XML Services
in Adaptive Server Enterprise

Adaptive Server® Enterprise
12.5.1
Contents

About This Book .......................................................................................................................... vii

CHAPTER 1 Introduction to XML Services ........................................................  1
  XML capabilities ..................................................................................................................  1
  XML in the database ...........................................................................................................  2
    Introduction.......................................................................................................................  2
    An overview of XML .......................................................................................................  2
    Character sets and XML data.........................................................................................  9

CHAPTER 2 XML Query Functions ...................................................................  11
  XML query functions ........................................................................................................  11
    Example sections ............................................................................................................  12
    xmlextract ......................................................................................................................  12
    xmltest ...........................................................................................................................  19
    xmlparse .......................................................................................................................... 22
    xmlrepresentation ......................................................................................................... 25
    option_strings: general format .................................................................................... 27

CHAPTER 3 XML Language and XML Query Language .................................  31
  Character set support ....................................................................................................... 31
  URI support ..................................................................................................................... 32
  Namespace support ......................................................................................................... 32
  XML schema support ...................................................................................................... 32
  Predefined entities in XML language ............................................................................ 33
  Predefined entities in XML query language ................................................................... 34
  White space ..................................................................................................................... 35
  Empty elements .............................................................................................................. 35
  XML Query Language .................................................................................................... 36
    XPath-supported syntax and tokens ............................................................................. 36
    XPath operators and functions .................................................................................... 37

CHAPTER 4 XML Mapping Functions...............................................................  41
# Contents

for xml clause .......................................................... 41
forxmlj, forxmltdj, forxmlschemaj, forxmlallj ................. 44
forsqrcodej, forsqlinsertj, forsqlscriptj ......................... 49

## CHAPTER 5

XML Mappings ................................................................. 53
- SQLX options .......................................................... 53
- SQLX option definitions ............................................. 54
- SQLX data mapping ................................................. 62
  - Mapping duplicate column names and unnamed columns ... 62
  - Mapping SQL names to XML names ......................... 65
  - Mapping SQL values to XML values ......................... 67
- SQLX schema mapping .............................................. 69
  - Overview ............................................................ 69
  - Option: columnstyle=element .................................. 72
  - Option: columnstyle=attribute .................................. 73
  - Option: nullstyle=omit .......................................... 73
  - Option: nullstyle=attribute ...................................... 74

## APPENDIX A

Setting up XML Services ......................................................... 77
- Enabling the native XML processor .......................... 77
- Installing the Java-based SQLX mapping functions ...... 77
- Java-based XML functions ........................................ 77
- Mapping function installation ..................................... 78

## APPENDIX B

The sample_docs Example Table ................................................ 81
- sample_docs table columns and rows ......................... 81
- Sample_docs table columns ....................................... 81
- sample_docs table rows ........................................... 82
- sample_docs tables .................................................. 84
- Table script (for publishers table) ......................... 84
- Publishers table representation .............................. 84
- Titles table representation ........................................ 85

## APPENDIX C

XML Services and External File System Access .................. 91
- Getting Started ......................................................... 91
- Enabling XML services and External File System Access .. 91
- Examples ............................................................... 92
- Setting up your XML documents and creating the proxy table 92
- Example: Extracting the book title from the XML documents .. 93
- Example Importing XML documents or XML query results to an
  Adaptive Server table ........................................ 93
- Example: Storing parsed XML documents in the file system .. 94

Example: Specifying the ‘xmlerror=’message’ option in xmlextract.

Example: Parsing XML and non-XML documents with the ‘xmlerror=’message’ option.

Example: Using the option ‘xmlerror=null’ for non-XML documents.

Introduction

Migrating documents and queries

Migrating documents between the Java-based XQL processor and the native XML processor

Migrating text documents between the Java-based XQL processor and the native XML processor

Migrating documents from regenerated copies

Regenerating text documents from the Java-based XQL processor

Regenerating text documents from the native XML processor

Migrating queries between the native XML processor and the Java-based XQL processor

APPENDIX D

Migrating Between the Java-based XQL Processor and the Native XML Processor

Introduction

Migrating documents and queries

Character set support

Migrating documents between the Java-based XQL processor and the native XML processor

Migrating text documents between the Java-based XQL processor and the native XML processor

Migrating documents from regenerated copies

Regenerating text documents from the Java-based XQL processor

Regenerating text documents from the native XML processor

Migrating queries between the native XML processor and the Java-based XQL processor

APPENDIX E

The Java-Based XQL Processor

Setting up the Java-based XQL processor

Setting the CLASSPATH environment variable

Installing the Java-based XQL processor in Adaptive Server

Memory requirements for running the Java-based XQL processor inside Adaptive Server
Using the Java-based XQL processor ............................................. 112
Converting a raw XML document to a parsed version .................. 112
Inserting XML documents .......................................................... 112
Updating XML documents ......................................................... 112
Deleting XML documents .......................................................... 113
Using XQL .............................................................................. 113
Query structures that affect performance .................................. 115
Examples ................................................................................. 116
Other usages of the Java-based XQL processor .............................. 116
com.sybase.xml.xql.XqlDriver syntax ........................................ 117
Validating your document ......................................................... 119
Using the Java-based XQL processor for standalone applications 119
Methods in com.sybase.xml.xql.Xql ............................................. 121
parse(String xmlDoc) ................................................................. 121
parse(InputStream xml_document, boolean validate) .................... 122
query(String query, String xmlDoc) .......................................... 123
query(String query, InputStream xmlDoc) ................................. 123
query(String query, SybXmlStream xmlDoc) ............................... 124
query(String query, JXml jxml) .................................................. 124
sybase.aseutils.SybXmlStream ................................................... 124
com.sybase.xml.xql.store.SybMemXmlStream ......................... 125
com.sybase.xml.xql.store.SybFileXmlStream ............................ 125
setParser(String parserName) ................................................... 125
reSetParser ........................................................................... 126

Index ........................................................................................................ 127
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,” is an introduction to XML in the database and to the new XML capabilities of the Sybase XML Services.

• Chapter 2, “XML Query Functions,” is for 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 \texttt{for xml} clause or \texttt{forxml} 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,” contains descriptions of the XML document and query language features that the XML query functions support, including specification of the XPath language subset supported.

• Chapter 4, “XML Mapping Functions,” discusses functions that map between SQL data and XML documents in the SQLX-XML format.

• Chapter 5, “XML Mappings,” is a description of the SQLX-XML format of the XML documents that the XML mapping functions support.

• Appendix A, “Setting Up XML Services,” includes guidelines for installing both the native, C++ processor and the Java processor that is 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.

The following documents provide XML-related reference material:

- XML Path Language (XPATH) Version 1.0, 16 November 1999, at http://www.w3.org/TR/xpath

Related Documents

The Sybase® 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 12.5.1, the system changes added to support those features, and the changes that may affect your existing applications.
About This Book

- **ASE Replicator User’s Guide** – describes how to use the ASE 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.
- **Configuring Adaptive Server Enterprise** for your platform – provides instructions for performing specific configuration tasks for Adaptive Server.
- **Error Messages and Troubleshooting Guide** – explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.
- **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.
- **Java in Adaptive Server Enterprise** – describes how to install and use Java classes as data types, 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).
- **Monitor Client Library Programmer’s Guide** – describes how to write Monitor Client Library applications that access Adaptive Server performance data.
- **Performance and Tuning Guide** – is a series of four books that explains how to tune Adaptive Server for maximum performance:
  - **Basics** – the basics for understanding and investigating performance questions in Adaptive Server.
• **Locking** – describes how the various locking schemas can be used for improving performance in Adaptive Server.

• **Optimizer and Abstract Plans** – describes how the optimizer processes queries and how abstract plans can be used to change some of the optimizer plans.

• **Monitoring and Analyzing** – explains how statistics are obtained and used for monitoring and optimizing performance.

• **Quick Reference Guide** – provides a comprehensive listing of the names and syntax for commands, functions, system procedures, extended system procedures, datatypes, and utilities in a pocket-sized book.

• **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** – provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources, security, user and system databases, and specifying character conversion, international language, and sort order settings.

• **System Tables Diagram** – illustrates system tables and their entity relationships in a poster format. Available only in print version.

• **Transact-SQL User’s Guide** – documents Transact-SQL, Sybase’s 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.

• **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’s Failover to configure an Adaptive Server as a companion server in a high availability system.

• **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.

### Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

2. Select Products from the navigation bar on the left.
3. Select a product name from the product list and click Go.
4. Select the Certification Report filter, specify a time frame, and click Go.
5. Click a Certification Report title to display the report.

❖ **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.

2. Click MySybase and create a MySybase profile.
Sybase EBFs and software updates

Finding the latest information on EBFs and software updates


2. Select EBFs/Updates. Enter user name and password information, if prompted (for existing Web accounts) or create a new account (a free service).

3. Select a product.

4. Specify a time frame and click Go.

5. Click the Info icon to display the EBF/Update report, or click the product description to download the software.

Java syntax conventions

This book uses these font and syntax conventions for Java items:

- Classes, interfaces, methods, and packages are shown in Helvetica within paragraph text. For example:

  `SybEventHandler` interface
  `setBinaryStream()` method
  `com.Sybase.jdbc` package

- Objects, instances, and parameter names are shown in italics. For example:

  "In the following example, `ctx` is a `DirContext` object."

  "`eventHandler` is an instance of the `SybEventHandler` class that you implement."

  "The `classes` parameter is a string that lists specific classes you want to debug."

- Java names are always case sensitive. For example, if a Java method name is shown as `Misc.stripLeadingBlanks()`, you must type the method name exactly as displayed.

Transact-SQL syntax conventions

This book uses the same font and syntax conventions for Transact-SQL as other Adaptive Server documents:

- Command names, command option names, utility names, utility flags, and other keywords are in Helvetica in paragraph text. For example:

  `select` command
About This Book

isql utility
-f flag

- Variables, or words that stand for values that you fill in, are in italics. For example:

user_name
server_name

- Code fragments are shown in a monospace font. Variables in code fragments (that is, words that stand for values that you fill in) are italicized. For example:

```java
Connection con = DriverManager.getConnection
("jdbc:sybase:Tds:host:port", props);
```

- You can disregard case when typing Transact-SQL keywords. For example, SELECT, Select, and select are the same.

Additional conventions for syntax statements in this manual are described in Table 1. You can find examples illustrating each convention in the System Administration Guide.

**Table 1: Syntax statement conventions**

<table>
<thead>
<tr>
<th>Key</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>Curly braces indicate that you choose at least one of the enclosed options. Do not include braces in your option.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets mean choosing one or more of the enclosed options is optional. Do not include brackets in your option.</td>
</tr>
<tr>
<td>( )</td>
<td>Parentheses are to be typed as part of the command.</td>
</tr>
<tr>
<td></td>
<td>The vertical bar means you may select only one of the options shown.</td>
</tr>
<tr>
<td>,</td>
<td>The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.</td>
</tr>
</tbody>
</table>

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 12.5.1.

Table 1-1: XML capabilities

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML capabilities</td>
<td>1</td>
</tr>
<tr>
<td>XML in the database</td>
<td>2</td>
</tr>
</tbody>
</table>

XML capabilities

XML Services provides the following capabilities:

- A native XML processor to run in Adaptive Server. This processor implements the XPath language, a significant subset of the XQuery language. This processor is upwardly compatible with the XQL language implemented by the Java-based SQL processor provided by Adaptive Server 12.5, so it provides a migration path for current applications.

- A for xml extension of select statements, and a Java-based forxmlj function, to map SQL result sets to the standard SQLX-XML representation. Java-based forsqlj functions are present to map a SQLX-XML document back to a SQL script that recreates the data.

The 12.5.1 XML Services feature builds on experience with the Adaptive Server 12.5 XML support, and provides a basis for further enhancement.
XML in the database

Introduction

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.

References

This chapter presents an overview of XML. For detailed information, refer to these Web documents:

- World Wide Web Consortium (W3C), at http://www.w3.org
- W3C, Document Object Model (DOM), at http://www.w3.org/DOM/
- W3C, Extensible Markup Language (XML), at http://www.w3.org/XML/

An overview of XML

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

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, XML has characteristics that make it better able to share documents:

- XML documents possess a strict phrase structure that makes it easy to find and access data. For example, opening tags of all elements must have both an opening tag and a corresponding closing tag, for example, `<p>` A paragraph. `</p>`.
XML lets you develop and use tags that distinguish different types of data, for example, customer numbers or item numbers.

XML lets you create an application-specific document type, which makes 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.

You can store XML documents in Adaptive Server as:

- XML in a field of a Java object
- XML in a text or image column
- XML in a char or varchar column
- Parsed XML in an image column

**A sample XML document**

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

It might display on screen like this:

**ORDER**

**Date:** July 4, 2003

**Customer ID:** 123

**Customer Name:** Acme Alpha

**Items:**

<table>
<thead>
<tr>
<th>Item ID</th>
<th>Item Name</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

The following is one representation of this data in XML:

```xml
<?xml version="1.0"?>
<Order>
  <Date>2003/07/04</Date>
</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>.

The XML document for the order data consists of:

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

XML represents documents as character data. In each document, you specify the character encoding (character set), either explicitly or implicitly. 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, the default, UTF8, is used.

**Note** When the default character sets of the client and server differ, Adaptive Server bypasses normal character-set translations so that the declared character set continues to match the actual character set. See “Character sets and XML data” on page 9.

- 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 document. 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
<?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>
  <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 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 8).

**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:

<table>
<thead>
<tr>
<th>Item ID</th>
<th>Item Name</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

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
<html>
<body>
<p>ORDER</p>
<p>Date: &nbsp;&nbsp;July 4, 1999</p>
<p>Customer ID: &nbsp;&nbsp;123</p>
<p>Customer Name: &nbsp;&nbsp;Acme Alpha</p>
<p>Items:</p>
<table bgcolor=white align=left border="3" cellspacing=3>
<tr><td><b>Item ID</b></td><td><b>Item Name</b></td><td><b>Quantity</b></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 are the indications for type style (\textit{....}), color (\texttt{bcolor=white}), and layout (\texttt{<table>....</table>}, and also 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 (“\texttt{<p>”), have optional closing tags.
  - Some elements, such as paragraph tags (“\texttt{<p>”) are used for many sorts of data, so it is difficult to distinguish between a “123” that is a Customer ID and a “123” that is an Item ID, without specialized inference 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:

```xml
<!ELEMENT Order (Date, CustomerId, CustomerName, Item+)>  
<!ELEMENT Date (#PCDATA)>  
<!ELEMENT CustomerId (#PCDATA)>  
<!ELEMENT CustomerName (#PCDATA)>  
<!ELEMENT Item (Itemld, ItemName, Quantity)>  
<!ELEMENT Itemld (#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:

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

• Here’s how an embedded DTD might look:

```xml
<?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.

### Character sets and XML data

If the declared character sets of your client and server differ, you must be careful when declaring the character set of your XML documents.

Every XML document has a character encoding that is either specified in the encoding declaration of the XML declaration or is UTF-8 by default.

• If you store an XML document in a character column that is not TEXT, Adaptive Server translates the document into the server’s character set before storing it. This is the way Adaptive Server normally translates character data, and you must ensure that the declared character set of the XML document matches that of the server.
**XML in the database**

- If you store an XML document in a text column, Adaptive Server recognizes the XML document from the XML declaration and does not translate the character set to that of the server. When you read such an XML document from the database, Adaptive Server does not translate the character set of the data to that of the client, since doing so might compromise the integrity of the XML document.

- If you store an XML document in an image column, Adaptive Server performs no conversions. This is the way Adaptive Server normally processes image data.
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. The functions are as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmlextract</code></td>
<td>A built-in function that applies an XML query expression to an XML document and returns the selected result.</td>
</tr>
<tr>
<td><code>xmltest</code></td>
<td>A SQL predicate that applies an XML query expression to an XML document and returns the boolean result.</td>
</tr>
<tr>
<td><code>xmlparse</code></td>
<td>A built-in function that parses an XML document for more efficient processing.</td>
</tr>
<tr>
<td><code>xmlrepresentation</code></td>
<td>A built-in function that determines whether a given image column contains a parsed XML document.</td>
</tr>
</tbody>
</table>
Example sections

The descriptions of these functions include examples that reference Appendix B, “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 
| return_type
| options_parameter return_type
options_parameter ::= [] option option_string
returns_type ::= [] returns [varchar(integer) | text | image] 
option_string ::= [] basic_string_expression

Description

• A basic_string_expression is a sql_query_expression whose datatype is character, varchar, or java.lang.String.

• A general_string_expression is a sql_query_expression whose datatype is text, image, character, varchar, or java.lang.String.

• An xmlextract expression can be used in SQL language wherever a character expression is allowed.

• The default options_parameter is an empty string. A null options parameter is treated as an empty string.

• If the xml_query_expression of an xmlextract call is null, then the result of the xmlextract call is null.

• The value of the xml_data_expression parameter is the runtime context for execution of the XML query expression.

• The datatype of an xmlextract call is specified by the returns_type.

• The default returns_type is text.
If a varchar specified as the returns_type without an integer, the default is 30.

If numeric or decimal is specified without a precision (the first integer), the default is 18. If it is specified without a scale (the second integer), the default is 0.

The initial result of an xmlextract call 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 varcharimage, text, or image, the initial result value is returned as a character-string document of that datatype.

If the returns_type specifies a numeric, money, or date-time 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 URL 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 27.

The option supported for the xmlextract function is:

\[
\text{xmlerror} = \{ \text{exception} | \text{null} | \text{message} \}
\]

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 that xmlerror=exception, an exception is raised:

invalid XML data

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.

If the return_type of the xmlextract_expression is varchar and the runtime result of evaluating the xml_query_expression parameter is longer than the maximum length of a varchar, then an exception is raised:

result exceeded maximum varchar length

Examples

The following examples use the sample_docs table described in Appendix B, “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_docs
------------------------------------
NULL
<pub_id>0736</pub_id>
<pub_id>0736</pub_id>
<pub_id>0736</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:

```sql
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:

```sql
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:

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

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

Adding the `text()` reference yields the following result:

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

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

```sql
select xmlextract
    ('/bookstore/book/price/text()',
     text_doc returns integer)
from sample_docs
where name_doc='bookstore'
```

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

```sql
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:

  ```sql
  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:

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

  ```sql
  (0 rows affected)
  ```

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

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

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

The xmlerror option is used by xmlextract only when the xml_data_expression is a varchar or text expression, or an image expression whose value is an unparsed XML document. In other words, 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:
In the following `xmlextract` call, the `xml_data_expression` is returned by an explicit call by the `xmlparse` function (see section “xmlparse” on page 22). 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:

```sql
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>
```

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:

```sql
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:

```sql
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

- A `basic_string_expression` is a `sql_query_expression` whose datatype is character, varchar, or `java.lang.String`.
- A `general_string_expression` is a `sql_query_expression` whose datatype is character, varchar, 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 an `xmltest` call is null, then the result of the `xmltest` call is unknown.
- The value of the `xml_data_expression` parameter is the runtime context for execution of the XPath expression.
- An `xmltest` call evaluates to boolean `true` or `false`, as follows:
  
  - The `xml_query_expression` of an `xmltest` call is an XPath expression whose result is `empty` (`not empty`), then the `xmltest` call returns `false` (`true`).
  - If the `xml_query_expression` of an `xmltest` call is an XPath expression whose result is a Boolean `false` (`true`), then the `xmltest` call returns `false` (`true`).
  - Otherwise, an exception is raised:
invalid xml expression for xmltest

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

- Restrictions on external URL references, XML namespaces, and XML schemas.
- Treatment of predefined entities and their corresponding characters: \&amp; (&), \&lt; (<), \&gt; (>), \&quot; (“), 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 27.

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:

  invalid XML data

- If the explicit or default options specifies `xmlerror=null` a null value is returned.

### Examples

These examples use the `sample.docs` table described in Appendix B, “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”.

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

---

20 Adaptive Server Enterprise
In the following example the `xmltest` predicate returns `false`/`true`, for a Boolean `false`/`true` result and for an `empty`/`not-empty` result.

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

```sql
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.

  ```sql
  select name_doc from sample_docs
  ```
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])

options_parameter ::= [] option option_string

option_string ::= basic_string_expression

Description

• A basic_string_expression is a sql_query_expression whose datatype is character, varchar, or java.lang.String.

• A general_string_expression is a sql_query_expression whose datatype is character, varchar, or java.lang.String.

• If any parameter of an xmlparse call 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.
An `xmlparse` call parses the `general_string_expression` as an XML document and returns an `image` value containing the parsed document.

**Note** See Chapter 3, “XML Language and XML Query Language,” for the following topics:
- Restrictions on external URL 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**
- The general format of the `option_string` is described in “option_strings: general format” on page 27. The options supported for the `xmlparse` function are:
  - `dtvalidate = {yes | no}`
  - `xmlerror = {exception | null | message}`

  If `dtvalidate=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 `dtvalidate=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, then:
- If the explicit or default options specifies `xmlerror=exception`, an exception is raised:
  - **invalid XML data**
- If the explicit or default options specifies `xmlerror=null`, then 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 \textit{text_doc} column of the \textit{sample_docs} table contains documents, and the \textit{image_doc} column is null. You can update the \textit{image_doc} columns to contain parsed XML versions of the \textit{text_doc} columns:

\begin{verbatim}
update sample_docs
set image_doc = xmlparse(text_doc)
\end{verbatim}

(3 rows affected)

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

\begin{verbatim}
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
\end{verbatim}

\begin{tabular}{lll}
\textit{name_doc} & \textit{extract_from_text_doc} & \textit{extract_from_image_doc} \\
\hline
bookstore & \texttt{<price>55</price>} & \texttt{<price>55</price>} \\
publishers & NULL & NULL \\
titles & NULL & NULL \\
\hline
\end{tabular}

(3 rows affected)

To illustrate the \textit{xmlerror} options, this command inserts an invalid document into the \textit{sample_docs} table:

\begin{verbatim}
insert into sample_docs (name_doc, text_doc) 
values ('invalid doc', '<a>unclosed element</a>)
\end{verbatim}

(1 row affected)

In this example, the \textit{xmlerror} options determine the treatment of invalid XML documents by the \textit{xmlparse} function:

- If \textit{xmlerror}=exception (the default), an exception is raised:

\begin{verbatim}
update sample_docs
set image_doc = xmlparse(text_doc option 'xmlerror=exception')
\end{verbatim}

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:

```sql
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:

```sql
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:

```sql
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(image_expression)
```

**Description**

- An image_expression is an sql_query_expression whose datatype is image.
- If the parameter of an xmlrepresentation call is null, the result of the call is null.
xmlrepresentation

• `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 B, “The sample_docs Example Table”.

This example illustrates the basic `xmlrepresentation` function.

```sql
-- 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
```

Columns of datatype `image` can contain both parsed XML documents (generated by the `xmlparse` function) and unparsed XML documents. After the update commands in the following 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).

```sql
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'
```

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

```sql
select name_doc, xmlrepresentation(image_doc) from
```
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.
```

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)

The following command restores the `sample_docs` table to its original state.

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

**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.
option_strings: general format

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 option_string parameter is:

      options_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, it is ignored.
- 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.
- The value of an option 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 2-2. See specific function descriptions for descriptions of options.
### Table 2-2: Option string values

<table>
<thead>
<tr>
<th>Option name</th>
<th>Option value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>hex</td>
<td>base64</td>
</tr>
<tr>
<td>columnstyle</td>
<td>element</td>
<td>attribute</td>
</tr>
<tr>
<td>didvalidate</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>format</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>header</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>incremental</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>nullstyle</td>
<td>attribute</td>
<td>omit</td>
</tr>
<tr>
<td>nullclause</td>
<td>null</td>
<td>empty</td>
</tr>
<tr>
<td></td>
<td></td>
<td>forsql(scriptj)</td>
</tr>
<tr>
<td>prefix</td>
<td>SQL name (C)</td>
<td>forxml and for xml clause</td>
</tr>
<tr>
<td>root</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>rowname</td>
<td>SQL name (row)</td>
<td>forxml and for xml clause</td>
</tr>
<tr>
<td>schematoc</td>
<td>quoted string with a URL</td>
<td>forxml and for xml clause</td>
</tr>
<tr>
<td>statement</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>tablename</td>
<td>SQL name (resultset)</td>
<td>forxml and for xml clause</td>
</tr>
<tr>
<td>targets</td>
<td>quoted string with a URL</td>
<td>forxml and for xml clause</td>
</tr>
<tr>
<td>xmlerror</td>
<td>exception</td>
<td>null</td>
</tr>
</tbody>
</table>

**Note** The defaults of options that specify keywords are underlined. The defaults of options that specify SQL names are parenthesized. The defaults of options that specify string values are the empty string, or a single-space character.
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 in Adaptive Server” supports.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI support</td>
<td>32</td>
</tr>
<tr>
<td>Namespace support</td>
<td>32</td>
</tr>
<tr>
<td>XML schema support</td>
<td>32</td>
</tr>
<tr>
<td>Predefined entities in XML language</td>
<td>33</td>
</tr>
<tr>
<td>Predefined entities in XML query language</td>
<td>34</td>
</tr>
<tr>
<td>White space</td>
<td>35</td>
</tr>
<tr>
<td>Empty elements</td>
<td>35</td>
</tr>
<tr>
<td>XML Query Language</td>
<td>36</td>
</tr>
</tbody>
</table>

### Character set support

The native XML processor supports only ASCII data for XML documents. It does not support non-ASCII characters such as Japanese Shift_JIS, accented characters, etc.

The encoding declaration of the XML declaration of an XML document may specify encodings of UTF-8, UTF-16, ISO-10646-UCS-4, ISO-8859-1, or ASCII or may default to UTF-8.

Regardless the encoding that is specified or defaulted, XML documents must contain only ASCII characters.
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 URLs. 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

You can parse and store XML documents with XML schema declarations, subject to the restrictions on external references described in “URI support” on page 32.

You can also query XML documents with XML schema declarations, using xmlextract and xmltest, with the restriction that the XML schema are ignored. All elements are treated as character data, and no schema validation is performed.
Predefined entities in XML language

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.

```sql
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:

```sql
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:

```sql
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 &apos;, as the following example shows:

```sql
select xmlextract("/", " "yes" and 'no' ")
-------------------------------------
"yes" and 'no'
```

You can also use '>' in attributes or elements, and it is replaced by the predefined entity &gt;, as this example demonstrates:

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

---

### Predefined entities in XML query language

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 &sn;&lt;', represented by their predefined entities, &amp;&lt; &amp;gt;

- The XML query specifies a character literal with those same XML special characters, also represented by their predefined entities.

```sql
select xmlextract('/a="&amp;&lt;&amp;quot;"', "&amp;&lt;&amp;quot;")
-----------------------------
```
```xml
<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.

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

White space

All whitespace is preserved, and is significant in queries.

```sql
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>
```

```sql
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&gt;</a>".

```sql
select xmlextract("/", "<doc><a/> <b/></b></doc>")
-----------------------------
<doc>
  <a/>
  <b/></b></doc>
```
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_exp::= unary_expr | unary_expr TOKEN_TO unary_expr
unary_expr::= TOKEN_MINUS 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
forward_step::= abbreviated_forward_step
abbreviated_forward_step::= name_test
| TOKEN_ATRARE 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
numeric_literal::= TOKEN_INT | TOKEN_FLOATVAL |
| TOKEN_MINUS TOKEN_INT
string_literal::= TOKEN_STRING
```
The following tokens are supported by the XML Services subset of XPath:

```plaintext
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 and functions**

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Path (Children): the child operator (\ '/') selects from immediate children of the left-side collection.</td>
</tr>
<tr>
<td>//</td>
<td>Descendants: the descendant operator (\ '//' ) selects from arbitrary descendants of the left-side collection.</td>
</tr>
<tr>
<td>*</td>
<td>Collecting element children: an element can be referenced without using its name by substituting the ‘*’ collection.</td>
</tr>
<tr>
<td>@</td>
<td>Attribute: attribute names are preceded by the ‘@’ symbol.</td>
</tr>
<tr>
<td>[]</td>
<td>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 \texttt{where} clause with any semantics. The filter contains a query within it, called the subquery. 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.</td>
</tr>
<tr>
<td>[n]</td>
<td>Index: index is mainly used to find a specific node within a set of nodes. Enclose the index within square brackets. The first node is index 1.</td>
</tr>
<tr>
<td>[-n]</td>
<td>Backtrack index: return the element that is n-1 units from the last element. -1 means the last element, -2 is the next to last element.</td>
</tr>
<tr>
<td>[m to n]</td>
<td>Subscript: returns elements m through n, where m is the first index and n is the last index.</td>
</tr>
<tr>
<td>text()</td>
<td>Selects the text nodes of the current context node.</td>
</tr>
</tbody>
</table>

### XPath set operators

Table 3-2, “XPath set operators,” on page 40, shows the supported XPath set operators.
Table 3-2: XPath set operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>intersect</td>
<td>Intersection: intersect operator returns the set of elements in common between two sets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group: you can use parentheses to group collection operators.</td>
</tr>
<tr>
<td>. (dot)</td>
<td>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.</td>
</tr>
</tbody>
</table>

Boolean Operators => Boolean expressions can be used within subqueries.

and

or

Boolean “and”.

Boolean “or”.

XPath comparison operators

Table 3-3 shows the supported XPath comparison operators.

Table 3-3: XPath comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>equality</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>non-equality</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>less than equal</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>greater than equal</td>
</tr>
</tbody>
</table>
CHAPTER 4

XML Mapping Functions

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>for xml clause</td>
<td>41</td>
</tr>
<tr>
<td>forxmlj, forxmllddj, forxmlschemaj, forxmlallj</td>
<td>44</td>
</tr>
<tr>
<td>forsqlcreatej, forsqlinsertj, forsqlscriptj</td>
<td>49</td>
</tr>
</tbody>
</table>

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][option option_string]
option_string ::= basic_character_expression
```

**Note** For more information about option strings, see “option_strings: general format” on page 27.

Description

- for_xml_clause is a new clause in SQL select statements. The syntax shown above for select includes all of the clauses, including for_xml_clause.
for xml clause

- for xml clause does not support the following datatypes: image, text, binary, varbinary, timestamp, unichar, univarchar, java.lang.String(Abstract types)


- 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.

- 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 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 column datatype is text. The value of that text 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
  ```
CHAPTER 4 XML Mapping Functions

XML in Adaptive Server Enterprise

```sql
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"
```

**Options**

The general format of the `option_string` is specified in “option_strings: general format” on page 27. The options for the `for_xml` clause are specified in “SQLX Options.”

**Exceptions**

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

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

**Example**

The `for_xml` clause:

```sql
select pub_id, pub_name
from pubs2.dbo.publishers
for xml
```

```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>
```
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 E, “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.

When xmlextract and xmltest process the output SQLX-XML documents that forxml functions generate, they provide an XML view of the SQL data that is represented by the SQLX-XML documents.

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 as text. 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 ::= basic_string_expression  
option_string ::= basic_string_expression
```

Description

- A basic_string_expression is a sql_query_expression whose datatype is character, varchar, 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.

• forxmlxmlj 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 forxmlxmlj 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 27. 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", ",")
go

<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>
```
The `forxmltijd` function:

```sql
set stringsize 16384
select forxmltijd
  ("select pub_id, pub_name
    from pubs2.dbo.publishers",
   "tablename=extract null=omit")
go
```

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

The `forxmlschemaj` function:

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

```xml
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <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:sequence>
</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
go

<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"
<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" nillable="true" />
<xsd:element name="pub_name" type="VARCHAR_40" nillable="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:element name="extract" type="TableType.extract">
</xsd:schema>

<!ELEMENT extract (row*)>
<!ELEMENT row (pub_id, pub_name)>
<!ELEMENT pub_id (#PCDATA)>
<!ELEMENT pub_name (#PCDATA)>
CHAPTER 4  XML Mapping Functions

**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. For instructions, see Appendix E, “The Java-Based XQL Processor”.

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 forsqlschema 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 forsqlinsertj 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 forsqlscriptj 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**

```plaintext
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 character, varchar, or java.lang.String.
- If any parameter of forsqlcreatej, forsqlschema, 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 27.

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

\[
\text{xmlerror} = \{ \text{exception | null | message} \}
\]

**Exceptions**

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

• If the explicit or default options specify:

\[
\text{xmlerror} = \text{exception}
\]

an exception is raised:

\[
\text{invalid XML data}
\]

• If the explicit or default options specify:

\[
\text{xmlerror} = \text{null}
\]

a null value is returned.

• If the explicit or default options specify:

\[
\text{xmlerror} = \text{message}
\]
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:

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

The `forsqlinsertj` function:

```sql
set stringsize 16384
declare @rs varchar(16384)
select @rs = forxmlj(
    "select pub_id, pub_name from pubs2.dbo.publishers")
select forsqlinsertj(@rs, "")
go
--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:

```sql
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")
go
(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)
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.

### SQLX options

**Note** In Table 5-1, underlined words specify the default value.
Table 5-1: Options for SQLX mappings

<table>
<thead>
<tr>
<th>Option name</th>
<th>Option value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>hex</td>
<td>base64</td>
</tr>
<tr>
<td>columnstyle</td>
<td>element</td>
<td>attribute</td>
</tr>
<tr>
<td>format</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>header</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>incremental</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>nullstyle</td>
<td>attribute</td>
<td>omit</td>
</tr>
<tr>
<td>prefix</td>
<td>SQL name</td>
<td>Base for generated names</td>
</tr>
<tr>
<td>root</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>rowname</td>
<td>SQL name</td>
<td>Name of the row element</td>
</tr>
<tr>
<td>schemaloc</td>
<td>quoted string with a URL</td>
<td>schemalocation value</td>
</tr>
<tr>
<td>statement</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>tablename</td>
<td>SQL name</td>
<td>Name of the root query</td>
</tr>
<tr>
<td>targets</td>
<td>quoted string with a URL</td>
<td>targetnamespace value (if any)</td>
</tr>
</tbody>
</table>

**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
```
<row>
  <C1>012131415161718191A1B1C1D1E1F1</C1>
</row>
</resultset>

This example shows `binary=base64`:

```sql
select forxmlj("select 0x012131415161718191A1B1C1D1E1F1",
  "binary=base64")
```

```xml
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>ASExQVFhcYGRobHB0eHx</C1>
  </row>
</resultset>
```

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

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

```sql
select pub_id, pub_name from pubs2..publishers
for xml option "columnstyle=element"
```

```xml
<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`:
**SQLX option definitions**

```
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>
```

**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"
-------------------------------
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
<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)**  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" standalone="yes" encoding="UTF-8"?>
```

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 example:

```
select 1,2 for xml option "header=yes"
----------------------------------
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
```
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.

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

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

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

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

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:nil=true attribute.

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

```sql
select 11, null union select null, 22
for xml option "nullstyle=omit"
```

-----------------------
This example shows *nullstyle=attribute*:

```xml
<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.

```xml
<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 65.
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=url  This option specifies a URL 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 URL 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:

```sql
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:

```sql
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.

```sql
select name_doc from sample_doc
where name_doc like "book%"
for xml option "statement=yes"
--------------------------------------------------
CHAPTER 5 XML Mappings

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 65.

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=url

This option specifies a URL 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">
```
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:

```sql
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
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>title_id</th>
<th>advance-t1.advance</th>
<th>price*t1.total_sales</th>
<th>price*t2.total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU2075</td>
<td>MC3021</td>
<td>4,875.00</td>
<td>55,978.78</td>
<td>66,515.54</td>
</tr>
<tr>
<td>MC2222</td>
<td>BU1032</td>
<td>5,000.00</td>
<td>40,619.68</td>
<td>81,859.05</td>
</tr>
<tr>
<td>MC2222</td>
<td>BU7832</td>
<td>5,000.00</td>
<td>40,619.68</td>
<td>81,859.05</td>
</tr>
</tbody>
</table>

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):

```sql
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
```
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:

```sql
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>
        <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>
```
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_':

```sql
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_"
```

```xml
<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>
  </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:

\texttt{U+nnnn}

is replaced with a string of characters of the form:

\texttt{_xnnnn_}

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

_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:nil="true">"
----------------------
```

```
a + b < c & d <a xsi:nil="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:nil="true">"
for xml
------------------------------------------------------
<xresultset 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_>
    </row>
</xresultset>
```

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”.

66

Adaptive Server Enterprise
For example:

```sql
select 1 as x, 2 as X, 3 as X99, 4 as XML, 5 as XmlDoc
for xml
------------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <x>1</x>
    <X>2</X>
    <X99>3</X99>
    <XML>4</XML>
    <XmlDoc>5</XmlDoc>
  </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:

```sql
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">
  <_row_x0020__x0026__x0020_columns_x0020_>
    <C_x0020_1>11</C_x0020_1>
    <C_x0020_2>12</C_x0020_2>
  </_row_x0020__x0026__x0020_columns_x0020_>
  <_row_x0020__x0026__x0020_columns_x0020_>
    <C_x0020_1>21</C_x0020_1>
    <C_x0020_2>22</C_x0020_2>
  </_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.
**SQLX data mapping**

**Numeric values**

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

```sql
select 1, 2.345, 67e8 for xml
```

```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;` ', `<`, `>`, and `&`.  

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.

```sql
select '<name>"Baker''s"</name>'
go
```

```xml
<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 `forxml` function doubles the embedded quotes.

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

```xml
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>&lt;name&gt;&quot;Baker&apos;s&quot;&lt;/name&gt;
  </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
\texttt{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 53.

SQLX schema mapping

The \texttt{forxmlschemaj} function and the \texttt{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 \texttt{varchar(3)}, \texttt{numeric(3,1)}, \texttt{varbinary(2)}, \texttt{numeric(3,1)}, and \texttt{numeric(3,2)}.

\begin{verbatim}
select 'abc', 12.3, 0x00, 45.6, 7.89
--- ------ ---- ------ ------
abc   12.3 0x00   45.6   7.89
\end{verbatim}

The SQLX-XML result set for this data is:

\begin{verbatim}
select forxmlj("select ‘abc’, 12.3, 0x00, 45.6, 7.89", ")
------------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001
/XMLOSchema-instance">
  <row>
    <C1>abc</C1>
  </row>
</resultset>
\end{verbatim}
The SQLX-XML schema describing this document is:

```xml
select forxmlschemaj("select 'abc', 12.3, 0x00, 45.6, 7.89", ""
------------------------------------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmns: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:schema>
```
This XML schema has five components:

- In the last part of the 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` declaration 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:

```sql
select forxmlschemaj("select 1,2", "columnstyle=element")
```

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <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="xsd:integer" />
      <xsd:element name="C2" type="xsd: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="xsd: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:

```xml
select forxmlschemaj("select 1,2", "columnstyle=attribute")
```

```xml
  <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:

```xml
select forxmlschemaj("select 1,2", "nullstyle=omit")
```

```xml
  <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>
```
SQLX schema mapping

```xml
select forxmlschemaj("select 1,null", "nullstyle=omit")

-----------------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmllns: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
`nillable="true"` attribute in each nullable columns declaration. For example:

```xml
select forxmlschemaj("select 1,null", "nullstyle=attribute")

-----------------------
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmllns: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" nillable="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>
SQLX schema mapping
Enabling the native XML processor

To use XML Services, you must enable it using this `sp_configure` command:

```
sp_configure "enable xml", 1
```

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`

---

Enabling the native XML processor

To use XML Services, you must enable it using this `sp_configure` command:

```
sp_configure "enable xml", 1
```

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`
Installing the Java-based SQLX mapping functions

- 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 A-1 already exist in the server utilities.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ISERVER</td>
<td>&quot;-S&quot; parameter for isql and installjava utilities</td>
</tr>
<tr>
<td>$INTERFACES</td>
<td>&quot;-I&quot; parameter for isql and installjava utilities</td>
</tr>
<tr>
<td>$DB</td>
<td>&quot;-D&quot; parameter for isql and installjava utilities</td>
</tr>
</tbody>
</table>

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, forxmlldtj, forxmlschema, 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
  (queryparm 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 forxmlldtdj
  (queryparm java.lang.String, optionparm java.lang.String)
returns java.lang.String
Installing the Java-based SQLX mapping functions

```java
language java parameter style java
external name "jcs.sqlx.ForXml.forXmlDTD"

create function forsqlcreatej
    (schema 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
    (schema 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 B

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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_docs table columns and rows</td>
<td>81</td>
</tr>
<tr>
<td>sample_docs tables</td>
<td>84</td>
</tr>
<tr>
<td>Publishers table representation</td>
<td>84</td>
</tr>
<tr>
<td>Titles table representation</td>
<td>85</td>
</tr>
</tbody>
</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:

```sql
create table sample_docs
```
sample_docs table columns and rows

(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:

```sql
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 Mary</publication>
  </author>
</book>
</bookstore>
```

Adaptive Server Enterprise
<book>55</book>
<?PI_sample Process Instruction ?>
<!--sample comment--> 
<magazine style='glossy' frequency='monthly'>
<title>Tracking Trenton</title>
</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>
</book>
<br>It was a dark and stormy night.</br>
<br>But then all nights in Trenton seem dark and stormy to someone who has gone through what<br>/emph/I have.</br>
<definition-list>
  <term>Trenton</term>
  <definition>misery</definition>
</definition-list>
</book>
<br>Who’s Who in Trenton</br>
</book>
</bookstore>
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 44.

**Table script (for publishers table)**

```sql
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 A, “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 84.

```sql
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
-------------------------------------------
<table>
<thead>
<tr>
<th>title_id</th>
<th>title</th>
<th>type</th>
<th>pub_id</th>
<th>price</th>
<th>advance</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>The Busy Executive’s Data Base Guide</td>
<td>business</td>
<td>1389</td>
<td>19.99</td>
<td>5000.00</td>
<td>4095</td>
</tr>
<tr>
<td>BU1111</td>
<td>Cooking with Computers: Surreptitious Balance Sheets</td>
<td>business</td>
<td>1389</td>
<td>11.95</td>
<td>5000.00</td>
<td>3876</td>
</tr>
<tr>
<td>BU2075</td>
<td>You Can Combat Computer Stress!</td>
<td>business</td>
<td>0736</td>
<td>2.99</td>
<td>10125.00</td>
<td>18722</td>
</tr>
<tr>
<td>BU7832</td>
<td>Straight Talk About Computers</td>
<td>business</td>
<td>1389</td>
<td>19.99</td>
<td>5000.00</td>
<td>4095</td>
</tr>
<tr>
<td>title_id</td>
<td>title</td>
<td>type</td>
<td>pub_id</td>
<td>price</td>
<td>advance</td>
<td>total_sales</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------</td>
<td>---------------</td>
<td>--------</td>
<td>--------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>MC2222</td>
<td>Silicon Valley Gastronomic Treats</td>
<td>mod_cook</td>
<td>0877</td>
<td>19.99</td>
<td>0</td>
<td>2032</td>
</tr>
<tr>
<td>MC3021</td>
<td>The Gourmet Microwave</td>
<td>mod_cook</td>
<td>0877</td>
<td>2.99</td>
<td>15000.00</td>
<td>22246</td>
</tr>
<tr>
<td>MC3026</td>
<td>The Psychology of Computer Cooking</td>
<td>UNDECIDED</td>
<td>0877</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC1035</td>
<td>But Is IT User Friendly?</td>
<td>popular_comp</td>
<td>1389</td>
<td>22.99</td>
<td>7000.00</td>
<td>8780</td>
</tr>
<tr>
<td>PC8888</td>
<td>Secrets of Silicon Valley</td>
<td>popular_comp</td>
<td>1389</td>
<td>20.00</td>
<td>8000.00</td>
<td>4095</td>
</tr>
<tr>
<td>Title ID</td>
<td>Title</td>
<td>Type</td>
<td>Publisher ID</td>
<td>Price</td>
<td>Advance</td>
<td>Total Sales</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------------</td>
<td>----------------</td>
<td>--------------</td>
<td>----------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>PC9999</td>
<td>Net Etiquette</td>
<td>popular_comp</td>
<td>1389</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PS1372</td>
<td>Computer Phobic and Non-Phobic Individuals: Behavior Variations</td>
<td>psychology</td>
<td>0877</td>
<td>21.59</td>
<td>7000.00</td>
<td>375</td>
</tr>
<tr>
<td>PS2091</td>
<td>Is Anger the Enemy?</td>
<td>psychology</td>
<td>0736</td>
<td>10.95</td>
<td>2275.00</td>
<td>2045</td>
</tr>
<tr>
<td>PS2106</td>
<td>Life Without Fear</td>
<td>psychology</td>
<td>0736</td>
<td>7.99</td>
<td>6000.00</td>
<td>111</td>
</tr>
<tr>
<td>PS3333</td>
<td>Prolonged Data Deprivation: Four Case Studies</td>
<td>psychology</td>
<td>0736</td>
<td>19.99</td>
<td>2000.00</td>
<td></td>
</tr>
</tbody>
</table>
<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)
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 using `sp_configure`:
  ```
  sp_configure "enable xml", 1
  ```
- Verify that the static configuration parameter `enable cis` is set to 1:
  ```
  sp_configure "enable cis"
  ```
- Enable file access using `sp_configure`:
sp_configure "enable file access", 1

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.

```sql
create proxy_table xmlxfsTab external directory
at "/remote/nets3/bharat/xmldocs;R"
select filename from xmlxfsTabgo 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

```sql
select filename , xmlextract("//book/title" , content)
from xmlxfsTabgo 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 ("/").
Getting Started

```
select filename ,xmlcol=xmlextract("/",content) into xmldoctab
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.

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 t
----------
(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'

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

---------------------------------------
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
The following code sample shows an XML query on both XML and non-XML data:

```sql
select filename , xmlextract("//book/title",content)
from xmlxfsTab
--------------
Msg 14702, Level 16, State 0:
Line 1:
XMEXTRACT(): 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.

```sql
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>

nonxmlloc.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:

```sql
select filename, xmlextract("//book/title", content option 'xmlerror = null')
from xmlxfsTabgo 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.

```sql
select filename from xmlxfsTab
where '/' not xmltest content
  option 'xmlerror = null'
```

```
picture.jpg
nonxmldoc.txt
```

(2 rows affected)

Introduction

The Java-based XQL processor and the native XML processor are similar but different. They 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 and 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:

```sql
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:

```sql
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:

```sql
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:

```sql
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:

```sql
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.xml.xql.Xql.parse`, you can regenerate the text form of the document in the new `xmlsource` column with the following SQL statement:

```sql
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.aseutils.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
Migrating documents and queries

- 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:

  ```sql
  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:

  ```sql
  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 "<xml_result>...</xml_result>" tags, and the native XML processor does not.
APPENDIX D

Migrating Between the Java-based XQL Processor and the Native XML Processor

Introduction

The Java-based XQL processor and the native XML processor are similar but different. They 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 xmltext.

- The parsed documents generated by the xmlparse built-in function can only be processed by the xmlextract and xmltext 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.

Character set support

The native XML processor supports only ASCII data for XML documents. The Java-based XQL processor supports additional character sets, including Unicode. Documents with such data cannot be processed by 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.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.xml.xql.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 xmltext 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:

```sql
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.aseutils.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:

```sql
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:

```sql
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:

```sql
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.
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

Whether you install the Java-based XQL processor as a standalone program or inside Adaptive Server, you must first access the XML parser. Sybase recommends the *xerces.jar (vs. 1.3.1)* parser, available at
Setting up the Java-based XQL processor

- $SYBASE/ASE-12_5/lib/xerces.jar (UNIX)
- %SYBASE%\ASE-12_5\lib\xerces.jar (Windows NT)

You can download the parser from:
You can also use any parser that is compliant with SAX 2.0.

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/ASE-12_5/lib/xerces.jar
$SYBASE/ASE-12_5/lib/xml.zip
```
For Windows NT, enter:
```
set CLASSPATH = D:\%SYBASE%\ASE-12_5\lib\xerces.jar
D:\%SYBASE%\ASE-12_5\lib\xml.zip
```

Installing the Java-based XQL processor in Adaptive Server

This section assumes you have already enabled Java in Adaptive Server.
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/ASE-12_5/lib/xml.zip and $SYBASE/ASE-12_5/lib/xerces.jar

For example, to install xml.zip, enter:

```
installjava -Usa -P -S server_name -f $SYBASE/ASE-12_5/lib/xml.zip
```

To install xerces.jar, enter:

```
installjava -Usa -P -S server_name -f $SYBASE/ASE-12_5/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 7-1. For more information on configuration parameters, see the Sybase Adaptive Server System Administration Guide.

**Table 5-2: Java Services memory parameters**

<table>
<thead>
<tr>
<th>Section</th>
<th>Reset value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable java</td>
<td>1</td>
</tr>
<tr>
<td>size of process object heap</td>
<td>5000</td>
</tr>
<tr>
<td>size of shared class heap</td>
<td>5000</td>
</tr>
<tr>
<td>size of global fixed heap</td>
<td>5000</td>
</tr>
</tbody>
</table>
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:

```sql
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:

```sql
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:

```sql
update XMLDAT
set 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:

  ```
  from xmlimage
  
  <xql_result>
  <book style=autobiography>
  <title>
  ```

- **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:

  ```
  from xmlimage
  
  <xql_result>
  ```

```
Using the Java-based XQL processor

- Equals operator, = – specifies the content of an element or the value of an attribute. The following query finds all examples where “last-name = Bob”:

```sql
select com.sybase.xml.xql.Xql.query
    ("/bookstore/book/author[last-name='Bob']", xmlcol)
from xmlimage
```

```
<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>
```

- 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:

```sql
select com.sybase.xml.xql.Xql.query
    ("/bookstore/book[author/first-name = 'Mary']", xmlcol)
from xmlimage
```

```
<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>
```
• 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:

```java
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>
    </author>
    <price>55</price>
  </book>
</xql_result>
```

• Boolean expressions – you can use Boolean expressions within filter operators. For example, this query returns all <author> elements that contain at least one <degree> and one <award>:

```java
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:

```sql
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>
        </author>
        Selected Short Stories of
        <first-name>Mary</first-name>
        <last-name>Bob</last-name>
        <publication>
        </publication>
        <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 12.5, 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 121.

**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 Mary</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 `URL()` method to read a Web page named `bookstore.xml` which contains information about all the books in the bookstore:

```java
String result;
FileInputStream XmlFile = new FileInputStream("bookstore.xml");
if ((result =
Other usages of the Java-based XQL processor

```java
Xql.query("/bookstore/book/author/first-name", XmlFile))
!= Xql.EmptyResult )
{
    System.out.println(result);
}else{
    System.out.println("Query returned false\n");
}
URL _url = new URL("http://mybookstore/bookstore.xml");
if ((result =
    Xql.query("/bookstore/book/author/first-name",url.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 XmlDocument object, then queries the document.

The remote interface looks like:

```java
public interface XmlBean extends javax.ejb.EJBObject
{
    /**
     * XQL Method*/
    public String XQL(String query, URL location)
    throws java.rmi.RemoteException;
}
```

The Bean implementation looks like:

```java
public class XmlBean extends java.lang.Object implements
    javax.ejb.SessionBean
{
    ....
    /**
     * XQL Method
     */
    public String XQL(String query, java.net.URL 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();
URL u = new URL("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.
parse(InputStream xml_document, boolean validate)

Examples

The following example:

```java
SybXmlStream xmlStream = Xql.parse("<xml>..</xml>\);
```

Returns `SybXmlStream`.

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

```java
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

Where:

- `InputStream` is an input stream.
- `xml_document` is the XML document where the input stream originates.

Examples

The following example:

```java
SybXmlStream is = Xql.parse(new FileInputStream("file.xml"), true);
```

Returns `SybXmlStream`.

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:

```java
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.

```java
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 URL as the search argument:

```java
URL xmlURL = new URL("http://mywebsite/doc.xml");
String result = Xql.query("/bookstore/book/author", xmlURL.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
```java
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`.
```java
SybXmlStream xmlStream = Xql.parse("<xml>..</xml>");
String result = Xql.query("/bookstore/book/author",xmlStream);
```

query(String query, JXml jxml)

Description
Queries an XML document stored in a JXML format.

Syntax
```java
query(String query, JXml jxml)
```
Where:
- `String query` is the string you are searching.
- `JXml jxml` is an object created from the classes located in `$SYBASE/ASE-12.5/samples/`

Examples
This example queries for authors in `bookstore.Xql`.
```java
JXml xDoc = new JXml("<xml>...</xml>");
String result = Xql.query("/bookstore/book/author", xDoc);
```

Usage
Allows you to execute a query on an JXML document using XQL.

sybase.aseutils.SybXmlStream

Description
Defines an interface that an InputStream needs to access parsed XML data while querying.

Syntax
```java
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:

```java
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
(\textit{xerces.jar}, Version. 1.3.1).

Syntax
reSetParser

Examples
This example resets your parser to the Sybase default parser.

\texttt{xql.resetParser()}
Index

Symbols

, (comma)
in SQL statements xiii
() (parentheses)
in SQL statements xiii
[] (square brackets)
in SQL statements xiii
{} (curly braces) in SQL statements xiii

A
Adaptive Server
installing XQL 110
additional information about XML 2
alias names, creating 79
alter table
command 112
aseutils methods, com.sybase.xml.xql.Xql
methods, specific to 124
at clause, command 93
attributes, embedded in element tags 5

B
base64, SQLX option 54
basic operators, XPath, supported 38
binary

datatype 54, 69
option 54
SQLX option 54
values 69
book

authors example 118
DTD conforming 119
filename 118
validate command 119
Web page 119
XML example 118
bookstore.xml, sample document 82
Boolean expressions, within filter operators 115

C
char datatype 68
character encoding. See character sets
character set support
if server and client differ 9
Java-based processor supports additional character sets, including Unicode 104
native XML processor supports only ASCII data 104
character sets
declared matching actual 5
default UTF8 5
specifying 4
translations, bypassed 5
XML 4
character value, example 68
child operator 113
CIS (Component Integration Services) 91
CLASSPATH

environment variable 110
environment variables for UNIX and NT 110
standalone program 110
xerces.jar, xml.zip, runtime.zip 110
code samples
HTML, Order example 6
XML, Info example 5
XML, Item example 6
columnstyle, SQLX option 55, 62
com 125

XML Services in Adaptive Server Enterprise 127
Index

com.sybase.xml.xql.store methods 125
com.sybase.xml.xql.store.SybMemXmlStream, XQL interface 125
com.sybase.xml.xql.Xql
methods, specific to 121, 122, 123, 125, 126
com.sybase.xml.xql.XqlDriver
local files 116
querying XML documents 116
standalone program 116
syntax 117
using 116, 117
comma (,) in SQL statements xiii
command
at clause 93
create proxy table 93
make install-sqlx 79
make install-xerces 78
make sqlx-aliases 79
select 57
commands
alter table 112
debug 117
FileInputStream() 119
help 117
infile 117
new 110
outfile 117
parse 119
parse() 112
qstring 117
query 119
update 110
URL 119
valid 119
validate 117, 119
where clause 116
Component Integration Services (CIS) 91
configuring memory requirements 111
conventions
Java-SQL syntax xii
Transact-SQL syntax xiii
create proxy table
command 93
creating
alias names 79
curly braces (()) in SQL statements xiii
customizing elements 5

data
selecting with XQL 109
database objects
xmlcol 112, 113
xmlimage 113
XMLTEXT 112
XQL, general query language for XML 113
datatypes
binary 54, 69
char 3, 68
image 3, 54, 69, 112
numeric 68
text 3, 68, 112
varbinary 54, 69
varchar 3, 68
DB environmental variable 78
debug command 117
deleting
XML documents 113
descendant operator 113
document
and queries, migrating 99, 104
migrating between Java and native processors 99, 104
validating 119
Document Type Definition. See DTD
DTD 8
#IMPLIED 8
#PCDATA DTD elements 8
asterisk (*) 8
ATTLIST 8
ELEMENT 8
elements of 8
for valid XML document 9
internal 9
not required in all documents 9
plus sign (+) 8
question mark (?) 8
duplicate column names, mapping 62
XML Services in Adaptive Server Enterprise

Index

E

EFS access 91
element tags
  customizing 5
  embedded attributes 5
  HTML 7
  strict nesting 5
  user-created 5
embedding DTD in XML 8
entities
  predefined, in XML language 33
  predefined, in XML query language 34
environmental variable
  $DB 78
  $ISERVER 78
environmental variables
  $INTERFACES 78
  DB 78
  table 78
environmental variables, table 78
equals, operator 114
example
  xmltest 19
example table, sample_docs 81
Extensible Style Language. See XSL
External Directory Recursive Access, maps proxy table 91
External File System access, in XML 91

F

File System Access, feature 91
FileInputStream(), command 119
filename, bookstore.xml 118
filter operator 114
filter operator, using Boolean expressions 115
font conventions
  for Java xi
for xml
  clause 53, 57
  clause, syntax and examples of 41
  description of 41
  extension of select statements 1
  extent of select statements 1
  in data mapping 62
  SQLX-XML format 62
format
  SQLX option 56
formatting
  instructions, provided in XSL 3
  SQLX-XML 53
forsqlcreatej
  description, syntax, examples 49
  function description 49
forsqlinsertj
  description, syntax, examples 49
  functions 49
forsqlscriptj
  description, syntax, examples 49
  function 49
forxml function 57
forxmlallj
  description, syntax, examples 44
  function 44
  installing 77
forxmlcreatej installing 78
forxmldeletej
  description, syntax, examples 44
  function 44
  installing 77
forxmlinsertj
  installing 78
forxmlinsertj installing 78
forxmlj
  function 53
  installing 77
  Java-based function 1, 44
  SQLX-XML format 62
forxmlschema
  description, syntax, examples 44
  installing 77
  Java-based function 44
  function
  XPath 37
function predicate xmltext 19
functions
  forsqlinsertj, Java-based mapping functions 49
  forsqlinsertj 49
  forsqlscriptj 49
  forxml 57
  forxmlallj 44
Index

forxmlldtdj 44
forxmlj 53
forxmlj, description, syntax, and examples 44
forxmlschemas 44
Java-based 44
Java-based SQLX mapping 77
mapping 41
syntax and examples 41
xmlextract 12
xmlparse 22
xmlrepresentation 25

H
header, SQLX option 56
help command 117
hex, SQLX option 54
HTML
display of Order data 6
DTD elements 8
element bracketing inconsistent 7
inconsistent element tags 7
Order code sample 6

I
image datatype 54, 69
image, datatype 112
image_doc, sample_docs table columns 81
inconsistent element tags, HTML 7
incremental, SQLX option 57
infile, command 117
info, XML code example 5
installing
SQLX mapping classes 79
SQLX mapping functions, Java-based 77
the parser 78
XQL in Adaptive Server 110
installjava utility 110
INTERFACES environmental variable 78
environmental variable
SISERVER 78

J
Java commands. See commands
Java mapping functions, installing 79
Java methods, specific to com.sybase.xml.xql.Xql 121
Java processor 1
Java Services
memory requirements table 111
table, memory parameters 111
Java-based
mapping functions 44
parser, installing 78
SQLX mapping functions, installation procedures 78
SQLX mapping functions, list of 77
Java-based mapping functions 44, 49
Java-based processor, migrating to native XML processor 103
Java-based processor, migrating with native XML processor 100, 104
Java-based processor, regenerating text documents 100, 105

L
languages
XML and XML query 31
XPath 1
XQL 1
local files, com.sybase.xml.xql.XqlDriver 116

M
make install-sqlx, command 79
make install-xerces command 78
make sqlx-aliases command 79
mapping
duplicate column names 62
duplicate column names, example 62
SQL names, objectives 65
SQL values 67
SQL values, examples 67
SQLX schema, examples 69
SQLX schema, overview 69
to XML names 65

Adaptive Server Enterprise
Index

XML Services in Adaptive Server Enterprise

O

operators
  child 113
  descendant 113
  equal 114
  filter 114
  subscript 115

option_string values, table of 29

option_strings
  description and examples 27
  discussion 27
  parameter 11

options
  saxparser 117
  SQLX 54
  SQLX, definitions 54

Order DTD, sample code 8

Order sample
  HTML 6
  XML code 3

outfile, command 117

P

parameters
  117
  help 117
  qstring 117
  validate 117

parentheses () in SQL statements xiii

parse command 119

parse methods 121

parse()
  command 112
  returns sybase.aseutils.SybXmlStream 112
  parse(), Java method, command 112
  parse(InputStream xmlDocument), XML method 122
  parse(String xmlDoc), XQL method 121

plus sign (+) in XML document type definition 8

prefix, SQLX option 67

processing
  effect of where clause 116

processor

to XML values 67

unnamed columns 62

mapping functions
  Java-based 44, 49

memory parameters, Java Services, table 111

memory requirements
  configuring 111
  for query engine 111
  Java Services 111

methods. See Java methods

migrating
  documents and queries 99, 104
  documents between Java-based processor and native XML processor 99, 104
  text documents between Java-based and native XML processors 100, 104
  migrating between Java-based and native processors 103
  migrating, documents from regenerated copies 100, 105
  migrating, queries 102, 107

name_doc, sample_docs table columns 81

namespace
  declarations and references 32
  support, XML documents 32

native XML processor
  migrating with Java-based processor 100, 104
  native XML processor, migrating from Java-based processor 103

native XML processor, regenerating text documents 101, 106

navigating XQL 113

new, command 110

nullstyle, SQLX option 57

nullstyle=attribute, option, example 74

nullstyle=omit, option, examples 73

numeric datatypes 68

numeric values 67

N

Index
Index

XML, Java-based processor, 1
XML, native processor, 1
proxy table, created by File System Access feature 91
publishers, pubs2 database table 82

Q
qstring command 117
queries
queries, migrating 99, 104
queries, migrating between Java-based and native processors 102, 107
query command 119
query engine
memory requirements 111
query functions
syntax and examples 12
table of 11
XML 11
query methods, com.sybase.xml.xql.Xql 123
methods, specific to 123, 124
query structures 116
query(String query, InputStream xmlDoc), XQL method 123
query(String query, String xmlDoc), XML method 123
query(String query,JXml jxml), XQL method 124
query(String query,SybXmlStream xmlDoc), XQL method 124
querying XML with com.sybase.xml.xql.XqlDriver 116

R
referencing, XML DTD externally 8
regenerating, text documents, from Java-based processor 100, 105
regenerating, text documents, from native XML processor 101, 106
root, SQLX option 58
rowname, SQLX option 59, 67

S
sample code
DTD, Order example 8
HTML, Order sample 6
XML, Info example 5
XML, Order example DTD 8
sample table, sample_docs 81
sample XML schema 71
sample_docs table
columns 81
publishers and titles tables 84
publishers table, XML representation 84
rows 82
structure 81
table script (publishers) 84
table script, Java 84
titles table, XML representation 85
sample_docs table columns
image_doc 81
name_doc 81
text_doc 81
saxparser, com.sybase.xml.xql.XqlDriver option 117
schema support 32
schemaloc, SQLX option 59
searching, XML documents stored on Web 2
select command 57
selecting data with XQL 109
setParser, XQL method 125
SGML, Standardized General Markup Language 2
sp_configure
enabling XML Services and External File System Access 91, 92
specifying character set 4
SQL extensions, query functions as 11
SQL names, examples 67
SQL statement symbol
() (parentheses) xiii
comma (,) in SQL statements xiii
SQL statement symbols
[ ] (square brackets) xiii
{ } (curly braces) xiii
sql_name, SQLX option 59
SQLX
data mapping 62
options, definitions 54
options, table 54
Index

XML Services in Adaptive Server Enterprise 133

schema mapping 69
schema mapping examples 69
SQLX data 62
SQLX mapping functions, installation procedures 77
SQLX options
  base64 54
  binary 54
  columnstyle 55, 62
  columnstyle=attribute, examples 73
  columnstyle=element, example 72
  format 56
  header 56
  hex 54
  incremental 57
  nullstyle 57
  nullstyle=omit 73
  prefix 67
  root 58
  rowname 59, 67
  schemaloc 59
  sql_name 59
  statement 60
  tablename 61, 67
  tablename=sqlname 61
  targetns 61
  targetns=url 61
SQLX-XML
  format 53
  schema, columnstyle=attribute, examples 73
  schema, columnstyle=element option 72
  schema, nullstyle=attribute, examples 74
  schema, nullstyle=omit, examples 73
  schema, sample 70
SQLX-XML format 62
square brackets [ ] in SQL statements xiii
standalone applications
  example 119
  using XQL 119
standalone program.com.sybase.xml.xql.XqlDriver 116
Standardized General Markup Language (SGML) 2
statement, SQLX option 60
parameters
  infile 117
  outfile 117
structures, query 116

subscription operator 115
Sybase
  product certifications xi
  sybase.asciutils 124
  sybase.aseutils.SybXmlStream, returned by parse() command 112
  SybFileXmlStream, XQL method 125
  SybXmlStream 121
  variable 121
syntax
  examples, xmlextract 12
  for xml clause, description 41
  forsqlcreatej 49
  forsqlinsertj 49
  forsqlscriptj 49
  forxmlallj 44
  forxmltdj 44
  forxmlj 44
  forxmlschemaj 44
  option_strings 27
  xmlparse 22
  xmltest 19
  XPath tokens 36
syntax conventions
  Java-SQL xii
  Transact-SQL  xii, xiii
syntax conventions, for Java xii
syntax, xmlrepresentation 25

T
  table
    environmental variables 78
    option_string values 29
    publishers, XML representation of 84
    SQLX options 54
    titles, XML representation of 85
    XPath basic operators 39
    XPath comparison operators 40
    XPath set operators 39
    tablename, SQLX option 61, 67
    tablename=sqlname, SQLX option 61
tags
  customizing in XML 2
  HTML, inconsistent bracketing 7
Index

HTML, paragraph 7
user-created 5
XML strictly nested 4
targetns, SQLX option 61
targetns=url, SQLX option 61
text data, XML 5
text datatype 68
text() XPath basic operator 39
text, datatype 112
text_doc, sample_docs table columns 81
titles, pubs2 database table 82
tokens XPath, supported 37

U
Unicode data, supported by Java-based XML processor 104
Universal Resource Indicator (URI) 32
unnamed columns, mapping 62
update, command 110
updating XML documents 112
URI (Universal Resource Indicator) 32
URL, command 119
user-created element tags 5
using
    com.sybase.xml.xql.XqlDriver 116
UTF8, default character set 5

V
valid XML document 9
valid, command 119
validate, command 117, 119
validating document 119
values
    binary 69
    character 68
    numeric 67
varbinary datatype 69
varchar datatype 68
variables
    SybXmlStream 121
variables, HTML
        &lt;b&gt;...&lt;/b&gt; 7
        &lt;table&gt;...&lt;/table&gt;, layout 7
        bgcolor, color 7
        CustomerID 6, 7
        CustomerName 7
        Data 7
        ItemID 6
        ItemName 6
        order 6
        Quantities 7
        Quantity 6
        units 6
variables, XML
tag 4

W
Web addresses
    W3C, Document Object Model (DOM) 2
    World Wide Web Consortium (W3C) 2
Web information on XML 2
Web, storage for XML documents 119
well-formed XML document 5
where clause
    affects processing 116
    command 116

X
xerces.jar
directory 110
XML
    additional information 2
    application-specific document types 2
    compared to SGML and HTML 2
    comparison with HTML 2
    custom tags 2
    declaration, to specify character set 4
    DTD elements, restrictions 8
    DTD not required in all documents 9
    DTD sample code, embedding 8
    DTD sample code, referencing externally 8

134

Adaptive Server Enterprise
Index

DTD, instruction 8
mapping functions 41
mappings 53
names, mapping to 65
overview 2
query functions 11
query language 31
read by HTML browsers and processors 2
representation of publishers table 84
representation of titles table 85
sample document 3
schema declarations 32
schema, sample 71
strict phrase structure 2
subset of SGML 2
suitable for data interchange 2
values, mapping to 67
Web documents for detailed information 2
XML document
as character data 4
deleting 113
DTD sample code 8
example, info 5
extracting book title from, sample code 93
generating from Adaptive Server 2
importing to ASE table, sample code 93
namespace support, XML documents 32
nested markup tags 4
no formatting instructions 5
parts of 4
querying 116
querying from subdirectory, sample code 94
sample code, Order 3
searching on Web 2
stored as OS file 119
stored on Web 119
storing in Adaptive Server 2
strict phrase structure 2
tags 2
updating 112
valid with DTD 9
well-formed 5
XML documents
creating proxy table, sample code 92
XML example, bookstore.xml 118
XML methods
parse(InputStream xml_document) 122
query(String query, String xmlDoc) 123
XML query language
subset of XPath 36
XML Services
with External File System Access 91
with External File System Access, using
sp_configure 91, 92
XML Services and external file system access 91
xml.zip, directory 110
xmlcol, database object 112, 113
xmlextract
built-in query function 11
function description 12
namespace prefixes 32
support for XQuery language subset 32
xmlimage, database object 113
xmlparse
built-in function 11
function description 22
syntax and examples 22
xmlrepresentation
built-in function, determines parsed image column 11
function description 25
syntax and examples 25
xmltest
function 19
syntax and description 19
xmltext
function predicate 19
namespace prefixes 32
SQL predicate, returns boolean result 11
XMLTEXT, database object 112
XPath
basic operators, table 39
comparison operators, table 40
language 1
language subset 36
operators and functions 37
set operators, table 39
supported tokens 37
syntax and tokens 36
XQL
developing standalone applications 119
displaying as XML 109
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJBs</td>
</tr>
<tr>
<td>installing in Adaptive Server</td>
</tr>
<tr>
<td>interface,</td>
</tr>
<tr>
<td>com.sybase.xml.xql.store.SybMemXmlStream</td>
</tr>
<tr>
<td>JavaBeans</td>
</tr>
<tr>
<td>JDBC clients</td>
</tr>
<tr>
<td>navigation</td>
</tr>
<tr>
<td>numbering system</td>
</tr>
<tr>
<td>operators</td>
</tr>
<tr>
<td>path-based query language</td>
</tr>
<tr>
<td>zero based</td>
</tr>
<tr>
<td>XQL methods</td>
</tr>
<tr>
<td>parse(String xmlDoc)</td>
</tr>
<tr>
<td>query(String query, InputStream xmlDoc)</td>
</tr>
<tr>
<td>query(String query, JXml jxml)</td>
</tr>
<tr>
<td>query(String query, SybXmlStream xmlDoc)</td>
</tr>
<tr>
<td>setParser</td>
</tr>
<tr>
<td>SybFileXmlStream</td>
</tr>
<tr>
<td>SybXmlStream</td>
</tr>
<tr>
<td>XQuery language, supported by xmlextract and xmltest</td>
</tr>
<tr>
<td>XSL, Extensible Style Language</td>
</tr>
</tbody>
</table>