>
        <author>
        <first-name>Toni</first-name>
        <last-name>Bob</last-name>
        <degree from=Trenton U>B.A.</degree>
        <degree from=Harvard>Ph.D.</degree>
        <award>Pulizer</award>
        <publication>Still in Trenton</publication>
        <publication>Trenton Forever</publication></author>
</xql_result>
```

Query structures that affect performance

This section describes examples that use the Java-based XQL processor in different ways.

Examples

The placement of the where clause in a query affects processing. For example, this query selects all the books whose author's first name is Mary:

```
select com.sybase.xml.xql.Xql.query
     ("/bookstore/book[author/first-name ='Mary']", xmlcol)
from XMLDAT
where
   com.sybase.xml.xql.Xql.query
    ("/bookstore/book
    [author/first-name= ‘Mary’]", xmlcol)!=
   convert(com.sybase.xml.xql.Xql, null)>>EmptyResult
----------------------------------------------------
<xql_result ><book style="textbook">
     <title>History of Trenton</title>
     <author>
     <first-name>Mary</first-name>
     <last-name>Bob</last-name>
     <publication>
     Selected Short Stories of
     <first-name>Mary</first-name>
     <last-name>Bob</last-name>
     </publication>
     </author>
    <price>55</price>
```

```
</book></xql_result>
```

Other usages of the Java-based XQL processor

Note Sybase does not support these usages of the XQL package. These usages require JDK 1.2 or higher.

You can query XML documents from the command line, using the standalone application com.sybase.xml.xql.XqlDriver.

You can use Java package methods provided in com.sybase.xml.xql.Xql to query XML documents in Java applications. You can also use these Java package methods to query XML documents in Adaptive Server, using the Java VM feature.

com.sybase.xml.xql.XqlDriver can parse and query only XML documents stored as files on your local system. You cannot use com.sybase.xml.xql.XqlDriver to parse or query XML documents stored in a database or over the network.

com.sybase.xml.xql.XqlDriver can be useful for developing XQL scripts and learning XQL. However, Sybase recommends that you use com.sybase.xml.xql.XqlDriver only as a standalone program, and not as part of another Java application, because com.sybase.xml.xql.XqlDriver includes a main() method. A Java program can only include one main() method, and if you include com.sybase.xml.xql.XqlDriver in another Java program that includes main(), the application attempts to implement both main() methods, which causes an error in Java.

Sybase recommends that applications use the com.sybase.xml.xql.Xql class to interface with the XML query engine. The methods of this class are specified in the section “Methods in com.sybase.xml.xql.Xql” on page 181.

com.sybase.xml.xql.XqlDriver syntax

The syntax for com.sybase.xml.xql.XqlDriver is:

java com.sybase.xml.xql.XqlDriver
-qstring *XQL_query*
-validate true | false
-infile *string*

-outfile *string*
-help
-saxparser *string*

Where:

- qstring specifies the XQL query you are running.
- validate checks the validity of the XML documents.
- infile is the XML document you are querying.
- outfile is the operating system file where you are storing the parsed XML document.
- help displays the com.sybase.xml.xql.XqlDriver syntax.
- saxparser specifies the name of a CLASSPATH parser that is compliant with SAX 2.0.

Sample queries

This query selects all the book titles from *bookstore.xml*:

```
java com.sybase.xml.xql.XqlDriver -qstring "/bookstore/book/title"
     -infile bookstore.xml

Query  returned true and the  result is

<xql_result>
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>
</xql_result>
```

This example lists all the author's first names from *bookstore.xml*. XQL uses a zero-based numbering system; that is, "0" specifies the first occurrence of an element in a file.

```
java com.sybase.xml.xql.XqlDriver
     -qstring "/bookstore/book/author/first-name[0]"
     -infile bookstore.xml
Query  returned true and the  result is

<xql_result>
          <first-name>Joe</first-name>
          <first-name>Mary</first-name>
          <first-name>Toni</first-name>
</xql_result>
```

The following example lists all the authors in *bookstore.xml* whose last name is “Bob”:

```
java com.sybase.xml.xql.XqlDriver
     -qstring "/bookstore/book/author[last-name='Bob']"
     -infile bookstore.xml
Query  returned true and the  result is

<xql_result>
     <author>
     <first-name>Joe</first-name>
     <last-name>Bob</last-name>
     <award>Trenton Literary Review Honorable Mention</award></author>
     <author>
     <first-name>Mary</first-name>
     <last-name>Bob</last-name>
     <publication>Selected Short Stories of
     <first-name>Mary</first-name>
     <last-name>Bob</last-name></publication></author>
     <author>
     <first-name>Toni</first-name>
     <last-name>Bob</last-name>
     <degree from=Trenton U>B.A.</degree>
     <degree from=Harvard>Ph.D.</degree>
     <award>Pulizer</award>
     <publication>Still in Trenton</publication>
     <publication>Trenton Forever</publication></author>
</xql_result>
```

Validating your document

The valid option invokes a parser that makes sure the XML document you are querying conforms to its DTD. Your standalone XML document must have a valid DTD before you run the validate option.

For example, this command makes sure the *bookstore.xml* document conforms to its DTD:

```
java com.sybase.xml.xql.XqlDriver -qstring "/bookstore" -validate
     -infile bookstore.xml
```

Using the Java-based XQL processor for standalone applications

You can use XQL to develop standalone applications, JDBC clients, JavaBeans, and EJBs to process XML data. The query() and parse() methods in com.sybase.xml.xql.Xql enable you to query and parse XML documents. Because you can write standalone applications, you do not have to depend on Adaptive Server to supply the result set. Instead, you can query XML documents stored as operating system files or stored out on the Web.

Example standalone application

The following example uses the FileInputStream() query to read *bookstore.xml*, and the URI() method to read a Web page named *bookstore.xml* which contains information about all the books in the bookstore:

```
String result;
FileInputStream XmlFile = new FileInputStream("bookstore.xml");
if ((result =
         Xql.query("/bookstore/book/author/first-name", XmlFile))
         != Xql.EmptyResult )
{
       System.out.println(result);
}else{
       System.out.println("Query returned false\n");
}
URI _uri = new URI("http://mybookstore/bookstore.xml");
if ((result =
         Xql.query("/bookstore/book/author/first-name",uri.openStream()))
         != Xql.EmptyResult )
{
       System.out.println(result);
}else{
       System.out.println("Query returned false\n");}
```

Example EJB example

You can write EJB code fragments that serve as query engines on an EJB server.

The code fragment below includes an EJB called *XmlBean*. *XmlBean* includes the query() method, which allows you to query any XML document on the Web. In this component, query() first creates an XmlDoc object, then queries the document.

The remote interface looks like:

```
public interface XmlBean extends javax.ejb.EJBObject
{
      /**
        * XQL Method*/
     public String XQL(String query, URI location)
     throwsjava.rmi.RemoteException;}
```

The Bean implementation looks like:

```
public class XmlBean extends java.lang.Object implements
javax.ejb.SessionBean
{
      ....
      /***
      * XQL Method
      */
     public String XQL(String query, java.net.URI location) throws
         java.rmi.RemoteException
{
try {
      String result;
      if((result =
       Xql.query(query, location.openStream())) !=
       Xql.EmptyResult)
{
       return (result);
       }else{
return (null);
       }
       }catch(Exception e){
            throw new java.rmi.RemoteException(e.getMessage()));
                    }
....}
}
```

And the client code looks like:

```
....Context ctx = getInitialContext();
// make the instance of the class in Jaguar
XmlBeanHome -beanHome =
(XmlBeanHome)ctx.lookup(“XmlBean”);
_xmlBean = (XmlBean)_beanHome.create();
URI u = new URI("http://mywebsite/bookstore.xml");
String res= xmlBean.XQL("/bookstore/book/author/first-name",u);
```

Methods in com.sybase.xml.xql.Xql

The following methods are specific to com.sybase.xml.xql.Xql.

parse(String xmlDoc)

Description

Takes a Java string as an argument and returns *SybXmlStream*. You can use this to query a document using XQL.

Syntax

parse(**String** *xml_document*)

Where:

- String is a Java string.
- *xml_document* is the XML document where the string is located.

Examples

This example returns *SybXmlStream*:

```
SybXmlStream xmlStream = Xql.parse("<xml>..</xml>")
```

Usage

The parser does not:

- Validate the document if a DTD is provided.
- Parse any external DTDs
- Perform any external links (for example, XLinks)
- Navigate through IDREFs

parse(InputStream xml_document, boolean validate)

Description

Takes an InputStream and a boolean flag as arguments.The flag indicates that the parser should validate the document according to a specified DTD. Returns *SybXmlStream*. You can use this to query a document using XQL.

Syntax

parse(**InputStream** *xml_document, boolean validate)*

Where:

- *InputStream* is an input stream.
- *xml_document* is the XML document where the input stream originates.

Examples

This example returns *SybXmlStream*

```
SybXmlStream is = Xql.parse(new
FileInputStream("file.xml"), true").
```

Usage

- A true value in the flag indicates that the parser should validate the document according to the specified DTD.
- A false value in the flag indicates that the parser does not validate the document according to the specified DTD.
- The parser does not:
    - Parse any external DTDs
    - Perform any external links (for example, XLinks)
    - Navigate through IDREFs

query(String query, String xmlDoc)

Description

Queries an XML document. Uses the XML document as the input argument.

Syntax

query(**String query,***String xmlDoc*)

Where:

- *String query* is the string you are searching for.
- *String xmldoc* is the XML document you are querying.

Examples

The following returns the result as a Java string:

```
String result= Xql.query("/bookstore/book/author",
"<xml>...</xml>");
```

Usage

Returns a Java string.

query(String query, InputStream xmlDoc)

Description

Queries an XML document using an input stream as the second argument.

Syntax

query(**String query,*InputStream xmlDoc***)

Where:

- *String query* is the string you are searching for.
- *Input Stream xmlDoc* is the XML document you are querying.

Examples

This example queries the bookstore for authors listed in *bookstore.Xql*.

```
FileInputStream xmlStream = new FileInputStream("doc.xml");
String result = Xql.query("/bookstore/book/author", xmlStream);
```

The following example queries an XML document on the Web using a URI as the search argument:

```
URI xmlURI = new URI("http://mywebsite/doc.xml");
String result = Xql.query("/bookstore/book/author", xmlURI.openStream());
```

Usage

Returns a Java string.

query(String query, SybXmlStream xmlDoc)

Description

Queries the XML document using a parsed XML document as the second argument.

Syntax

query(String query, *SybXmlStream*)

Where:

- *String query* is the string you are searching for.
- *xmldoc* is the parsed XML document you are querying.

Examples

This example queries the bookstore for authors listed in *bookstore.Xml*.

```
SybXmlStream xmlStream = Xql.parse("<xml>..</xml>");
String result = Xql.query("/bookstore/book/author",xmlStream);
```

sybase.aseutils.SybXmlStream

Description

Defines an interface that an InputStream needs to access parsed XML data while querying.

Syntax

sybase.aseutils.SybXmlStream interface

com.sybase.xml.xql.store.SybMemXmlStream

Description

Holds the parsed XML document in main memory, an implementation of SybXMLStream that Sybase provides.

Syntax

com.sybase.xml.xql.store.SybMemXmlStream

Usage

The parse() method returns an instance of SybMemXmlStream after parsing an XML document.

com.sybase.xml.xql.store.SybFileXmlStream

Description

Allows you to query a file in which you have stored a parsed XML document.

Syntax

com.sybase.xml.xql.store.SybFileXmlStream {*file_name*}

Where *file_name* is the name of the file in which you stored the parsed XML document.

Examples

In the following, a member of the RandomAccessFile reads a file and positions the data stream:

```
SybXmlStream xis = Xql.parse("<xml>..</xml>");
FileOutputStream ofs = new FileOutputStream("xml.data");
((SybMemXmlStream)xis).writeToFile(ofs);

SybXmlStream is = new SybFileXmlStream("xml.data");
String result = Xql.query("/bookstore/book/author", is);
```

setParser(String parserName)

Description

This static method specifies the parser that the parse method should use. You should make sure that the specified parser class is accessible through the CLASSPATH and is compliant with SAX 2.0.

Syntax

setParser *(String parserName)*

Where *string* is the name of the parser class.

Examples

```
Xql.setParser("com.yourcompany.parser")
```

resetParser

Description

This static method resets the parser to the default parser that Sybase supplies (*xerces.jar*).

Syntax

resetParser

Examples

This example resets your parser to the Sybase default parser.

```
xql.resetParser()
```

APPENDIX E Migrating Between the Java-based XQL Processor and the Native XML Processor

Introduction

The Java-based XQL processor and the native XML processor both implement query languages and return documents in parsed form, but they use different functions and methods.

- The native XML processor implements XML query language. It provides a built-in function, xmlparse, that returns, in parsed form, a document suitable for efficient processing with the xmlextract and xmltext built-in functions.
- The Java-based XQL processor is an earlier facility that implements the XQL query language. It provides a Java method, com.sybase.xml.xql.Xql.parse, that returns a parsed form of a document that is a sybase.aseutils.SybXmlStream object, suitable for processing with the com.sybase.xml.xql.Xql.query method.

If you want to migrate documents between the Java-based XQL processor and the native XML processor, you should be aware of the following possibilities and restrictions:

- Documents in text form can be processed directly by both the Java-based XQL processor and the native XML processor.
- The sybase.aseutils.SybXmlStream documents generated by com.sybase.xml.xql.Xql.parse can only be processed by the Java-based XQL processor. They cannot be processed by the built-in functions xmlextract or xmltest.
- The parsed documents generated by the xmlparse built-in function can only be processed by the xmlextract and xmltest built-in functions. They cannot be processed by the Java-based XQL processor.

Migrating documents and queries

The following sections describe techniques for migrating documents and queries between the Java-based XQL processor and the native XML processor.

Migrating documents between the Java-based XQL processor and the native XML processor

There are two approaches you can use to migrate documents between the Java-based XQL processor to the native XML processor:

- You can use the text form of the documents, if it is available.
- You can generate a text version of the documents from the parsed form of the documents.

Migrating text documents between the Java-based XQL processor and the native XML processor

Suppose that you have a table such as the following, in which you have stored the text form of documents in the xmlsource column:

```
create table xmltab (xmlsource text, xmlindexed image)
```

If you want to process the documents with the native XML processor, using the xmlextract and xmltest built-in functions, you can update the table as follows:

```
update xmltab
set xmlindexed = xmlparse(xmlsource)
```

If you want to process the documents with the Java-based XQL processor, using the com.sybase.xml.xql.Xql.query method, you can update the table as follows:

```
update xmltab
set xmlindexed
    = com.sybase.xml.xql.Xql.parse(xmlsource)
```

Migrating documents from regenerated copies

Suppose that you have stored only parsed forms of some documents, using either the xmlparse built-in function for the native XML processor or the com.sybase.xml.xql.Xql.parse method for the Java-based XQL processor. For example, you might have such documents in a table as the following:

```
create table xmltab (xmlindexed image)
```

If you want to regenerate the text for such documents, you can alter the table to add a text column:

```
alter table  xmltab add xmlsource text null
```

Regenerating text documents from the Java-based XQL processor

This section demonstrates regenerating the text form of the documents from the form generated for the Java-based XQL processor.

If the xmlindexed column contains sybase.aseutils.SybXmlStream data generated by com.sybase.xmlxql.Xql.parse, you can regenerate the text form of the document in the new xmlsource column with the following SQL statement:

```
update xmltab
set xmlsource
   = xmlextract("/xql_result/*",
      com.sybase.xml.xql.Xql.query("/",xmlindexed) )
```

This statement generates text form of the document in two steps:

1. The com.sybase.xml.xql.Xql.query call with the "/" query generates a text form of the document, enclosed in an XML tag <*xql_result*>...</*xql_result*>.
2. The xmlextract call with the "/xql_result/*" query removes the <*xql_result*>...</*xql_result*> tag, and returns the text form of the original document.

You can then process the xmlsource column directly with the native XML processor, using the xmlextract and xmltest built-in functions, or you can update the xmlindexed column for the native XML processor, as follows:

```
update xmltab
set xmlindexed = xmlparse(xmlsource)
```

If you don't want to add the xmlsource column, you can combine these steps, as in the following SQL statement:

```
update xmltab
set xmlindexed
  = xmlparse(xmlextract("/xql_result/*",
      com.sybase.xml.xql.Xql.query("/",xmlindexed) ) )
```

Before this update statement is executed, the xmlindexed column contains the sybase.aseutiles.SybXmlStream form of the documents, generated by the com.sybase.xml.xql.Xql.parse method. After the update statement, that column contains the parsed form of the documents, suitable for processing with the xmlextract and xmlparse methods.

Regenerating text documents from the native XML processor

This section demonstrates regenerating the text form of the documents from the form generated for the native XML processor.

If the xmlindexed column contains data generated by the xmlparse function, you can regenerate the text form of the document in the new xmlsource column with the following SQL statement:

```
update xmltab
set xmlsource  = xmlextract("/",  xmlindexed)
```

You can then

- process the xmlsource column directly with the Java-based XQL processor, using com.sybase.xml.xql.Xql.query, OR
- update the xmlindexed column with the parsed form suitable for processing with the Java-based XQL processor, using the following statement:

```
update xmltab
set xmlindexed
  = com.sybase.xml.xql.Xql.parse(xmlsource)
```

If you don't want to add the xmlsource column, you can combine these steps, as in the following SQL statement:

```
update xmltab
set xmlindexed
    = com.sybase.xml.xql.Xql.parse
      (xmlextract("/", xmlindexed))
```

Before this update statement is executed, the xmlindexed column contains the parsed form of the documents, generated by the xmlparse built-in function. After the update statement, that column contains the parsed form of the documents, generated by com.sybase.xml.xql.Xql.parse, suitable for processing with com.sybase.xml.xql.Xql.query.

Migrating queries between the native XML processor and the Java-based XQL processor

The XQL language implemented by the Java-based XQL processor and the XML Query language implemented by the native XML processor are both based on the XPath language. There are two primary differences between them:

- Subscripts begin with "1" in the XML Query language, and with "0" in the XQL Language.
- The Java-based XQL processor returns results enclosed in "<*xql_result*>...</*xql_result*>" tags, and the native XML processor does not.

APPENDIX F

Sample Application for *xmltable()*

Sample table

This section shows a sample XML document, depts.xml, which illustrates an application of xmltable().

```
<sample>
<depts>
   <dept>
       <dept_id>D123</dept_id>
       <dept_name>Main</dept_name>
   <emps>
    <emp>
       <emp_id>E123</emp_id>
       <emp_name>Alex Allen</emp_name>
       <salary>912.34</salary>
       <phones>
          <phone><phone_no>510.555.1987</phone_no></phone>
          <phone><phone_no>510.555.1867</phone_no></phone>
       </phones>
    </emp>
   <emp>
       <emp_id>E234</emp_id>
       <emp_name>Bruce Baker</emp_name>
       <salary>923.45</salary>
       <phones>
        <phone><phone_no>230.555.2333</phone_no></phone>
         </phones>
  </emp>
   <emp>
       <emp_id>E345</emp_id>
       <emp_name>Carl Curtis</emp_name>
       <salary>934.56</salary>
       <phones>
         <phone><phone_no>408.555.3123</phone_no></phone>
         <phone><phone_no>415.555.3987</phone_no></phone>
        <phone><phone_no>650.555.3777</phone_no></phone>
       </phones>
   </emp>
  </emps>
<emps_summary>
<salary_summary>
       <max_salary>934.56</max_salary>
       <total_salary>2770.35</total_salary>
</salary_summary>
</emps_summary>
<projects>
<project>
   <project_id>PABC</project_id>
   <budget>598.65</budget>
</project>
<project>
   <project_id>PBDC</project_id>
   <budget>587.65</budget>
</project>
<project>
   <project_id>PCDE</project_id>
   <budget>576.54</budget>
</project>

  </projects>
  <projects_summary>
 <budget_summary>
   <max_budget>598.76</max_budget>
   <total_budget>1762.95</total_budget>
 </budget_summary>
  </projects_summary>
</dept>
<dept>
       <dept_id>D234</dept_id>
       <dept_name>Auxiliary</dept_name>
   <emps>
    <emp>
       <emp_id>E345</emp_id>
       <emp_name>Don Davis</emp_name>
       <salary>945.67</salary>
       <phones>
     <phone><phone_no>650.555.5001</phone_no></phone>
        </phones>
     <emp>
       <emp_id>E345</emp_id>
       <emp_name>Earl Evans</emp_name>
       <phones>
     <phone><phone_no>650.555.5001</phone_no></phone>
```

```
        </phones>

    </emp>
  </emps>
<emps_summary>
<salary_summary>
       <max_salary>945.67</max_salary>
       <total_salary>945.67</total_salary>
</salary_summary>

</emps_summary>
<projects>
 <project>

 <project>
   <project_id>PDEF</project_id>
< /project>
 <project>
   <project_id>PEFG</project_id>
   <budget>554.32</budget>
</project>
  </project>
  </projects>
  <projects_summary>
 <budget_summary>
   <max_budget>554.32</max_budget>
   <total_budget>554.32</total_budget>
 </budget_summary>
  </projects_summary>
</dept>
</dept>
<dept>
       <dept_id>D345</dept_id>
       <dept_name>Repair</dept_name>
   <emps>
    <emp>
       <emp_id>E678</emp_id>
       <emp_name>Fred Frank</emp_name>
       <salary>967.89</salary>
       <phones>
          <phone><phone_no>408.555.6111</phone_no></phone>
         </phones>
   </emp
   <emp>>
       <emp_id>E789</emp_id>
       <emp_name>George Gordon</emp_name>
```

```
      <salary>978.90</salary>
      <phones>
    <phone><phone_no>510.555.7654</phone_no></phone>
       </phones>
  </emp>
  <emp>
      <emp_id>E901</emp_id>
      <emp_name>Hank Hartley</emp_name>
      <salary>990.12</salary>
      <phones\>
  </emp
  <emp>
      <emp_id>E678</emp_id>
      <emp_name>Isaak Idle</emp_name>
      <salary>990.12</salary>
      <phones>
       <phone><phone_no>925.555.9991</phone_no></phone>
       <phone><phone_no>650.555.9992</phone_no></phone>
       <phone><phone_no>415.555.9993</phone_no></phone>
       </phones>
  </emp>
      <emps>
      <emps_summary>
  <salary_summary>
      <max_salary>990.12</max_salary>
      <total_salary>2936.91</total_salary>
 </salary_summary>
 </emps_summary>
 <projects>
   <project>
     <project_id>PFGH</project_id>
     <budget>543.21</budget>
  </project>
  <project>
   <project_id>PGHI</project_id>
  </project>
  <project>
    <project_id>PHIJ</project_id>
    <budget>521.09</budget>
  </project>
  </project>
</projects>
 <projects_summary>
<budget_summary>
  <max_budget>543.21</max_budget>
  <total_budget>1064.30</total_budget>
 </budget_summary>
  </projects_summary>
</dept>
</depts>
</sample>
```

Using the *depts* document

The *depts* document is stored in a new row of the sample_docs table in Appendix A, “The sample_docs Example Table.” To reference this document in examples:

```
declare @dept_doc xml
select @dept_doc from sample_docs where name_doc='depts'
```

The *depts* document structure

The structure of the *depts* document is as follows:

```
<depts>
   <dept>
      <emps> - repeats under <depts>
       <emp> - one for each <dept>
          <emp_id> - one for each <emp>
          <emp_name> - one for each <emp>
          <phones> - one for each <emp>
            <phone> - repeats under <phones>
               <phone_no> - one for each <phone>
      <projects> - one for each <dept>
       <project> - repeats for each under <projects>
           <project_id> - one for each <project>
           <dept_id> - one for each <project>
```

Creating SQL tables from the *depts* document

Normalizing the data from the *depts* document

You can normalize this data into SQL tables. For example:

- depts – A row for each <*dept*> element, containing the <*dept_id*> and <*dept_name*>.
- emps – A row for each <*emp*> element, containing the <*emp_id*> and <*emp_name*>, as well as the <*dept_id*> element, which contains them.
- emp_phones – A row for each <*phone*>, containing the <*phone_no*> and the <*dept_id*> element, which contains all these elements.
- projects – A row for each <*project*>, containing the <*project_id*> and <*budget*> elements, and the containing <*dept_id*> element.

Generating tables using *select*

All the tables generated in this section, except the depts table, have a column pattern that uses the XPath ancestor notation to reference:

- Leaf elements, such as <*project*>, under <*projects*>, or <*salary*>, under <*emp*>.
- The element that contains the element defining the table, as <*emp*>, for example, contains <*emp_id*>>.

This notation "flattens" nested data. For more information about flattening XML data, see Chapter 7, "xmltable()".

emps table

In this select statement, the column pattern for the dept_id column references the <*dept_id*>element in the <*dept*>, which contains the current <*emp*>.

```
declare @ dept_doc xml
select @dept_doc = doc from sample_docs where name_doc = 'depts'
select * into emps from xmltable('//emp' passing @dept_doc
       columns emp_id char(4),
       emp_name varchar(50),
       salary money,
       dept_id char(4) pattern '../../dept_id') as dept_extract
select * from emps
--------------------
emp_id              emp_name                salary         dept_id
---------           -------------------     --------       --------
E123                Alex Allen              912.34         D123
E234                Bruce Baker             923.45         D123
E345                Carl Curtis             934.56         D123
E456                Don Davis               945.67         D234
E567                Earl Evans              956.78         D234
```

```
E678                Fred Frank             967.89        D345
E789                George Gordon          978.90        D345
E890                Hank Hartley           NULL          D345
E901                Isaak Idle             990.12        D345
```

phones **table**

In the phones table, the column pattern for the emp_id column references the <*emp_id*> element in <*emp*>, which contains the current element <*phone*>.

```
declare @ dept_doc xml
select @ dept_doc
   = doc from sample_docs where name_doc='depts'
select * into phones
   from xmltable('//phone' passing @ dept_doc
      columns emp_id char(4),'../../emp_id'
         phone_no varchar(50)) as dept_extract
select * from phones
---------------
emp_id                 phone_no
--------                ------------
E123                    510.555.1987
E123                    510.555.1876
E234                    203.555.2333
E345                    408.555.3123
E345                    415.555.3987
E345                    650.555.3777
E567                    650.555.5001
E678                    408.555.6111
E678                    408.555.6222
E789                    510.555.7654
E901                    925.555.9991
E901                    650.555.9992
E901                    415.555.9993
```

projects **table**

In the projects table, the column pattern for the dept_id column references the <*dept_id*> element, in the <*dept*> that contains the current <*project*>.

```
declare @ dept_doc xml
select @ dept_doc
   = doc from sample_docs where name_doc='depts'
select * into projects
   from xmltable('//project' passing @ dept_doc
      columns project_id char(4),
      budget money,
         dept_id char(4)pattern '../../dept_id')
         as dept_extract
select * from projects
---------------
project_id      budget         dept_id
----------      -------        --------
PABC            598.76         D123
PBCD            587.65         D123
PCDE            576.54         D123
PDEF            565.43         D234
PEFG            554.32         D234
PFGH            543.21         D345
PGHI            NULL           D345
PHIJ            521.09         D345
```

***depts* table**

```
declare @ dept_doc xml
select @ dept_doc
   = doc from sample_docs where name_doc='depts'
select * into depts
   from xmltable('//dept' passing @ dept_doc
      columns dept_id char(4),
         dept_name char(4)) as dept_extract
select * from depts
---------------
dept_id                 dept_name
--------                ------------
D123                    Main
D234                    Auxiliary
D345                    Repair
```

Index