## Contents

### CHAPTER 1  Introduction to XML Services ........................................................ 1
- XML capabilities .................................................................................. 1
- Overview .......................................................................................... 2
  - XML in the database ................................................................ 2
  - A sample XML document .......................................................... 3
  - XML document types .................................................................. 7

### CHAPTER 2  XML Query Functions ................................................................... 11
- XML query functions ......................................................................... 11
  - Example sections .......................................................................... 12
  - xmlextract ..................................................................................... 12
  - xmltest .......................................................................................... 19
  - xmlparse ......................................................................................... 23
  - xmlrepresentation ......................................................................... 27
  - xmlvalidate ..................................................................................... 29
  - option_strings: general format ................................................. 39
  - Option values for query functions ............................................ 40

### CHAPTER 3  XML Language and XML Query Language ................................. 41
- Character set support .......................................................................... 41
- URI support ....................................................................................... 41
- Namespace support ........................................................................... 42
- XML schema support ......................................................................... 42
- Predefined entities in XML documents ........................................... 42
- Predefined entities in XPath queries ............................................... 44
- White space ...................................................................................... 44
- Empty elements ................................................................................ 45
- XML Query Language ........................................................................ 45
  - XPath-supported syntax and tokens ........................................... 45
  - XPath operators ............................................................................ 47
  - XPath functions ............................................................................ 49
  - Parenthesized expressions ........................................................... 54
  - Parentheses and subscripts ......................................................... 55
### Contents

Parentheses and unions.......................................................... 56

**CHAPTER 4**

*for xml Mapping Function* ............................................................. 59  
for xml clause........................................................................... 59  
for xml subqueries.................................................................. 63  
for xml schema and for xml all............................................. 65

**CHAPTER 5**

*XML Mappings* ................................................................................. 71  
SQLX options............................................................................. 71  
SQLX option definitions......................................................... 72  
SQLX data mapping................................................................. 82  
Mapping duplicate column names and unnamed columns .... 82  
Mapping SQL names to XML names ...................................... 85  
Mapping SQL values to XML values .................................... 88

**CHAPTER 6**

*XML Support for I18N* ................................................................. 91  
Overview .................................................................................. 91  
Unicode datatypes................................................................. 91  
Surrogate pairs................................................................. 92  
Numeric character representation........................................... 92  
Client-server conversions......................................................... 92  
Character sets and XML data.................................................... 93  
I18N in for xml ........................................................................... 93  
Option strings ........................................................................ 94  
Numeric Character Representation in for xml ....................... 94  
header option ........................................................................ 95  
Exceptions ........................................................................... 95  
Examples .............................................................................. 95  
I18N in xmlparse ........................................................................ 99  
Options .............................................................................. 99  
I18N in xmlextract ................................................................. 99  
NCR option ........................................................................ 100  
Sort ordering in xmlextract ..................................................... 100  
Sort ordering in XML Services ............................................. 100  
I18N in xmlvalidate ................................................................. 101  
NCR option ........................................................................ 101

**CHAPTER 7**

*xmltable()* .................................................................................. 103  
Introduction ............................................................................ 103  
xmltable and derived table syntax ....................................... 103  
xmltable ................................................................................ 104
Contents

APPENDIX A

The sample_docs Example Table.............................................. 123
  sample_docs table columns and rows ................................... 123
  Sample_docs table columns ............................................. 123
  sample_docs table rows .................................................. 124
  sample_docs tables ....................................................... 125
  Table script (for publishers table) ................................... 126
  Publishers table representation ....................................... 126
  Titles table representation ............................................. 127

APPENDIX B

XML Services and External File System Access ....................... 133
  Getting Started ............................................................ 133
  Enabling XML services and External File System Access ....... 133
  Character set conversions with external file systems............ 134
  Examples ................................................................. 134
  Setting up your XML documents and creating the proxy table 135
  Example: extracting the book title from the XML documents 136
  Example: importing XML documents or XML query results to an
  Adaptive Server table .................................................. 136
  Example: storing parsed XML documents in the file system. 137
  Example: ‘xmlerror’ option capabilities with External File Access
  138
  Example: specifying the ‘xmlerror=message’ option in xmlextract
  139
  Example: parsing XML and non-XML documents with the
  ‘xmlerror=message’ option ............................................. 140
  Example: using the option ‘xmlerror=null’ for non-XML documents
  141

APPENDIX C

Migrating Between the Java-based XQL Processor and the Native
XML Processor ................................................................. 143
  Introduction ................................................................. 143
  Migrating documents and queries ..................................... 144
  Migrating documents between the Java-based XQL processor and
  the native XML processor ............................................. 144
  Migrating text documents between the Java-based XQL processor
  and the native XML processor ..................................... 144
  Migrating documents from regenerated copies ..................... 145
  Regenerating text documents from the Java-based XQL processor
  145
  Regenerating text documents from the native XML processor 146
  Migrating queries between the native XML processor and the Java-
  based XQL processor .................................................. 147
APPENDIX D

Sample Application for xmltable() ............................................... 149
Sample table ............................................................................. 149
Using the depts document ......................................................... 153
The depts document structure ................................................... 153
Creating SQL tables from the depts document ......................... 153

Index .......................................................................................... 157
Introduction to XML Services

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML capabilities</td>
<td>1</td>
</tr>
<tr>
<td>Overview</td>
<td>2</td>
</tr>
</tbody>
</table>

XML capabilities

XML Services provides the following capabilities:

- Generating XML: A for xml clause in select commands, which returns the result set as an XML document in the standard SQLX format.

- Storing XML:
  - Support for XML documents stored as either character data in char, varchar, text, unichar, univarchar, or unitext columns, or as parsed XML.
  - xmlparse, which parses and indexes and XML document and generates a parsed and indexed representation for storage.
  - xmlvalidate, which validates the XML document against DTD or XML schema definitions.

- Querying and shredding XML: xmltest and xmlextract, which query and extract data from XML documents.

- I18N support: Support for Unicode and non-ASCII server character sets in XML documents, including support for generating, storing, querying and extracting XML documents containing non-ASCII data.
Overview

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

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

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

XML in the database

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

- XML lets you develop and use tags that distinguish different types of data, such as customer numbers or item numbers.

- XML lets you create an application-specific document type, making it possible to distinguish one kind of document from another.

- XML documents allow different displays of the XML data. XML documents, like HTML documents, contain only markup and content; they do not contain formatting instructions. Formatting instructions are normally provided on the client.

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

You can store XML documents in Adaptive Server as:

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

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

It might display on your screen like this:

ORDER
Date: July 4, 2003
Customer ID: 123
Customer Name: Acme Alpha

Items:

<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>
  <CustomerId>123</CustomerId>
  <CustomerName>Acme Alpha</CustomerName>
  <Item>
    <ItemId>987</ItemId>
    <ItemName>Coupler</ItemName>
    <Quantity>5</Quantity>
  </Item>
  <Item>
    <ItemId>654</ItemId>
    <ItemName>Connector</ItemName>
    <Quantity unit="12">3</Quantity>
  </Item>
  <Item>
    <ItemId>579</ItemId>
    <ItemName>Clasp</ItemName>
  </Item>
</Order>
```
The XML document has two unique characteristics:

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

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

- The XML declaration, <?xml version="1.0"?>, identifying “Order” as an XML document.
- 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>
  <Quantity>1</Quantity>
  <Item/>
</Order>
```
<OneTag>1999/07/04</OneTag>
<AnotherTag>123</AnotherTag>
>LastTag>Acme Alpha</LastTag>
<Thing>
  <ThingId>987</ThingId>
  <ThingName>Coupler</ThingName>
  <Amount>5</Amount>
</Thing>
<Thing>
  <ThingId>654</ThingId>
  <ThingName>Connector</ThingName>
</Thing>
<Thing>
  <ThingId>579</ThingId>
  <ThingName>Clasp</ThingName>
  <Amount>1</Amount>
</Thing>
</Info>

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

**HTML display of Order data**

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

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

ORDER

Date: July 4, 1999

Customer ID: 123

Customer Name: Acme Alpha

Items:

<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>
</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" cellpadding=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 indicate type style (<b>....</b>), color (<bcolor=white>), and layout (<table>....</table>, as well as the supplementary field names, such as Customer Name, and so on.
- The structure of HTML documents is not well suited for extracting data.
Some elements, such as tables, require strictly bracketed opening and closing tags, but other elements, such as paragraph tags ("<p>") have optional closing tags.

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

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

**XML document types**

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

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

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

Line by line, this DTD specifies that:

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

The “<ATTLIST…>” definition in the last line specifies that quantity elements have a “units” attribute; “#IMPLIED”, at the end of the last line, indicates that the “units” attribute is optional.

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

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

### XML query functions

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

The functions are:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML query functions</td>
<td>11</td>
</tr>
<tr>
<td>xmlextract</td>
<td>12</td>
</tr>
<tr>
<td>xmltest</td>
<td>19</td>
</tr>
<tr>
<td>xmlparse</td>
<td>23</td>
</tr>
<tr>
<td>xmlrepresentation</td>
<td>27</td>
</tr>
<tr>
<td>xmlvalidate</td>
<td>29</td>
</tr>
<tr>
<td>option_strings: general format</td>
<td>39</td>
</tr>
</tbody>
</table>
**xmlextract**

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

**Example sections**

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

**xmlextract**

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

**Syntax**

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

Description

Note For information on I18N data, see “XML Support for I18N” on page 91.

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

- If the `returns_type` specifies a `string_type`, the initial result value is returned as a character-string document of that datatype.
- If the `returns_type` specifies a `computational_type` or `datetime_type` datatype, the initial result is converted to that datatype and returned. The conversion follows the rules specified for the `convert` built-in function.

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

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

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

**option_string**

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

The options supported for the `xmlextract` function are:

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

For a description of the `ncr` option, including its default value, see “XML Support for I18N” on page 91.

**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.
• If the explicit or default option specifies xmlerror=null a null value is returned.

• If the explicit or default options specifies xmlerror=message, a character string containing an XML element, which contains the exception message, is returned. This value is valid XML.

• Global variable @@error returns the error number of the last error, whether the value of xmlerror is exception, null, or message.

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

Examples

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

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

```sql
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

```sql
select xmlextract('//row[price<10 | city="Boston"] /pub_id',
    text_doc) from sample_docs
```

```
NULL
XML Services<pub_id>0736</pub_id>
```
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
selectxmlextract
('/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
'&lt;price&gt;12&lt;/price&gt;' 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'
```

```
12
```

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

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

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

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

To illustrate the xmlerror options, the following command inserts an invalid document into the sample_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
```
XMLPARSE(): XML parser fatal error
<<The input ended before all started tags were ended. Last tag started was 'a'>>
at line 1, offset 23.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This command restores the sample_docs table to its original state:

    delete from sample_docs
    where na_doc='invalid doc'

xmltest

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

Syntax

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

**Description**

*Note* For information on processing I18N data, see Chapter 6, “XML Support for I18N.”

- A *basic_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, or java.lang.String.
- A *general_string_expression* is a *sql_query_expression* whose datatype is character, varchar, unichar, univarchar, text, unitext, or java.lang.String.
- An xmltest predicate can be used in SQL language wherever a SQL predicate is allowed.
- An xmltest call specifying that:
  \[ X \ not \ xmltest \ Y \ options \ Z \]

  is equivalent to:

  \[ \text{not} \ X \ xmltest \ Y \ options \ Z \]

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

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

• Restrictions on external URI references, XML namespaces, and XML schemas.

• Treatment of predefined entities and their corresponding characters: `&amp; (&), &lt; (<), &gt; (>), &quote; (’), and &apos; (‘`). Be careful to include the semicolon as part of the entity.

• Treatment of whitespace.

• Treatment of empty elements.

**Options**

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

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

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

**Exceptions**

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

• If the explicit or default option specifies `xmlerror=exception`, an exception is raised.

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

• If you specify `xmlerror=message`, a null value is returned.

**Examples**

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

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

```sql
select name_doc from sample_docs
where '//row[city="Boston"]' xmltest text_doc
name_doc
------------------------
publishers
(1 row affected)
```
In the following example the xmltest predicate returns false/true, for a Boolean false/true result and for an empty/not-empty result.

-- A boolean true is 'true':
select case when '/a=“A”' xmltest '<a>A</a>'
    then 'true' else 'false' 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:

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

(1 row affected)

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

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

23

Chapter 2   XML Query Functions

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

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

To display the contents of @@error, enter:

    select @@error
    14702
(1 row affected)

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

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

(0 rows affected)

To display the contents of @@error, enter:

    select @@error
    14701
(1 row affected)

This command restores the sample_docs table to its original state:

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

xmlparse

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

Syntax

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

XML Services
xmlparse

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

Description

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

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

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

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

Options

- The general format of the option_string is described in “option_strings: general format” on page 39. The options supported for the xmlparse function are:
  
  dtdvalidate = { yes | no }
  xmlerror = { exception | null | message }
If `dtdvalidate=yes` is specified, the XML document is validated against its embedded DTD (if any).

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

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

For the `xmlerror` option, see “Exceptions” below.

**Exceptions**

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

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

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

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

**Examples**

These examples use the `sample_docs` table described in

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

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

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

```sql
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
```

<table>
<thead>
<tr>
<th>name_doc</th>
<th>extract_from_text_doc</th>
<th>extract_from_image_doc</th>
</tr>
</thead>
<tbody>
<tr>
<td>bookstore</td>
<td>&lt;price&gt;55&lt;/price&gt;</td>
<td>&lt;price&gt;55&lt;/price&gt;</td>
</tr>
<tr>
<td>publishers</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>titles</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

(3 rows affected)

To illustrate the `xmlext` options, this 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 `xmlext` options determine the treatment of invalid XML documents by the `xmlparse` function:

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

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

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

- If `xmlext=null`, a null value is returned:

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

select image_doc from sample_docs
where name_doc='invalid doc'

• 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'
```

This command restores the sample_docs table to its original state:

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

### `xmlrepresentation` function

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

**Syntax**

```sql
xmlrepresentation_call ::= xmlrepresentation(parsed_xml_expression)
```

**Description**

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

**Examples**

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

**Example 1**  This example illustrates the basic `xmlrepresentation` function.

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

```sql
select xmlrepresentation(
    xmlextract('/', '<a>A</a>' returns image)
--------------
1

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

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

```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'
```

**Example 3**  
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 sample_docs

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

(3 rows affected)

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

```sql
update sample_docs set image_doc = xmlparse(image_doc)
```
Msg 14904, Level 16, State 0:
Line 1:
XMLPARSE: Attempt to parse an already parsed XML
document.

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

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

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

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

xmlvalidate

Validates an XML document.

**Syntax**

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

**Description**

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

- A `basic_string_expression` is a `sql_query_expression` whose datatype is
  character, varchar, unichar, univarchar, or java.lang.String.
- A `general_string_expression` is a `sql_query_expression` whose datatype is
  character, varchar, unichar, univarchar, text, unitext, or java.lang.String.
• If any parameter of xmlvalidate() is null, the result of the call is null.

• The result datatype of an xmlvalidate_call is the datatype specified by the returns_type.

Options

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

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

schema_uri_list ::=  
    schema_uri [schema_uri]...

namespace_schema_uri_list ::=  
    namespacename schema_uri  
    [ namespacename schema_uri ]...

schema_uri ::= character_string

namespacename ::= character_string

Options description

• The defaults for validation_options are:
  
  dtdvalidate = See below  
  schemavalidate = no  
  schemalocation = ""  
  nonamespaceschemalocation = ""  
  xmlerror = exception  
  xmlvalid = document

• Keywords in a validation_option are not case-sensitive, but the schema_uri_list and namespace_schema_uri_list are case-sensitive.

• Refer to the document you parse or store as the subject XML document.

• The default for dtdvalidate depends on the implicit or explicit value of the schemavalidate option. If the schemavalidate option value is no, the default value of dtdvalidate is no. If the schemavalidate option value is yes, the default dtdvalidate option value is strict.

• If you specify schemavalidate = yes, you must either specify dtdvalidate = strict or omit dtdvalidate.

• If you specify dtdvalidate = no with schemavalidate = no, the document is checked for well-formedness only.
• If you specify schemavalidate = no, the clauses nonamespaceschemalocation and schemalocation are ignored.

• The values specified in the clauses nonamespaceschemalocation and schemalocation are character literals. If the Transact-SQL quoted_identifier option is off, you can choose either apostrophes (') or quotation marks ("), to surround the option_string, and use the other to surround the values specified by nonamespaceschemalocation and schemalocation. If the Transact-SQL quoted_identifier option is on, you must surround the option_string with apostrophes ('), and you must surround the values specified by nonamespaceschemalocation and schemalocation by quotation marks (").

• nonamespaceschemalocation specifies a list of schema URIs, which overrides the list of schema uris specified in the xsi:noNameSpaceschemalocation clause in the subject XML document.

• schemalocation specifies a list of pairs, each pair consisting of a namespace name and a schema URI.
  • a namespace name is the name an xmlns attribute specifies for a namespace. http://acme.com/schemas.contract is declared as the default namespace in this example:

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

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

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

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

• A schema_uri is a character string literal that contains a schema URI. The maximum length of a URL_string is 1927 characters, and it must specify http. The schema referenced by a schema_uri must be encoded as either UTF8 or UTF16.

• The dtdvalidate option values are:
  dtdvalidate=no: No DTD or schema validation is performed; the document is checked to ensure that it is well-formed.

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

  dtdvalidate=strict: This option depends on the schemavalidate option.
• **schemavalidate=no:**
  You must specify a DTD in the subject XML document, and the document is validated against that DTD.

• **schemavalidate=yes:**
  You must declare every element in the subject XML document in a DTD or a schema, and each element is validated against those declarations.

- The **schemavalidate** option values are:
  - If you specify `schemavalidate=no`, no schema validation is performed for the subject XML document.
  - If you specify `schemavalidate=yes`, schema validation is performed.

- The following results apply when a `general_string_expression`, for instance `XC`, is an XML document that passes the validation options specified in the `option_string` clause:
  - If `xmlvalid` specifies `doc`, the result of `xmlvalidate` is:
    ```
    convert(text, XC)
    ```
  - If `xmlvalid` specifies `message`, the result of `xmlvalidate` is this XML document:
    ```xml
    <xmlvalid/>
    ```

- The following results apply when a `general_string_expression` is not an XML document that passes the validation options specified in the `option_string` clause:
  - If the `option_string` specifies `xmlerror=exception`, an exception is raised carrying the `exception` message.
  - If `option_string` specifies `xmlerror=message`, an XML document of the following form is returned. E1, E2, and so forth are messages that describe the validation errors.
    ```xml
    <xml_validation_errors>
    <xml_validation_error>E1</xml_validation_error>
    <xml_validation_error>E2</xml_validation_error>
    ...
    <xml_validation_warning>W1</xml_validation_warning>
    <xml_validation_fatal_error>E3</xml_validation_fatal_error>
    </xml_validation_errors>
    ```
  - If `option_string` specifies `xmlerror=null`, a null value is returned.

**Exceptions**

- If the value of the `xml_data_expression` is not valid XML:
Chapter 2    XML Query Functions

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

Examples
The XML DTDs and schemas shown in Table 2-2 illustrate validation clauses.
- dtd_emp and schema_emp define a single text element, "<emp_name>"
- dtd_cust and schema_cust define a single text element, "<cust_name>"
- ns_schema_emp and ns_schema_cust are variants that specify a target namespace.

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

<table>
<thead>
<tr>
<th>URI</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://test/dtd_emp.dtd">http://test/dtd_emp.dtd</a></td>
<td>&lt;!ELEMENT emp_name (#PCDATA)&gt;</td>
</tr>
<tr>
<td><a href="http://test/dtd_cust.dtd">http://test/dtd_cust.dtd</a></td>
<td>&lt;!ELEMENT cust_name (#PCDATA)&gt;</td>
</tr>
<tr>
<td><a href="http://test/schema_emp.xsd">http://test/schema_emp.xsd</a></td>
<td>&lt;xsd:schema xmlns:xsd</td>
</tr>
<tr>
<td></td>
<td>=&quot;<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>&quot;</td>
</tr>
<tr>
<td></td>
<td>targetNamespace</td>
</tr>
<tr>
<td></td>
<td>=&quot;<a href="http://test/ns_schema_emp%22%3E">http://test/ns_schema_emp&quot;&gt;</a></td>
</tr>
<tr>
<td></td>
<td>&lt;xsd:element name=&quot;emp_name&quot;</td>
</tr>
<tr>
<td></td>
<td>type=&quot;xsd:string&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xsd:schema&gt;</td>
</tr>
<tr>
<td><a href="http://test/ns_schema_emp.xsd">http://test/ns_schema_emp.xsd</a></td>
<td>&lt;xsd:schema xmlns:xsd</td>
</tr>
<tr>
<td></td>
<td>=&quot;<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>&quot;</td>
</tr>
<tr>
<td></td>
<td>targetNamespace</td>
</tr>
<tr>
<td></td>
<td>=&quot;<a href="http://test/ns_schema_emp%22%3E">http://test/ns_schema_emp&quot;&gt;</a></td>
</tr>
<tr>
<td></td>
<td>&lt;xsd:element name=&quot;emp_name&quot;</td>
</tr>
<tr>
<td></td>
<td>type=&quot;xsd:string&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xsd:schema&gt;</td>
</tr>
</tbody>
</table>
xmlvalidate

<table>
<thead>
<tr>
<th>URI</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://test/schema_cust.xsd">http://test/schema_cust.xsd</a></td>
<td>&lt;xsd:schema xmlns:xsd=&quot;<a href="http://www.w3.org/2001/XMLSchema%22%3E%5Cn">http://www.w3.org/2001/XMLSchema&quot;&gt;\n</a></td>
</tr>
</tbody>
</table>

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

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

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

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

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

```
insert into text_docs
values(xmlvalidate(\n   '<emp_name>John Doe</emp_name>',\n   option 'dtdvalidate=strict'))
----------
EXCEPTION
```
Example 4  The last example raised an exception when validation failed. Instead, you can use the option xmlerror to specify that xmlvalidate should return null when validation fails.

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

null
```

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

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

<xml_validation_errors>
  <xml_validation_error>(1:15)Document is invalid:
    no grammar found.<xml_validation_error>
  <xml_validation_error>(1:15)Document root element
    "employee name",must match DOCTYPE root
    "null."</xml_validation_error>
</xml_validation_errors>
```

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

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

(1 row inserted)
```

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

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

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

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

---

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

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

(1 row inserted)

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

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

(1 row inserted)

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

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

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

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

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

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


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

```sql
insert into text_docs
values (xmlvalidate(
    '<emp:emp_name
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:emp="http://test/schema_emp"
    xsi:schemaLocation="http://test/ns_schema_emp
    http://test/schema_cust.xsd">
    John Doe</emp:emp_name>',
```
Example 14  This example shows `xmlvalidate` specifying a document with a schema declaration and the validation option `schemavalidate=yes`, as well as the clauses `schemalocation` and `nonamespaceschemalocation`.


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

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


This command fails, because the document doesn’t match the schema referenced in `xmlvalidate`. The document does, however, match the schema referenced in the document.
option_strings: general format

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

Any function that has an option_string parameter accepts the union of all options, and ignores any options that do not apply to that particular function. This “union options” approach lets you use a single option_string variable for all XML Services functions.

Syntax

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>option_string::= basic_string_expression</td>
</tr>
<tr>
<td>option_string_value ::= option [ , ] option ...</td>
</tr>
<tr>
<td>option ::= name = value</td>
</tr>
<tr>
<td>name ::= option name as listed below</td>
</tr>
<tr>
<td>value ::= simple_identifier</td>
</tr>
</tbody>
</table>

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

- If an option_string parameter is null, the empty strings are all blanks.
- You can use any amount of white space before the first option, after the last option, between options, and around the equals signs.
- You can separate options using commas or by white space.
- An option_value can be either a simple identifier, beginning with a letter and continuing with letters, digits, and underscores, or a quoted string. Quoted strings are formed using the normal SQL conventions for embedded quotes.
The set of options, and the functions to which they are applicable, are shown in Table 2-3. See specific function descriptions for descriptions of options.

### Option values for query functions.

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

<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>dtdvalidate</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>entitize</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>nonnamespaceschemalocation</td>
<td>See xmlvalidate</td>
<td>xmlvalidate</td>
</tr>
<tr>
<td>ncr</td>
<td>non_ascii</td>
<td>non_server</td>
</tr>
<tr>
<td>nullstyle</td>
<td>attribute</td>
<td>omit</td>
</tr>
<tr>
<td>prefix</td>
<td>SQL name (C)</td>
<td>The default value is C.</td>
</tr>
<tr>
<td>root</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>rowname</td>
<td>SQL name (row)</td>
<td>for xml clause</td>
</tr>
<tr>
<td>schemalocation</td>
<td>See xmlvalidate</td>
<td>for xml clause</td>
</tr>
<tr>
<td>schemavalidate</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>statement</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>tablename</td>
<td>SQL name (resultset)</td>
<td>for xml clause</td>
</tr>
<tr>
<td>targets</td>
<td>quoted string with a URI</td>
<td>for xml clause</td>
</tr>
<tr>
<td>xmlerror</td>
<td>exception</td>
<td>null</td>
</tr>
<tr>
<td>xmlvalid</td>
<td>document</td>
<td>message</td>
</tr>
<tr>
<td>xsidectl</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
CHAPTER 3

XML Language and XML Query Language

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character set support</td>
<td>41</td>
</tr>
<tr>
<td>URI support</td>
<td>41</td>
</tr>
<tr>
<td>Namespace support</td>
<td>42</td>
</tr>
<tr>
<td>XML schema support</td>
<td>42</td>
</tr>
<tr>
<td>Predefined entities in XML documents</td>
<td>42</td>
</tr>
<tr>
<td>Predefined entities in XPath queries</td>
<td>44</td>
</tr>
<tr>
<td>White space</td>
<td>44</td>
</tr>
<tr>
<td>Empty elements</td>
<td>45</td>
</tr>
<tr>
<td>XML Query Language</td>
<td>45</td>
</tr>
<tr>
<td>Parenthesized expressions</td>
<td>54</td>
</tr>
</tbody>
</table>

Character set support

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

URI support

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

There are no restrictions on the use of URIs as href attributes or document text, and XML Services resolves external reference URIs that specify http URIs. External-reference URIs that specify file, ftp, or relative URIs are not supported.

Namespace support

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

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

XML schema support

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

Predefined entities in XML documents

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

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

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

Msg 14702, Level 16, State 0:
Line 1:
XMLPARSE(): XML parser fatal error <<A '<' character cannot be used in attribute 'atr', except through '<&gt;' at line 1, offset 14.
select xmlparse("<a atr1='&'>")

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

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

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

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

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

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

<a atr="&lt; &amp;"> &lt; &amp; </a>

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

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

<a atr=" "yes" "/></a>

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

<a atr=" 'yes' " /></a>

You can use quotation marks and apostrophes within elements. They are replaced by the predefined entities &quot; and &apos;, as the following example shows:
Predefined entities in XPath queries

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

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

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

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

White space

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

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

```xml
<a atr=" this or that ">
  <b> which or what </b>
</a>
```
Empty elements

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

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

```xml
<doc>
  <a/>
  <b/></doc>
```

XML Query Language

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

XPath-supported syntax and tokens

XML Services supports the following XPath syntax:

```
xpath::= or_expr
  or_expr::= and_expr | and_expr TOKEN_OR or_expr
  and_expr::= union_expr | union_expr TOKEN_AND and_expr
  union_expr::= intersect_expr
    | intersect_expr TOKEN_UNION union_expr
  intersect_expr::= comparison_expr
    | comparison_expr TOKEN_INTERSECT intersect_expr
  comparison_expr::= range_exp
    | range_expr general_comp comparisonRightHandSide
  general_comp::= TOKEN_EQUAL | TOKEN_NOTEQUAL
```
<table>
<thead>
<tr>
<th>TOKEN_LESSTHAN</th>
<th>TOKEN_LESSTHANEQUAL</th>
<th>TOKEN_GREATERTHAN</th>
<th>TOKEN_GREATERTHANEQUAL</th>
</tr>
</thead>
</table>
range_expr::= unary_expr | unary_expr TOKEN_TO unary_expr
unary_expr::= TOKEN_MINUS path_expr
| path_expr
comparisonRightHandSide::= literal
pathExpr::= relativepath_expr | TOKEN_SLASH relativepath_expr
| TOKEN_DOUBLESLASH relativepath_expr
relativepath_expr::= step_expr
| step_expr TOKEN_SLASH relativepath_expr
| step_expr TOKEN_DOUBLESLASH relativepath_expr
step_expr::= forward_step predicates
| primary_expr predicates
| predicates
primary_expr::= literal | function_call | xpath
function_call::=
| tolower([xpath])
| toupper([xpath])
| normalize-space([xpath])
| concat([xpath [,xpath]...])
forward_step::= abbreviated_forward_step
abbreviated_forward_step::= name_test
| TOKEN_ATRATE name_test
| TOKEN_PERIOD
name_test::= q_name | wild_card | text_test
text_test ::= TOKEN_TEXT TOKEN_LPAREN TOKEN_RPAREN
literal::= numeric_literal | string_literal
wild_card::= TOKEN_ASTERISK
q_name::= TOKEN_ID
string_literal::= TOKEN_STRING
numeric_literal::= TOKEN_INT | TOKEN_FLOATVAL |
| TOKEN_MINUS TOKEN_INT
| TOKEN_MINUSTOKEN_FLOATVAL
predicates::=
| TOKEN_LSQUARE expr TOKEN_RSQUARE predicates
| TOKEN_LSQUARE expr TOKEN_RSQUARE
The following tokens are supported by the XML Services subset of XPath:

APOS ::= "'
DIGITS ::= [0-9]+
NONAPOS ::= "\nNONQUOTE ::= "\nNONSTART ::= LETTER | DIGIT | '.' | '-' | '_' | '\'
QUOTE ::= "'
START ::= LETTER | '\'
TOKEN_AND ::= "and"
TOKEN_ASTERISK ::= "'
TOKEN_ATRATE ::= '@'
TOKEN_COMMA ::= ','
XPath operators

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

XPath basic operators

Table 3-1 shows the supported basic XPath operators.
Table 3-1: XPath basic operators

<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 where clause with any semantics. The filter contains a query within it, called the sub-query. If a collection is placed within the filter, a Boolean “true” is generated if the collection contains any members, and a “false” is generated if the collection is empty.</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>text()</td>
<td>Selects the text nodes of the current context node.</td>
</tr>
</tbody>
</table>

XPath set operators

Table 3-2 on page 49, shows the supported XPath set operators.
Chapter 3  XML Language and XML Query Language

Table 3-2: XPath set operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>Union: union operator (shortcut is ‘</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 (and and or)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean operators can be used within subqueries.</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>Boolean “and”.</td>
</tr>
<tr>
<td>or</td>
<td>Boolean “or”.</td>
</tr>
</tbody>
</table>

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>=</td>
<td>equality</td>
</tr>
<tr>
<td>!=</td>
<td>non-equality</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>less than equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>greater than equal</td>
</tr>
</tbody>
</table>

XPath functions

Adaptive Server supports the following XPath string functions:

- toupper
- tolower
General guidelines and examples

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

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

**Example 1**

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

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

```
joe
```

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

**Example 2**

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

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

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

**Example 3**

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

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

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

Example 4
Both top-level and relative functions can use literals as parameters. For example:

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

| abc | abc |

Example 5
String functions operate on the text of their parameters. This is an implicit application of `text()`. For example, this query returns a first-name element as an XML fragment:

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

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

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

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

```
Joe
```

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

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

```
joe
```

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

```sql
select xmlextract
( '//book[title="Seven Years in Trenton"]//tolower(first-name/text())', 
```
Example 6  
You apply a relative function call as a step in a path. Evaluating that path produces a sequence of XML nodes, and performs a relative function call for each node. The result is a sequence of the function call results. For example, this query produces a sequence of first_name nodes:

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

```
<joe>
</joe>
<joe>Mary</joe>
<joe>Toni</joe>
```

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

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

```
JOEMARYTONI
```

Now you can use concat to punctuate the sequence of the function results. See the example in “concat” on page 54.

Example 7  
tolower, toupper, and normalize-space each have a single parameter. If you omit the parameter when you specify these functions in a relative function call, the current node becomes the implicit parameter. For instance, this example shows a relative function call of tolower, explicitly specifying the parameter:

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

```
joe
```

This example of the same query specifies the parameter implicitly:

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

```
joe
```

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

This section describes the individual functions that enhance XML Services.

tolower and toupper

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

Syntax
tolower(string-parameter)
toupper(string-parameter)

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

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

normalize-space

Description
Makes two changes when it returns its argument value:

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

Syntax
normalize-space(string-parameter)

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

```
select xmlextract('normalize-space(" Normal space example. ")', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------------
Normalize space example.
```
normalize-space and tolower or toupper are useful in XPath predicates, when you are testing values whose use of white space and case is not known. The following predicate is unaffected by the case and whitespace usage in the title elements:

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

**concat**

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

**Syntax**
```
cat(string-parameter[, string-parameter]...)
```

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

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

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

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

**Parenthesized expressions**

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

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

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

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

```
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Tracking Trenton</title>
<title>Treanton Today, Trenton Tomorrow</title>
<title>Whos Who in Trenton</title>
```

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

```sql
select xmlextract(/bookstore/book/title[1], text_doc)
from sample_docs where name_doc='bookstore'
```

```
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Tracking Trenton</title>
<title>Treanton Today, Trenton Tomorrow</title>
<title>Whos Who in Trenton</title>
```

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

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

```
NULL
```

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

```sql
select smlextract
  ('(/bookstore/booktitle)[2]', text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------------------
<title>History of Trenton</title>
You can group any path with parentheses. For example:
```n
```sql
select xmlextract('(/title)[2]', text_doc)
from sample_docs where name_doc='bookstore'
---------------------------------------------
<title>History of Trenton</title>
```

Parentheses and unions

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

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

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

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

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

```sql
select xmlextract('/bookstore/*/title', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Tracking Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>
<title>Who's Who in Trenton</title>
```
If the bookstore contains elements other than books and magazines—such as calendars and newspapers—you can query only for book and magazine titles by using the union (vertical bar) operator, and parenthesizing it in the query path. For example:

```
select xmlextract('/bookstore/(book|magazine)/title', text_doc)
from sample_docs where name_doc='bookstore'
----------------------------------
<title>Seven Years in Trenton</title>
<title>History of Trenton</title>
<title>Tracking Trenton</title>
<title>Trenton Today, Trenton Tomorrow</title>
<title>Whos Who in Trenton</title>
```
Parenthesized expressions
CHAPTER 4

for xml Mapping Function

This chapter describes the for xml XML mapping function in detail, and provides examples.

for xml clause

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

Syntax

\[
select ::= 
select [ all | distinct ] select_list
[ into_clause ]
[ where_clause ]
[ group_by_clause ]
[ having_clause ]
[ order_by_clause ]
[ compute_clause ]
[ read_only_clause ]
[ isolation_clause ]
[ browse_clause ]
[ plan_clause ]
[ for_xml_clause ]
\]

for_xml_clause ::= 
for xml [schema | all] [option option_string] [returns_clause]

option_string ::= basic_string_expression

returns_clause ::= 
returns { char [(integer)] | varchar [(integer)]
| unichar [(integer)] | univarchar [(integer)]
| text | untext | java.lang.String}
**for xml clause**

**Description**

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

- The syntax and description of the other `select` statement clauses are in *Sybase Adaptive Server Reference Manual, Volume 2: “Commands.”*

- The `for xml` clause supports the `java.lang.string` datatype, represented as `string`. Any other Java type is represented as `objectID`.

- **Note** For a description of `for xml schema` and `for xml all`, see “for xml schema and for xml all” on page 65.

The variants of the `for xml` clause are as follows:

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

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

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

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

c. A `for xml all select` command or subquery has a `for_xml_clause` that specifies `all`.

- A `for xml select` statement cannot include an `into_clause`, `compute_clause`, `read_only_clause`, `isolation_clause`, `browse_clause`, or `plan_clause`.

- `for xml` 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:

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

But this statement is not allowed:

```sql
select * from T for xml
union
select * from U
```
The value of `for xml select` is an XML representation of the result of the basic `select` statement. The format of that XML document is the SQLX format described in Chapter 5, “XML Mappings.”

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

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

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

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

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

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

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

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

```sql
select getdate() for xml
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
< row >
< C1 > 2008-05-30 11:42:19 < /C1 >
< /row >
< /resultset >
```

But in Adaptive Server 15.0.2, the results from the same query are:
select getdate() for xml
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <row>
 <C1>2008-05-30T11:41:42</C1>
 </row>
</resultset>

Options
The general format of the option_string is specified in “option_strings: general format” on page 39. The options for the for xml clause are specified in “SQLX options” on page 71.

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
----------------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <row>
 <pub_id>0736</pub_id>
 <pub_name>NewAgeBooks</pub_name>
 </row>

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

 <row>
 <pub_id>1389</pub_id>
 <pub_name>Algodata Infosystems</pub_name>
 </row>
</resultset>
```
**for xml subqueries**

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

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

**Syntax**

```
subquery ::= select [all | distinct ] select_list
            (select select_list
             [from table_reference [, table_reference]... ]
             [where search_conditions]
             [group by aggregate_free_expression [aggregate_free_expression]...]
             [having search_conditions]
             [for_xml_clause])

for_xml_clause ::= See “for xml schema and for xml all” on page 64


table_reference ::= table_view_name | ANSI_join | derived_table


derived_table ::= (subquery) as table_name
```

**Description**

- A `select` command containing a `for xml` clause generates an XML document that represents the results of the `select` statement, and returns that XML document as a result set, with a single row and a single column. You can access that result set using normal techniques for processing result sets.

- For a general description of the `for xml` clause and its `option_string`, see “for xml clause” on page 59. For a description of extensions to the `for xml` clause that support the `SCHEMA` keyword and the `return` clause, see “for xml schema and for xml all” on page 65.

- A `for xml` subquery is a subquery that contains a `for xml` clause.

- You can use a `for xml` subquery as an expression subquery, though there are some differences between them; for example, the following restrictions apply to ordinary expression subqueries, but not to `for xml` subqueries:
  - No multiple items in the `select` list
  - No text and image columns in the `select` list
  - No group by or having clauses
for xml clause

- You cannot specify a for xml subquery within a for xml select or within another for xml subquery.
- You cannot use a for xml subquery in these commands:
  - for xml select
  - create view
  - declare cursor
  - select into
  - as a quantified predicate subquery, such as any/all, in/not in, exists/not exists
- A for xml subquery cannot be a correlated subquery. For more information on correlated subqueries, see the Transact-SQL User’s Guide.
- A for xml clause that returns a text or unitext datatype cannot be used in a nested scalar subquery.
- The datatype of a for xml subquery is specified by the returns clause of the for_xml_clause. If a returns clause specifies no datatype, the default datatype is text.

Exceptions

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

Examples

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

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

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

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

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

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

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

for xml schema and for xml all

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

Description

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

  (subquery for xml schema option option_string)

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

Options

The general format of the option_string is specified in “option_strings: general format” on page 39. The options for the for xml clause are specified in “SQLX options” on page 71.

Exceptions

The exceptions to extensions are the same as those specified in “SQLX options” on page 71.

Examples—Usage

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

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

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

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

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

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

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

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

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

Examples—results

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

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

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

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

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

```xml
select "a", 1 for xml schema
--------------
<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="VARCHAR_1"/>
    <xsd:element name="C2" type="INTEGER"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="TableType.resultset"
  <xsd:sequence>
    <xsd:element name="row" type="RowType.resultset"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="VARCHAR_1">
  <xsd:restriction base="xsd:string">
    <xsd:length value="1"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="INTEGER">
  <xsd:restriction base="xsd:integer">
    <xsd:maxInclusive value="2147483647"/>
    <xsd:minInclusive value="-2147483648"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="resultset" type="TableType.resultset"/>
</xsd:schema>
(1 row affected)
```

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

```sql
select 'a', 1 for xml all
```
for xml clause

---------
<multiple results>
<multiple-results-item type="result-set">
<multiple-results-item-dtd>
<!DOCTYPE resultset [ 
<!ELEMENT resultset (row*)>
<!ELEMENT row (C1,C2)>
<!ELEMENT C1(#PCDATA)>
<!ELEMENT C2(#PCDATA)>
]>
</multiple-results-item-dtd>
</multiple-results-item-schema>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
<xsd:import namespace="http://2www.w3.org/2001/XMLSchema"
schemaLocation="http://www.iso-standards.org/mra/9075/sqlx.xsd"/>
<xsd:complexType name="RowType.resultset">
<xsd:sequence>
<xsd:element name="C1" type="VARCHAR_1" />
<xsd:element name="C2" type="INTEGER" />
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="TableType.resultset">
<xsd:sequence>
<xsd:element name="row" type="RowType.resultset" minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="VARCHAR_1">
<xsd:restriction base="xsd:string">
<xsd:length value="1"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:element name="resultset" type="TableType.resultset"/>

68 Adaptive Server Enterprise
<xsd:schema>

</xsd:schema>

<multiple-results-item-data>

<multiple-results-item-data>

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

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

</resultset>

</multiple-results-item-data>

</multiple-results-item>

</multiple-results>

(1 row affected)
for xml clause
CHAPTER 5

XML Mappings

The `FOR XML` clause in select statements maps 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 the `FOR XML` clause.

**Note** When you use `isql` to generate an XML document with the `FOR XML` clause, the documents you generate may be invalid, due to a leading blank space added as a column separator by `isql`.

<table>
<thead>
<tr>
<th>Topics</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLX options</td>
<td>71</td>
</tr>
<tr>
<td>SQLX data mapping</td>
<td>82</td>
</tr>
</tbody>
</table>

**SQLX options**

**Note** In Table 5-1, underlined words specify the default value.
### SQLX option definitions

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

<table>
<thead>
<tr>
<th>Option name</th>
<th>Option value</th>
<th>Purpose</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>entityize</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>multipleentitize</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>nullstyle</td>
<td>attribute</td>
<td>omit</td>
</tr>
<tr>
<td>ncr</td>
<td>non_ascii</td>
<td>non_server</td>
</tr>
<tr>
<td>prefix</td>
<td>SQL name</td>
<td>Base for generated names. Default value is C.</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. Default value is row.</td>
</tr>
<tr>
<td>schemaloc</td>
<td>quoted string with a URI</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 element. Default value is resultset.</td>
</tr>
<tr>
<td>targets</td>
<td>quoted string with a URI</td>
<td>targetnamespace value (if any)</td>
</tr>
<tr>
<td>xsidel</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
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 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"
---------------------------------
<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:

```sql
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"
```
entitize = \{yes | no | cond\}

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

For example, this example entitizes all string columns:
select 'a<b' for xml option 'entitize=yes'
----------
<resultset>
 <row>
   <C1><a&lt;b</C1>
 </row>
</resultset>

This example, however, entitizes no string column:
select '<ab>' for xml option 'entitize=no'
-------
<resultset>
 <row>
   <C1><ab></C1>
 </row>
</resultset>

This example entitizes string columns that do not begin with "<":
select '<ab>', 'a<b' for xml option 'entitize=cond'
---------
<resultset>
 <row>
   <C1><ab></C1>
   <C2>a&lt;b</C2>
 </row>
</resultset>

format={yes | no}  This option specifies whether or not to include formatting for newline and tab characters.

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

header=
{yes | no | encoding}  This option indicates whether or not to include an XML header line in the generated SQLX-XML documents. The XML header line is as follows:

<?xml version="1.0?>
SQLX options

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

For a description of the encoding option, see “XML Support for I18N” on page 91.

For example:

```sql
select 1,2 for xml option "header=yes"
```

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

incremental=(yes | no)  This option applies only to the for xml clause, not to the forxml function. It specifies which of the following a select statement with a for xml clause returns:

- `incremental=no` – returns a single row with a single column of datatype text, containing the complete SQLX-XML document for the result of the select statement. `incremental=no` is the default option.

- `incremental=yes` – returns a separate row for each row of the result of the select statement, with a single column of datatype text that contains the XML element for that row.

  - If the `root` option is `yes` (the default), the `incremental=yes` option returns two additional rows, containing the opening and closing elements for the `tablename`.
  
  - If the `root` option is `no`, the `tablename` option (explicit or default) is ignored. There are no two additional rows.

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

```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"
```
multipleentitize= {yes | no}

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

See “Numeric character representation” on page 92.

ncr= {no | non_ascii | non_server}

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:

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

```
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <row>
   <C1>11</C1>
 </row>
 <row>
   <C2 xsi:nil="true"/>
 </row>
</resultset>
```

This example shows nullstyle=attribute:

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

```
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <row>
   <C1>11</C1>
   <C2 xsi:nil="true"/>
 </row>
 <row>
   <C1 xsi:nil="true"/>
   <C2>22</C2>
 </row>
</resultset>
```
SQLX options

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

<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>11</C1>
    <C2>12</C2>
  </row>
  <row>
    <C1>21</C1>
    <C2>22</C2>
  </row>
</resultset>

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

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 85.

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">
      <row>
        <C1>11</C1>
        <C2>12</C2>
      </row>
      <row>
        <C1>21</C1>
        <C2>22</C2>
      </row>
    </resultset>
This option specifies a URI to be included as the xsi:SchemaLocation or xsi:noNamespaceSchemaLocation attribute in the generated SQLX-XML document. This option defaults to the empty string, which indicates that the schema location attribute should be omitted.

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

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

```sql
select 1,2
for xml option "schemaloc='http://thiscompany.com/schemalib' "
```

```xml
<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'"
```

```xml
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```
**SQLX options**

```xml
<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"
```

```
<resultset statement="select name_doc
from sample_docs where name_doc like 'book%'">
  <row>
    <name_doc>bookstore</name_doc>
  </row>
</resultset>
```

**tablename=sql_name**  This option specifies a name for the result set. The default **tablename** is **"resultset"**.

The **tablename** option is a SQL name, which can be a regular identifier or delimited identifier. Delimited identifiers are mapped to XML names as described in “Mapping SQL names to XML names” on page 85.

This example shows **tablename=SampleTable**.

```sql
select 11, 12 union select 21, 22
for xml option "tablename=SampleTable"
```

```
<SampleTable xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>11</C1>
    <C2>12</C2>
  </row>
  <row>
    <C1>21</C1>
    <C2>22</C2>
  </row>
</SampleTable>
```
targetns=uri

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

```sql
select 1,2
for xml
option "targetns='http://thiscompany.com/samples'"
```

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

xsidecl=(yes | no)

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

For example:

```sql
select 1 for xml option 'xsidecl=yes'
```

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

```sql
select 1 for xml option 'xsidecl=no'
```

```xml
<resultset>
  <row>
    <C1>1</C1>
  </row>
</resultset>
```

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

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

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

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

SQLX data mapping

This section describes the SQLX-XML format used by the documents
generated by the for xml clause in select statements. 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>---</th>
<th>---</th>
<th>---</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):
select t1.title_id, t2.title_id, t2.advance-t1.advance, 
  t1.price*t1.total_sales, t2.price*t2.total_sales 
from pubs2..titles t1, pubs2..titles t2 
where t1.price=t2.price and t2.advance-t1.advance>3000 
for xml 
----------------------------------------------------
<resultset 
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <title_id1>BU2075</title_id1>
    <title_id2>MC3021</title_id2>
    <C1>4875.00</C1> 
    <C2>55978.78</C2> 
    <C3>66515.54</C3> 
  </row>
  <row>
    <title_id1>MC2222</title_id1>
    <title_id2>BU1032</title_id2>
    <C1>5000.00</C1> 
    <C2>40619.68</C2> 
    <C3>81859.05</C3> 
  </row>
  <row>
    <title_id1>MC2222</title_id1>
    <title_id2>BU7832</title_id2>
    <C1>5000.00</C1> 
    <C2>40619.68</C2> 
    <C3>81859.05</C3> 
  </row>
</resultset>

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

select t1.title_id, t2.title_id, t2.advance-t1.advance, 
  t1.price*t1.total_sales, t2.price*t2.total_sales as C1 
from pubs2..titles t1, pubs2..titles t2 
where t1.price=t2.price and t2.advance-t1.advance>3000 
for xml 
-----------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <title_id1>BU2075</title_id1>
    <title_id2>MC3021</title_id2>
    <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>
  </row>
  <row>
    <title_id1>MC2222</title_id1>
    <title_id2>BU1032</title_id2>
    <column_1>5000.00</column_1>
    <column_2>40619.68</column_2>
    <column_3>81859.05</column_3>
  </row>
  <row>
    <title_id1>MC2222</title_id1>
    <title_id2>BU7832</title_id2>
    <column_1>5000.00</column_1>
    <column_2>40619.68</column_2>
    <column_3>81859.05</column_3>
  </row>
</resultset>
```
### Mapping SQL names to XML names

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

"salary + bonus: "

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

The objectives of the SQLX name mappings are:

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

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

U+nnnn

is replaced with a string of characters of the form:

_xnnnn_

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

_x

and suffixes them with an underscore.

For example, consider the following SQL result set:

```
set quoted_identifier on
select 1 as "a + b < c & d", 2 as "<a xsi: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
------------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <a_x0020__x002B__x0020_b_x0020__x003C__x0020_c_x0020__x0026__x0020_d_x0020_>
      1
    </a_x0020__x002B__x0020_b_x0020__x003C__x0020_c_x0020__x0026__x0020_d_x0020_>
  </row>
</resultset>
```

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

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

For example:

```sql
select 1 as x, 2 as X, 3 as X99, 4 as XML, 5 as XmlDoc
forxml
----------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <x>1</x>
    <X>2</X>
    <X99>3</X99>
    <_x0078_ML>4</_x0078_ML>
    <_x0058_mLdoc>5</_x0058_mLdoc>
  </row>
</resultset>
```

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

```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">
  <_x0020_row_x0020__x0026__x0020_columns_x0020_>
    <C_x0020_1>11</C_x0020_1>
    <C_x0020_2>12</C_x0020_2>
  </_x0020_row_x0020__x0026__x0020_columns_x0020_>
  <_x0020_row_x0020__x0026__x0020_columns_x0020_>
    <C_x0020_1>21</C_x0020_1>
    <C_x0020_2>22</C_x0020_2>
  </_x0020_row_x0020__x0026__x0020_columns_x0020_>
</table_x0020__x0040__x0020_start>
```
Mapping SQL values to XML values

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

Numeric values

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

```sql
select 1, 2.345, 6.78e8 for xml
-----------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>1</C1>
    <C2>2.345</C2>
    <C3>6.78E9</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>'
----------------------

The following example shows SQLX mapping of that character value, with the XML markup characters replaced by their XML entity representations:

```sql
select '<name>Baker's</name>' for xml
---------------------------------------
```
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>&lt;name&gt;"Baker's"&lt;/name&gt;</C1>
  </row>
</resultset>

Binary values

Binary values contained in binary, varbinary, or image columns are represented in either hex or base64 encoding, depending on the option `binary={hex|base64}`. The base64 encoding is more compact. The choice between the two representations depends on the applications that process the XML data.
SQLX data mapping
CHAPTER 6

XML Support for I18N

This chapter discusses the extension of XML Services to support non-ASCII data. This is necessary both to support XML standards that specify a Unicode base and to support XML-based applications across multiple languages.

In this document the term “I18N” stands for internationalization, which begins with “I” and 18 characters later ends in “n.” This term refers to support for Unicode and other characters beyond the ASCII set.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>91</td>
</tr>
<tr>
<td>I18N in for xml</td>
<td>93</td>
</tr>
<tr>
<td>I18N in xmlparse</td>
<td>99</td>
</tr>
<tr>
<td>I18N in xmlextract</td>
<td>99</td>
</tr>
<tr>
<td>I18N in xmlvalidate</td>
<td>101</td>
</tr>
</tbody>
</table>

Overview

The I18N extensions fall into three categories:

- I18N support in the for xml clause, to generate documents that contain non-ASCII data.
- I18N in xmlparse, to store documents containing non-ASCII data.
- I18N in xmlextract and xmltest, to process XML documents and queries containing non-ASCII data.

Unicode datatypes

The following terms refer to categories of datatypes used for Unicode:

- “String datatypes” refers to char, varchar, text, and java.lang.String.
Overview

- “Unicode datatypes” refers to unichar, univarchar, unitext, and java.lang.String.
- “String/Unicode columns” refers to columns whose datatypes are “string datatypes” or “Unicode datatypes.”

Surrogate pairs

“Surrogate pairs” refers to the pair of 16-bit values that Unicode uses to represent any character that may require more than 16 bits.

Most characters are represented within the range [0x20, 0xFFFF], and can be represented with a single 16-bit value. A surrogate pair is a pair of 16 bit values that represent a character in the range [0x010000..0x10FFFF]. See “Example 7” on page 98 for more details.

Numeric character representation

Numeric Character Representation (NCR) is a technique for representing arbitrary characters in ASCII hexadecimal notation in XML documents. For example, the NCR representation of the Euro sign “€” is “&#x20AC;”. This notation is similar to the SQL hexadecimal character notation, u&\20ac;.

Client-server conversions

Unicode data in the server can be:

- UTF-16 data, stored in unichar, univarchar, unitext, and java.lang.String.
- UTF-8 data, stored in char, varchar, and text, when the server character set is UTF-8.

Transferring data between client and server Any one of the three following techniques transfers univarchar or unitext data between client and server:

- Use CTLIB, or, or BCP. Transfer the data as a bit string. The client data is UTF-16, and byte order is adjusted for client-server differences.
- Use ISQL or BCP. Specify “-J UTF-8”. The data is converted between the client UTF-8 and the server UTF-16.
• Use Java. Specify the client character set (whether source or target) in data transfers. You can specify UTF-8, UTF-16BE, UTF-16L, UTF-16LE, UTF-16 (with BOM), US-ASCII, or another client character set.

**Note** If you want to store Unicode XML documents through JDBC, you must mention the connection property 'DISABLE_UNICODE_SENDING', a “false” property that allows you to send Unicode data from the JDBC connection to Adaptive Server.

Techniques for specifying the character set of client files, whether input or output, in client Java applications appear in Java applications in the following sample directory.

$SYBASE/$SYBASE_ASE/sample/JavaXml/JavaXml.zip

This directory also supplies the documents *Using-SQLX-mappings*, section *Unicode and SQLX result set documents*.

**Character sets and XML data**

If you store an XML document in a string column or variable, XML Services assumes that document to be in the server character set. If you store it in a Unicode column or variable, XML Services assumes it to be UTF-16. Any encoding clause in the XML document is ignored.

**I18N in **for xml**

This section discusses extending the **for xml** clause to handle non-ASCII data.

You can specify Unicode columns and string columns containing non-ASCII characters in the **select_list** of the **for xml** clause.

The default datatype in the **returns** clause is **text**.

The resulting XML document is generated internally as a Unicode string and converted, if necessary, to the datatype of the **returns** clause.

For detailed documentation of this clause, see “for xml clause” on page 59.
Option strings

The option string of a for xml clause can specify a u& form of literal, and then contain the SQL notation for characters. Then you can specify Unicode characters for the rowname, tablename, and prefix options. For example, enter:

```sql
select * from T
for xml
options u&'tablename = \0415\0416 rowname = \+01d6d prefix = \0622'
```

If a specified tablename, rowname, or prefix option contains characters that are not valid in simple identifiers, you must specify the option as a quoted identifier. For example, enter:

```sql
select * from T
for xml
options u&'tablename = "chars\0415 and \0416"
rowname = "\+01d6d1 & \+01d160"
prefix = "\0622-"
```

Numeric Character Representation in for xml

The option_string of a select for xml statement includes an ncr option that specifies the representation of string and Unicode columns:

```sql
ncr = {no | non_ascii | non_server}
```

- **ncr = no** specifies that string and Unicode columns are represented as plain values. These plain values are entitized or not entitized according to the entitize option.
- **ncr = non_ascii** and **ncr = non_server** specify that string and Unicode columns that are, respectively, non-ASCII or not members of the default server character set are represented as NCRs. Any characters not converted to NCRs are either entitized or not, according to the entitize option.

The default NCR option in the for xml clause is **ncr = non_ascii**.

The ncr option applies only to column values, not to column names or to names specified in the tablename, rowname, or prefix options. XML does not allow NCRs in element or attribute names.
header option

The header option of the for xml clause is extended with a new encoding value:

header = {yes | no| encoding}

With header=encoding, the header is:

    <?xml version = "1/0" encoding = "UTF-16?">

Using the encoding value indicates that the XML header should be included, and that it should contain an XML encoding declaration.

The default header option is no if:

- The returns datatype is a Unicode datatype
- The ncr option is non-ascii
- The server character set is ISO1, ISO8859_15, ascii_7, or UTF-8.

Otherwise, the default header option is encoding.

Exceptions

None.

Examples

Use the example table generated by the following commands for all the examples following.

create table example_I18N_table (name varchar(10) null, uvcol univarchar(10) null)
------------------
insert into example_I18N_table values('Arabic', u'\622\623\624\625\626')
insert into example_I18N_table values('Hebrew', u'\5d2\5d3\5d4\5d5\5d6')
insert into example_I18N_table values('Russian', u'\410\411\412\413\414')

The example table in Figure 6-1 has two columns:

- A varchar column indicating a language.
I18N in for xml

• A univarchar column with sample characters of that language. The sample characters consist of strings of consecutive letters.

```sql
select * from example_I18N_table
name     uvcol
--------- -----------------------
Arabic   0x06220623062406250626
Hebrew   0x05d205d305d405d505d6
Russian  0x04100411041204130414
```

(3 rows affected)

Example 1

A select command with no variables specified displays the table:

```sql
select * from example_I18N_table
name     uvcol
--------- -----------------------
Arabic   0x06220623062406250626
Hebrew   0x05d205d305d405d505d6
Russian  0x04100411041204130414
```

(3 rows affected)

Example 2

To generate a SQL XML document using a for xml clause, enter:

```sql
select * from example_I18N_table for xml
---------------------------------------------------
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <name>Arabic</name>
    <uvcol>&#x622;&#x623;&#x624;&#x625;&#x626;</uvcol>
  </row>
  <row>
    <name>Hebrew</name>
    <uvcol>&#x5d2;&#x5d3;&#x5d4;&#x5d5;&#x5d6;</uvcol>
  </row>
  <row>
    <name>Russian</name>
    <uvcol>&#x410;&#x411;&#x412;&#x413;&#x414;</uvcol>
  </row>
</resultset>
```
Example 3

By default, the generated SQLX XML document represents the non-ASCII characters with NCRs. If you set the character set property of your browser to Unicode, the document displays the actual non-ASCII characters, respectively Arabic, Hebrew, or Russian, or any non-ASCII characters you select.

If the browser's character set property is not set to Unicode, the Arabic, Hebrew, and Russian characters appear as question marks.

Example 4

If you want the SQLX XML document to contain non-ASCII as plain characters, specify no in the ncr option.

```sql
select * from example_I18N_table for xml
option 'ncr=no' returns unitext
```

Example 5

If you retrieve the Unicode document generated in Example 3 into a client file, specifying UTF-16 or UTF-8 as the target character set, you can display it in a browser. It will then show the actual non-ASCII characters you select.

Example 6

The options `ncr=non_ascii` and `ncr=non_server` in `ncr` translate a character to an NCR only if it is either not ASCII or not in the default server character set. In this example, the expression concatenates ASCII string values with both the ASCII `name` column and the Unicode `uvcol` column. The result of this expression is a string that contains both ASCII and non-ASCII characters. In the generated SQLX XML document, only non-ASCII characters are translated to NCRs:

```sql
select name + '(' + uvcol + ')'  from example_I18N_table2>
for xml option 'ncr=non_ascii'
```

```xml
<resultset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <C1>Arabic(&#x622;&#x623;&#x624;&#x625;&#x626;)</C1>
  </row>
  <row>
    <C1>Hebrew(&#x5d2;&#x5d3;&#x5d4;&#x5d5;&#x5d6;)</C1>
  </row>
</resultset>
```
I18N in for xml

A browser displays the document showing the actual non-ASCII characters, respectively Arabic, Hebrew, and Russian.

Example 7

Most characters are represented by code points in the range [0x20, 0xFFFF], and can be represented with a single 16-bit value. A surrogate pair is a pair of 16 bit values that represent a character in the range [0x010000..0x10FFFF]. The first half of the pair is in the range [0xD800..0xDBFF], and the second half of the pair is in the range [0xDC00..0xDFFF]. Such a pair (H, L) represents the character computed as follows (hex arithmetic):

\[(H - 0xD800) * 400 + (L - 0xDC00)\]

For example, the character “&#x01D6D1” is a lower-case bold mathematical symbol, represented by the surrogate pair D835, DED1:

```sql
select convert(unitext, u'\+1d6d1')
---------------------
0xd835ded1
```

When you specify ncr=non ascii or ncr=non_server to generate a SQLX XML document containing non-ASCII data with surrogate pair characters, the surrogate pairs appear as single NCR characters, not as pairs:

```sql
select convert(unitext, u'\+1d6d1')
for xml option 'ncr=non_ascii"
---------------------
<resultset
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</row>
  <C1>&#x1d6d1;</C1>
</row>
</resultset>
```
CHAPTER 6    XML Support for I18N

I18N in xmlparse

xmlparse supports Unicode datatypes (unichar, univarchar, unitext, and java.lang.String) for the input XML document.

Options

xmlparse parses an XML document and returns a representation of it as an image value containing the parsed document and its internal index. This representation is called a Unicode parsed image XML. *Unicode parsed image XML* is stored in columns of image.

xmlparse converts string datatypes to Unicode. Since string datatypes are in the server character set, which is always a subset of Unicode, conversion is a change in datatype that never raises a conversion exception.

Sort ordering in xmlparse

For details of XML sort ordering, see “Sort ordering in XML Services” on page 100.

xmlparse uses the sort ordering specified by the sp_configure option default xml sort order, and the same ordering for XML indexes. XML stores the sort ordering name in the image generated by xmlparse, called the *parsed XML sort order* of the document.

All functions that reference a parsed XML document raise an exception when the parsed XML sort order is different from the current default XML sort order.

I18N in xmlextract

xmlextract applies an XML query expression to an XML document, and returns the result you select. The input document can be a string datatype, a Unicode datatype, or an image datatype containing either character data or parsed XML.

The returns clause can specify a Unicode datatype as the datatype of the value extracted.
NCR option

xmlextract supports the ncr option:

\[
ncr = \{ \text{non}_\text{ascii}|\text{non}_\text{server}|\text{no} \}
\]

At runtime, the ncr option is applied if:

- The result datatype is a string or Unicode datatype, not numeric or datetime or money, for instance.
- The XPath query does not specify `text()`.

The default ncr option is:

- If the returns datatype is a Unicode datatype, the default value is `ncr=no`.
- If the returns datatype is a string datatype, the default value is `ncr=non_server`.

Sort ordering in xmlextract

Sort ordering in xmlextract is discussed in “Sort ordering in XML Services” on page 100.

xmlextract uses the parsed XML sort order stored in the input XML document, not the current default sort order in the server.

Sort ordering in XML Services

**sp_configure option**  XML Services defines the `sp_configure` option default xml sort order, which has three distinguishing characteristics:

- It is static; you must restart Adaptive Server to execute this configuration.
- The option value is the name of a Unicode sort order. For details see the table “Default Unicode sort order,” in the *System Administration Guide, Volume 1*.
- The default option value is `binary`.

**xmlparse**  `xmlparse` returns a parsed representation of the argument document, including an index of the document’s elements and attributes and their values. The parsed representation specifies the default xml sort order as it exists when the document is parsed.
xmlextract  xmlextract evaluates XPath queries that compare terms, such as “//book[author='John Doe’]”. xmlextract compares the current *default xml sort order* with the document’s *parsed xml sort order*. If they are different, xmlextract raises an exception.

xmlextract uses the XML sort order stored in the input XML document, not the current default sort order in the server.

**Note**  XML Services uses a single default order, the *default xml sort order*. It does not use *both* default Unicode xml sort order and default xml sort order.

**Modifying the default xml sort order**  You can modify the *default xml sort order* with `sp_configure`.


```
update xmldocs
set doc = xmlparse(xmlextract('/', doc))
```

### I18n in xmlvalidate

xmlvalidate() supports Unicode datatypes, unichar, univarchar, unitext, and java.lang.String, as well as string and image datatypes. The returns clause of xmlvalidate can specify a Unicode datatype as the datatype of the extracted value.

### NCR option

xmlvalidate() supports the ncr option:

```
ncr={non_ascii | non_server | no}
```

At runtime, the ncr option applies only if the datatype of the result clause is a string or Unicode datatype. For example, the option does not apply to numeric, datetime, or money datatypes.

**Default NCR option**

- The default NCR option value is ncr=no if the returns datatype is a Unicode datatype: unichar, univarchar, unitext, or java.lang.String.
The default NCR option value is `ncr=non_server` if the returns datatype is a string datatype: `char`, `varchar`, or `text`. 
CHAPTER 7

xmltable()

This chapter describes the xmltable() function in detail.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>103</td>
</tr>
<tr>
<td>xmltable and derived table syntax</td>
<td>103</td>
</tr>
</tbody>
</table>

Introduction

xmltable() extracts a sequence of multi-valued elements from an XML document, and assembles a SQL table of those elements. A single call to xmltable() replaces a T-SQL loop performing multiple calls to xmlextax on each iteration. This function is invoked as a derived table (a parenthesized subquery specified in the from clause of another SQL query). Calling xmltable() is equivalent to executing a single xmlextax expression for each row of the table generated by xmltable().

xmltable() is a generalization of xmlextax. Both functions return data extracted from an XML document that is an argument in the function. The differences are:

- xmlextax returns the data identified by a single XPath query.
- xmltable() extracts the sequence, or row pattern, of the data identified by an XPath query, and extracts from each element of that sequence the data identified by a list of other XPath queries, the column patterns. It returns all the data in a SQL table.

xmltable and derived table syntax

These syntax sections show the basic syntax of xmltable() and where and how to use xmltable().
xmltable

Description
Extracts data from an XML document and returns it as a SQL table.

Syntax
xmltable_expression ::= xmltable
(row_pattern passing xml_argument
columns column_definitions
options_parameter)
row_pattern ::= character_string_literal
xml_argument ::= xml_expression
column_definitions ::= column_definition [ { , column_definition } ]
ordinality_column | regular_column
ordinality_column ::= column_name datatype for ordinality
regular_column ::= column_name datatype [ default literal ] [ null | not null ]
[ path column_pattern ]
column_pattern ::= character_string_literal
options_parameter ::=[,] option option_string
options_string ::= basic_string_expression

Derived table syntax
Returns a SQL table from within a SQL from clause.

from_clause ::= from table_reference [ , table_reference ] ...
table_reference ::= table_view_name ANSI_join derived_table
table_view_name ::= See the select command in Reference Manual Volume 2, "Commands".
ANSI_join ::= See the select command in Reference Manual Volume 2, "Commands".
derived_table ::= (subquery) as table_name [ (column_name [ , column_name ] ...) ]
xmltable_expression as table_name

Examples
xmltable as derived tables
This example shows a simple xmltable() call, returning a derived table.

```
select * from xmltable('/doc/item'
passing '<doc><item><id>1</id><name>Box</name></item>'
  + '<item><id>2</id><name>Jar</name></item></doc>'
columns id int path 'id', name varchar(20) path 'name') as items_table
id  name
-----  -----
1       Box
2       Jar
(2 rows affected)
```

Example 1 The syntax of derived tables requires you to specify a table name (items_table), even if you do not reference it. For example, this example is incorrect.
select * from xmltable ('/doc/item'  
  passing '<doc><item><id>1</id><name>Box</name></item>  
  + '<item><id>2</id><name>Jar</name></item></doc>'  
  columns id int path 'id', name varchar(20) path 'name')
-----------------
Msg 102 Level 15, State 1:  
Incorrect syntax near ')

Simple document reference examples  In document references, the 
argument following passing is the input XML document. In this example the 
document is specified as a character string literal.

elect * from xmltable ('/doc/item'  
  passing '<doc><item><id>1</id><name>Box</name></item>  
  + '<item><id>2</id><name>Jar</name></item></doc>'  
  columns id int path 'id', name varchar(20) path 'name') as items_table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

(2 rows affected)

Example 2  This example shows storing the document in a T-SQL variable, 
and referencing that variable in the xmltable() call.

declare @doc varchar(16384)
set @doc=''<doc><item><id>1</id><name>Box</name></item>  
  + '<item><id>2</id><name>Jar</name></item></doc>'

select * from xmltable('/doc/item' passing @doc  
  columns id int path 'id', name varchar(20) path 'name') as items_table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

(2 rows affected)

Example 3  To store the document in a table and reference it with a scalar 
subquery:

select 100 as doc_id,  
'='<doc><item><id>1</id><name>Box</name></item><item><id>2</id>  
  <name>Jar</name></item></doc>' as doc  
into #sample_docs  
select* from xmltable('/doc/item'  
  passing '<doc><item><id>1</id><name>Box</name></item>  
  + '<item><id>2</id><name>Jar</name></item></doc>'  
  columns id int path 'id', name varchar(20) path 'name') as items_table  

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>
XML table

    passing(select doc from #sample_docs where doc_id=100)
    columns id int path 'id', name varchar(20) path 'name' as items_table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

(2 rows affected)

**Row patterns**  The first argument in the `xmltable` call, the *row-pattern* (`'/doc/item'`) is an XPath query expression whose result is a sequence of elements from the specified document. The `xmltable` call returns a table with one row for each element in the sequence.

**Example 4** If the row pattern returns an empty sequence, the result is an empty table:

    select * from xmltable ('//item_entry'
    passing '<doc><item><id>1</id><name>Box</name></item>
    +'<item><id>2</id><name>Jar</name></item></doc>'
    columns id int path 'id',
    name varchar(20) path 'name') as items_table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
</table>

(0 rows affected)

**Example 5** The row pattern expression cannot be an XPath function:

    select * from xmltable ('/doc/item/tolower()'
    passing '<doc><item><id>1</id><name>Box</name></item>
    +'<item><id>2</id><name>Jar</name></item></doc>'
    columns id int path 'id',
    name varchar(20) path 'name') as items_table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
</table>

Msg 14825, Level 16, State 0:
Line 1:
XPath function call must be at leaf level.
Column patterns  The arguments following the columns keyword is the list of column definitions. Each column definition specifies a column name and datatype, as in create table, and a path, called the column pattern. The column-pattern is an XPath query expression that applies to an element of the sequence returned by the row-pattern, to extract the data for a column of the result table.

Example 6  When the data for a column is contained in an XML attribute, specify the column pattern using "@" to reference an attribute. For example:

```sql
select * from xmltable ('/doc/item'
                          passing '<doc><item id="1"><name>Box</name></item>'
                                   + '<item id="2"><name>Jar</name></item></doc>'
                          columns id int path '@id', name varchar(20)) as items_table
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

(2 rows affected)

Default column patterns  A column-pattern is commonly the same as the specified column name, for example name. In this case, omitting the column-pattern results in defaulting to the column_name:

```sql
select * from xmltable ('/doc/item'
                          passing '<doc><item><id>1</id><name>Box</name></item>'
                                   + '<item><id>2</id><name>Jar</name></item></doc>'
                          columns id int, name varchar(20)) as items_table
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

(2 rows affected)

Example 7  If you want a column pattern to default to the column name, in a column whose value is in an XML attribute, use a quoted identifier. You must then quote such identifiers when you reference them in the results:

```sql
set quoted_identifier on
select "@id", name from xmltable ('/doc/item'
                          passing '<doc><item id="1"><name>Box</name></item>'
                                   + '<item id="2"><name>Jar</name></item></doc>'
                          columns "@id" int, name varchar(20)) as items_table
```

<table>
<thead>
<tr>
<th>@id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 8  You can also use quoted identifiers to specify column names as default column patterns, using column names that are more complex XPath expressions. For example:

```sql
set quoted_identifier on
select "@id", "name/short", "name/full" from xmltable ('/doc/item'
    passing '<doc><item id="1"><name><short>Box</short>
        <full>Box, packing, moisture resistant, plain</full>
    </name></item>'
    +'<item id="2"><name><short>Jar</short>
        <full>Jar, lidded, heavy duty</full>
    </name></item></doc>'
columns "@id" int, "name/short" varchar(20), "name/full" varchar(50))
as items_table
```

<table>
<thead>
<tr>
<th>@id</th>
<th>name/short</th>
<th>name/full</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
<td>Box, packing, moisture resistant, plain</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
<td>Jar, lidded, heavy duty</td>
</tr>
</tbody>
</table>

(2 rows affected)

Implicit text()  This example demonstrates the function text(), which is generally implicit in the column pattern. text() removes XML element tags. For example, this XPath query returns the selected element with the XML markup:

```sql
1> declare @doc varchar(16384)
2> set @doc= '<doc><item><id>1</id><name>Box</name></item>'
   +'<item><id>2</id><name>Jar</name></item></doc>'
3> select xmlextract('/doc/item[2]/name', @doc)
```

```
<name>Jar</name>
```

Example 9

Adding text() to the XPath query removes the XML tags:

```sql
1> declare @doc varchar(16384)
2> set @doc= '<doc><item><id>1</id><name>Box</name></item>'
   +'<item><id>2</id><name>Jar</name></item></doc>'
```
3> select xmlextact('*/doc/item[2]/name/text()', @doc)

------------
Jar

**Example 10**  *text() is implicit in most column patterns. This example does not specify text() in the column pattern for either the id or name column:*

```sql
select * from xmltable ('/doc/item' passing '<doc><item><id>1</id><name>Box</name></item>' + '<item><id>2</id><name>Jar</name></item></doc>'
columns id int path 'id', name varchar(20) path 'name') as items_table
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

(2 rows affected)

**Datatype conversion**  *You can derive column values in datatype conversions by applying an implicit SQL convert statement to the data extracted from the column pattern. For example:*

```sql
select * from xmltable ('/emps/emp' passing '<emps>
<emp><id>1</id><salary>123.45</salary><hired>1/2/2003</hired></emp>
+<emp><id>2</id><salary>234.56</salary><hired>2/3/2004</hired></emp>'
columns id int path 'id', salary dec(5,2), hired date) as items_table
```

<table>
<thead>
<tr>
<th>id</th>
<th>salary</th>
<th>hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123.45</td>
<td>Jan 2, 2003</td>
</tr>
<tr>
<td>2</td>
<td>234.56</td>
<td>Feb 3, 2004</td>
</tr>
</tbody>
</table>

(2 rows affected)

**Example 11**  *The extracted XML data for the column must be convertible to the column datatype, or an exception is raised:*

```sql
select * from xmltable ('/emps/emp' passing '<emps>
+<emp><id>1</id><salary>123.45</salary><hired>1/2/2003</hired></emp>
+<emp><id>2</id><salary>234.56 C$</salary><hired>2/3/2004</hired></emp>'
+)
xmltable

columns id int path 'id', salary dec(5,2), hired date)  
as items_table
----------------------

Msg 14841, Level 16, State 3:  
Line 1:  
XMLTABLE:Failed to convert column pattern result to DECML for column 1.

**Example 12** To handle XML data whose format is not suitable for a SQL convert function, extract the data to a string column (varchar, text, image, java.lang.String)

```sql
select * from xmltable ('/emps/emp'
  passing '<emps>
    +<emp><id>1</id><salary>123.45</salary><hired>1/2/2003</hired></emp'
    +<emp><id>2</id><salary>234.56 </salary><hired>2/3/2004</hired></emp>'
  +</emps>'
columns id int, salary varchar(20), hired date)  
as items_table

id     salary    hired
-------    --------    --------
  1       123.45     Jan 2, 2003
  2       234.56     Feb 3, 2004
(2 rows affected)
```

**Ordinality columns** The order of elements in XML documents can be significant.

Elements are sometimes ordered by the value of contained elements. In this example, the `<item>` elements are ordered by the value of the contained `<id>` elements.

```xml
<doc>
  <item><id>1</id><name>Box</name></item>'
  <item><id>2</id><name>Jar</name></item>'
</doc>
```

You can also order elements in an arbitrary but significant manner. In the following example, the order of the `<item>` elements is based on no values, but may reflect a priority ordering: first in, first out. Such an ordering can be significant for the application of the data.

```xml
<doc>
  <item><id>25</id><name>Box</name></item>'
  <item><id>15</id><name>Jar</name></item>'
</doc>
```

**Example 13** You can use an ordinality_column in xmltable to record the ordering of elements in the input XML document:
declaredoc varchar(16384)
set @doc = '<doc><item><id>25</id><name>Box</name></item>'
+ '<item><id>15</id><name>Jar</name></item></doc>
select * from xmltable('/doc/item' passing @doc
  columns item_order int for ordinality,
        id int path 'id',
        name varchar(20) path 'name') as items_table
order by item_order
---
1 25  Box
2 15  Jar
(2 rows affected)

Without the for ordinality clause and the item_order column, there is nothing in
the id and name columns that indicates that the row of id 25 precedes the row
of id 15. The for ordinality clause provides a way to make sure that the ordering
of the output SQL rows is the same as the ordering of the elements in the input
XML document.

The datatype of an ordinality column can be any fixed numeric datatype: int,
tinyint, bigint, numeric, or decimal. numeric and decimal must have a scale of 0.
An ordinality column cannot be real or float.

**Null values** If a column pattern returns an empty result, the action taken
depends on the default and {null | not null} clauses.

**Example 14** This example omits the <name> element from the second <item>.
The name column allows names by default.

```sql
select * from xmltable ('//item' passing '<doc><item><id>1</id><name>Box</name></item>'
  + '<item><id>2</id></item></doc>'
columns id int path 'id', name varchar(20), path 'name')
as items_table
------------------------
id    name
---    ----
1    Box
2    NULL
(2 rows affected)
```

**Example 15** This example omits the <name> element from the second <item>,
and specifies not null for the name column:
Adaptive Server Enterprise

```
select * from xmltable ('/doc/item'
    passing '<doc><item><id>1</id><name>Box</name></item>'
    + '<item><id>2</id><name>Jar</name></item></doc>'
columns id int path 'id', name varchar(20) path 'name'
) as items_table
```

---

**Msg 14847, Level 16, State 1:**

Line 1:

XMLTABLE column 0, does not allow null values.

---

**Example 16** This example adds a default clause to the name column, and omits the `<name>` elements from the second `<item>`.

```
select * from xmltable ('/doc/item'
    passing '<doc><item><id>1</id><name>Box</name></item>'
    + '<item><id>2</id></item></doc>'
columns id int path 'id' name varchar(20) default '***' path 'name'
) as items_table
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>***</td>
</tr>
</tbody>
</table>

(2 rows affected)

---

**Context of xmltable calls**  These examples show SQL commands in which you can use an xmltable call in a derived table expression.

**Example 17** `select` – You can use `xmltable()` in a simple `select` statement:

```
select * from xmltable ('/doc/item'
    passing '<doc><item><id>1</id><name>Box</name></item>'
    + '<item><id>2</id><name>Jar</name></item></doc>'
columns id int path 'id'
    name varchar(20) path 'name') as items_table
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

(2 rows affected)

---

**Example 18** `View definition` – Specify `select` using `xmltable` in a view definition. This example stores a document in a table and references that stored document in a `create view` statement, using `xmltable` to extract data from the table:

```
select 100 as doc_id,
    '<doc><item><id>1</id><name>Box</name></item>'
```

---
create view items_table as
select * from xmltable('/doc/item'
    passing (select doc from sample_docs where id=100)
    columns id int path 'id'
    name varchar(20) path 'name')as xml_extract

id  name
----------------
1   Box
2   Jar
(2 rows affected)

Example 19 Cursor declaration – Specify select using xmltable to declare a
cursor:

declare cursor C for
select * from xmltable ('/doc/item'
    passing (select doc from sample_docs where id=100)
    columns id int path 'id'
    name varchar(20) path 'name')as items_table
declare @idvar int
declare @namevar varchar(20)
open C
while @@sqlstatus=0
begin
    fetch C into @idvar, @namevar
    print 'ID "%1!" NAME "%2!"', @idvar, @namevar
end
-------------------------
ID "1" NAME "Box"
ID "2" NAME "Jar"
(2 rows affected)

In applications that require multiple actions for each generated row, such as
executing update, insert, or delete from other tables, based on the data in each
generated row, process an xmltable result with a cursor loop. Alternatively, you
can store the xmltable result in a temporary table and process that table with a
cursor loop.

Example 20 select into – Specify select using xmltable in select into:

select * into #extracted_table
from xmltable('/doc/item'
    passing (select doc from sample_docs where doc_id=100
    columns id int path 'id'

xmltable

    name varchar(20) path 'name') as items_table

select * from #extracted_table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

**Example 21** insert – Specify select using xmltable in an insert command:

create table #extracted_data (idcol int, namecol varchar(20))
insert into #extracted_data
select * into #extracted_table from xmltable('/doc/item'
    passing (select doc from sample_docs where doc_id=100
    columns id int path 'id', name varchar(20) path 'name')as items_table
select * from extracted_data

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Box</td>
</tr>
<tr>
<td>2</td>
<td>Jar</td>
</tr>
</tbody>
</table>

(2 rows affected)

**Example 22** Scalar subquery – Specify select using xmltable in a scalar subquery. xmltable returns a SQL table, so the scalar subquery must perform either an aggregation or a selection to return a single row and column for the scalar subquery result.

declare @idvar int
set @idvar = 2
select @idvar,
    (select name from xmltable ('/doc/item'
        passing (select doc from sample_docs where doc_id=100
        columns id int path 'id', name varchar(20) path 'name') as item_table
        where items_table.id=@idvar)

2  Jar
(1 rows affected)

**Example 23** Joins – Join an xmltable result with other tables, using either comma-list joins or outer joins:

create table prices (id int, price decimal (5,2))
insert into prices values(1,123.45)
insert into prices values (2,234.56)
select prices.id,extracted_table.name, prices.price
from prices,(select * from xmltable('/doc/item'
XML Services

**Processing tables of documents**  You can apply `xmltable()` to the XML document in each row of a table of XML documents. For example, the next example creates a table containing two columns:

- The `pub_id` of one of the three publishers in the `pubs2_publishers` table.
- An XML document containing the title and price of each document published by that publisher. To reduce the size of the example table only titles whose price is greater than $15.00 are included:

    ```sql
    create table high_priced_titles
        (pub_id char(4), titles varchar (1000))
    insert into high_priced_titles
    select p.pub_id, 
        (select title_id, price from pubs2..titles t,pubs2..publishers p
            where price> 15 and t.pub_id=p.pub_id
            for xml
                option 'tablename=expensive_titles, rowname=title')
                returns varchar(1000))as titles
    from pubs2..publishers p
    select * from high_priced_titles
    ---------------
    pub_id | titles
    ------ | ------------------------------------------
    0736   | <expensive_titles>
            | <title> <title_id>PS3333</title_id> <price>19.99</price></title>
            | </expensive_titles>
    0877   | <expensive_titles>
            | <title> <title_id>MC2222</title_id> <price>19.99</price></title>
            | <title> <title_id>PS1372</title_id> <price>21.59</price></title>
            | <title> <title_id>TC3218</title_id> <price>20.95</price></title>
            | </expensive_titles>
    01389  | <expensive_titles>
            | <title> <title_id>BU1032</title_id> <price>19.99</price></title>
            | <title> <title_id>BU7832</title_id> <price>19.99</price></title>
```
Example 24 Use xmltable in a scalar subquery to process the XML document in each row, as a SQL table. For example, list the maximum title price for each publisher:

```
select pub_id
(select max(price)
from xmltable('//title' passing hpt.titles
columns title_id char(4), price money)
as extracted_titles, high_priced_titles hpt) as max_price
from high_priced_titles hpt
```

<table>
<thead>
<tr>
<th>pubid</th>
<th>max_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>19.99</td>
</tr>
<tr>
<td>0877</td>
<td>21.59</td>
</tr>
<tr>
<td>1389</td>
<td>22.95</td>
</tr>
</tbody>
</table>

This high_priced_titles table is essentially hierarchic: each row is an intermediate node, which contains, in its title column, a leaf node for each title element in the XML document. high_priced_titles has three rows.

You can flatten that hierarchy, producing a table with a row for each title element. To flatten the data in the titles column and produce a table, high_priced_titles_flattened, which has eight rows (one for each of the titles/title elements), use one of the following solutions.

Solution 1 You can produce high_priced_titles_flattened by using a loop that processes high_priced_titles, and applies xmltable to the titles document in each row. In the example below, notice the from clause:

```
from(select @pub_id_var)as ppp,
    xmltable('//title' passing @titles_var
    columns title_id char(6), price money)as ttt
```

The variables @pub_id_var and @titles_var are the pub_id and titles columns from the current row of high_priced_titles. The from clause joins two derived tables:

- (select @pub_id_var) as ppp

  This is a table with one row and one column, containing the pub_id.
XML Services

• `xmltable(...)` as `ttt`

  This generates a table with a row for each *title* element in the *titles* document of the current *high_priced_titles* row.

  To flatten the hierarchy, join these two derived tables, which appends the `pub_id` column to each row generated from the `titles` column:

  ```sql
  create table high_priced_titles_flattened_1
  (pub_id char(4), title_id(char(6), price money)
  
  declare C cursor for select * from high_priced_titles
  declare @pub_id_var char(4)
  declare @titles_var char(1000)
  open C

  while @@sqlstatus =0
  begin
  fetch C into @pub_id_var, @titles_var
  
  insert into high_priced_titles_flattened_1
  select *
  from (select @pub_id_var) as ppp,(col1),
  xmltable('//title' passing @titles_var
  columns title_id char (6), price money) as ttt
  end

  select * from high_priced_titles_flattened_1
  
<table>
<thead>
<tr>
<th>pub_id</th>
<th>title_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>PS3333</td>
<td>19.99</td>
</tr>
<tr>
<td>0877</td>
<td>MC2222</td>
<td>19.99</td>
</tr>
<tr>
<td>0877</td>
<td>PS1372</td>
<td>21.59</td>
</tr>
<tr>
<td>0877</td>
<td>TC3218</td>
<td>20.95</td>
</tr>
<tr>
<td>1389</td>
<td>BU1032</td>
<td>20.95</td>
</tr>
<tr>
<td>1389</td>
<td>BU7832</td>
<td>19.99</td>
</tr>
<tr>
<td>1389</td>
<td>PC1035</td>
<td>19.99</td>
</tr>
<tr>
<td>1389</td>
<td>PC8888</td>
<td>20.00</td>
</tr>
</tbody>
</table>
  
  Solution 2  You can also generate the *high_priced_titles* table using a special join.

  This example joins two tables: *high_priced_titles* as `hpt`, and the table generated by `xmltable`. The passing argument of `xmltable` references the preceding `hpt` table. Normally, it is illegal to reference a table in a `from` clause, in a derived table expression within the same `from` clause. However, `xmltable` is allowed to reference other tables in the same `from` clause, as long as these tables precede the `xmltable` call in the same `from` clause.
xmltable

```sql
select hpt.pub_id, extracted_titles.*
into high_priced_titles_flattened_3
from high_priced_titles as hpt,
     xmltable('//title'
          passing htp.titles,
          columns
          title_id char(6)
          price money)as extracted_titles
```

<table>
<thead>
<tr>
<th>pub_id</th>
<th>title_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>PS3333</td>
<td>19.99</td>
</tr>
<tr>
<td>0877</td>
<td>MC2222</td>
<td>19.99</td>
</tr>
<tr>
<td>0877</td>
<td>PS1372</td>
<td>21.59</td>
</tr>
<tr>
<td>0877</td>
<td>TC3218</td>
<td>20.95</td>
</tr>
<tr>
<td>1389</td>
<td>BU1032</td>
<td>20.95</td>
</tr>
<tr>
<td>1389</td>
<td>BU7832</td>
<td>19.99</td>
</tr>
<tr>
<td>1389</td>
<td>PC1035</td>
<td>19.99</td>
</tr>
<tr>
<td>1389</td>
<td>PC8888</td>
<td>20.00</td>
</tr>
</tbody>
</table>

Usage

- `xmltable` is a built-in, table-valued function.
- The result type of an `xmltable` expression is a SQL table, whose column names and their datatypes are specified by `column_definitions`.
- These keywords are associated with `xmltable`:
  - Reserved: for, option
  - Not reserved: columns, ordinality, passing, path, `xmltable`
- The expressions in the arguments of an `xmltable` call can reference the column names of preceding tables in the `from` clause containing the `xmltable` call. Only tables that precede the `xmltable` call can be referenced. Such a reference, to a column of a preceding table in the same `from` clause, is called a lateral reference. For example:

```sql
select * from T1, xmltable(...passing T1.C1...) as XT2, xmltable(...passing XT2.C2...)as XT3
```

- You cannot use `xmltable` in the `from` clause of an update or delete statement. For example, the following statement fails:

```sql
update T set T.C=...
```
from T,xmltable(...)  
where...

- You cannot update the SQL table returned by an xmltable expression.
- Datatypes in regular_columns can be any SQL datatype.
- The literal following a default in a regular_column must be assignable to the datatype of the column.
- There can be no more than one ordinality_column; the datatype specified for this variable must be integer, smallint, tiny int, decimal, or numeric. decimal and numeric must have a scale of zero.
- An ordinality_column, if one exists, is not nullable. The nullable property of other columns is specified by the \{null | not null\} clause. The default is null.

**Note** This default is different from the default value of create table.

- The current setting of set quoted_identifier applies to the clauses of an xmltable expression. For example,
  - If set quoted_identifier is on, column names can be quoted identifiers, and string literals in row_pattern, column_pattern, and default literals must be surrounded with single quotation marks.
  - If set quoted_identifier is off, column names cannot be quoted identifiers, and string literals in row_pattern, column_pattern, and default literals can be surrounded with either single or double quotation marks.
- The general format of the option_string is described in the section “option_strings: general format.”.

**xmltable row and column patterns** xmltable row and column patterns are allowed to be only simple paths. Simple paths in XPath consist only of forward traversals using '/' and element/attribute names.

- If the row_pattern does not begin at the root level of the document specified by xml_argument, an exception is raised. The row pattern must begin at the root of the XML document.
- If the row_pattern specifies an XML function, an exception is raised. The row pattern must not specify an XML function.
If a column_definition does not specify a path, then the default column_pattern is the column_name of the column definition. This default is subject to the case sensitivity of the server. For example, consider this statement:

```
select * from xmltable(...columns name varchar(30),...)
```

If the server is case-insensitive, this is equivalent to the following:

```
select * from xmltable(...columns name varchar(30) path 'name',...)
```

If the server is case sensitive, the first statement is equivalent to:

```
select * from xmltable (...columns name varchar(30)path 'NAME',...)
```

Generating the rows of the result table

The result value of an xmltable expression is a T-SQL table RT, defined as follows:

- RT has a row for each element in the XML sequence that results from applying the row_pattern to the xml_argument.
- The rows of RT have a column for each column_definition, with the column_name and datatype specified in the column_definition.
- If a column_definition is a ordinality_column, its value for the Nth row is the integer N.
- If a column_definition is a regular_column, its value for the Nth row corresponds to the following:
  - Let XVAL be the result of applying this XPath expression to the xml_argument:
    
    \[
    (\text{row_pattern}[N] / \text{column_pattern}/\text{text})
    \]
  - If XVAL is empty, and the column_definition contains a default clause, the value of the column is that default value.
    
    If XVAL is empty and the column_definition specifies not null, an exception is raised.

    Otherwise, the value of the column is the null value.
  - If XVAL is not empty, and the datatype of the column is char, varchar, text, unitext, unichar, univarchar, or java.lang.String, de-entitize XVAL.
  - The value of the column is the result of:
convert (datatype, XVAL)

See also
For a sample application using xmltable, see Appendix D, “Sample Application for xmltable,” in XML Services.
The sample_docs Example Table

The descriptions of the XML query functions reference an example table named sample_docs. This chapter shows you how to create and populate that table.

The sample_docs table has three columns and three rows.

sample_docs table columns and rows

This section shows the structure of the sample_docs table.

Sample_docs table columns

The sample_docs table has three columns:

- name_doc
- text_doc
- image_doc

In a specified example document, name_doc specifies an identifying name, text_doc specifies the document in a text representation, and image_doc specifies the document in a parsed XML presentation stored in an image column. The following script creates the table:

```
create table sample_docs
    (name_doc varchar(100),
     text_doc text null,
     image_doc image null)
```
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
            <first-name>Mary</first-name>
            <last-name>Bob</last-name>
        </author>
        <price>55</price>
    </book>
<?PI_sample Process Instruction ?>
<!--sample comment--> 
<magazine style='glossy' frequency='monthly'>
    <title>Tracking Trenton</title>
```

```sql```
APPENDIX A   The sample_docs Example Table

<price>2.50</price>
<subscription price='24' per='year'/>
</magazine>
<book style='novel' id='myfave'>
<title>Trenton Today, Trenton Tomorrow</title>
<author>
  <first-name>Toni</first-name>
  <last-name>Bob</last-name>
  <degree from='Trenton U'>B.A.</degree>
  <degree from='Harvard'>Ph.D.</degree>
  <award>Pulizer</award>
  <publication>Still in Trenton</publication>
  <publication>Trenton Forever</publication>
</author>
<price intl='canada' exchange='0.7'>6.50</price>
<excerpt>
<p>It was a dark and stormy night.</p>
<p>But then all nights in Trenton seem dark and stormy to someone who has gone through what <emph>I</emph> have.</p>
<definition-list>
  <term>Trenton</term>
  <definition>misery</definition>
</definition-list>
</excerpt>
</book>

<book style='leather' price='29.50' xmlns:my='http://www.placeholdernamehere.com/schema/'>
<title>Who's Who in Trenton</title>
<author>Robert Bob</author>
</book>

</bookstore>

sample_docs tables

The other two rows of the sample_docs table are XML representations of the publishers and titles tables of the pubs2 database. The pubs2 database is an database of example tables that is described in the Transact-SQL User's Guide.
The publishers and titles tables are two of the tables in this sample database. To shorten the example, the XML representation of the titles table includes only selected columns.

**Table script (for publishers table)**

These two insert statements add a row for the publishers table and a row for the authors table to the the sample_docs table. Each row contains a column that identifies the row (‘publishers’, ‘titles’), and a text_doc column. Call the select command with the for xml option to generate the XML document:

```sql
insert into sample_docs (name_doc, text_doc)
select 'publishers',
(select * from publishers for xml)
insert into sample_docs (name_doc, text_doc)
select 'titles', (select title_id, title, type, pub_id,
price, advance, total_sales
from titles for xml)
```

**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 125.

```xml
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>
```

---

Adaptive Server Enterprise
APPENDIX A  The sample_docs Example Table

<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'

<table>
<thead>
<tr>
<th>text_doc</th>
</tr>
</thead>
</table>
| <titles xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <row>
      <title_id>BU1032</title_id>
      <title>The Busy Executive’s Data Base Guide</title>
      <type>business</type>
      <pub_id>1389</pub_id>
      <price>19.99</price>
      <advance>5000.00</advance>
      <total_sales>4095</total_sales>
   </row>

</titles> |
<row>
  <title_id>BU1111</title_id>
  <title>Cooking with Computers: Surreptitious Balance Sheets</title>
  <type>business</type>
  <pub_id>1389</pub_id>
  <price>11.95</price>
  <advance>5000.00</advance>
  <total_sales>3876</total_sales>
</row>

<row>
  <title_id>BU2075</title_id>
  <title>You Can Combat Computer Stress!</title>
  <type>business</type>
  <pub_id>0736</pub_id>
  <price>2.99</price>
  <advance>10125.00</advance>
  <total_sales>18722</total_sales>
</row>

<row>
  <title_id>BU7832</title_id>
  <title>Straight Talk About Computers</title>
  <type>business</type>
  <pub_id>1389</pub_id>
  <price>19.99</price>
  <advance>5000.00</advance>
  <total_sales>4095</total_sales>
</row>

<row>
  <title_id>MC2222</title_id>
  <title>Silicon Valley Gastronomic Treats</title>
  <type>mod_cook</type>
  <pub_id>0877</pub_id>
  <price>19.99</price>
  <advance>0</advance>
  <total_sales>2032</total_sales>
</row>

<row>
  <title_id>MC3021</title_id>
  <title>The Gourmet Microwave</title>
  <type>mod_cook</type>
  <pub_id>0877</pub_id>
</row>
<table>
<thead>
<tr>
<th>Title ID</th>
<th>Title</th>
<th>Type</th>
<th>Publisher ID</th>
<th>Price</th>
<th>Advance</th>
<th>Total Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC3026</td>
<td>The Psychology of Computer Cooking</td>
<td>UNDECIDED</td>
<td>0877</td>
<td>2.99</td>
<td>15000.00</td>
<td>22246</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>PC9999</td>
<td>Net Etiquette</td>
<td>popular_comp</td>
<td>1389</td>
<td>21.59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PS1372</td>
<td>Computer Phobic and Non-Phobic</td>
<td>psychology</td>
<td>0877</td>
<td>21.59</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
<row>
<title_id>PS2091</title_id>
<title>Is Anger the Enemy?</title>
<type>psychology</type>
<pub_id>0736</pub_id>
<price>10.95</price>
<advance>2275.00</advance>
<total_sales>2045</total_sales>
</row>

<row>
<title_id>PS2106</title_id>
<title>Life Without Fear</title>
<type>psychology</type>
<pub_id>0736</pub_id>
<price>7.99</price>
<advance>6000.00</advance>
<total_sales>111</total_sales>
</row>

<row>
<title_id>PS3333</title_id>
<title>Prolonged Data Deprivation: Four Case Studies</title>
<type>psychology</type>
<pub_id>0736</pub_id>
<price>19.99</price>
<advance>2000.00</advance>
<total_sales>4072</total_sales>
</row>

<row>
<title_id>PS7777</title_id>
<title>Emotional Security: A New Algorithm</title>
<type>psychology</type>
<pub_id>0736</pub_id>
<price>7.99</price>
<advance>4000.00</advance>
<total_sales>3336</total_sales>
</row>
APPENDIX A  The sample_docs Example Table

<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)
The Adaptive Server External File System Access feature provides access to operating system files as SQL tables. This appendix describes the use of the native XML processor with the File System Access Feature. For more detailed information, see the Adaptive Server Component Integration Services User’s Guide.

When you use the File System Access feature, you create a proxy table that maps an entire directory tree from the external file system, using Adaptive Server’s Component Integration Services (CIS) feature. Then you use the built-in functions of the native XML processor on the data in the proxy table to query XML documents stored in the external file system.

With External Directory Recursive Access, you can map a proxy table to a parent directory, and to all its subordinate files and subdirectories.

Getting Started

This section explains how to set up XML Services with External File System Access capabilities.

Enabling XML services and External File System Access

- Enable XML Services, CIS, and file access, using `sp_configure`:
  ```sql
  sp_configure "enable xml", 1
  ```
- Verify that the configuration parameter `enable cis` is set to 1:
  ```sql
  sp_configure "enable cis",1
  ```
- Enable file access using `sp_configure`:
sp_configure "enable file access", 1

Character set conversions with external file systems

In general, the content columns of external files system tables are treated as image. However, special conversions are performed when a content column is assigned to a Unicode column, i.e. a column of datatype unichar, univarchar, unitext, or java.lang.String. Such assignment of a content column to a Unicode column occurs in the following contexts:

- An insert command used to insert a Unicode column from a subquery that references a content column.
- An update command used to update a Unicode column with a new value that references a content column.
- A convert function call that specifies both a target Unicode datatype and a source value that is a content column.

In assigning a content column to Unicode, use these rules:

- If the source document has a BOM (Byte Order Mark), to convert the source document the BOM must indicate UTF-8 or UTF-16. If the BOM indicates UCS-4, an error is raised. UCS-4 is not supported.
- If the source document has an XML header that includes an encoding clause, but no BOM, the encoding clause must specify the server character set or UTF-8 to convert the source data. An encoding clause that specifies a character set other than the server set or UTF-8 raises an error.
- If the source document has no XML header, a header with no encoding clause, and no BOM, the processor treats the character set as UTF-8, and converts the source data.
- If an error occurs during a conversion, an error is raised, but the statement continues.

Examples

The following examples show how you can use various XML built-ins to query XML documents in the external file system.
Setting up your XML documents and creating the proxy table

These examples use two XML documents stored in the files named *bookstore.1.xml* and *bookstore.2.xml*, that you create:

```
cat bookstore.1.xml
<?xml version='1.0' standalone = 'no'?>
<!-- bookstore.1.xml example document--!>
<bookstore specialty='novel'>
  <book style='autobiography'>
    <title>Seven Years in Trenton</title>
    <author>
      <first-name>Joe</first-name>
      <last-name>Bob</last-name>
      <award>Trenton Literary Review Honorable Mention</award>
    </author>
    <price>12</price>
  </book>
</bookstore>
```

```
cat bookstore.2.xml
<?xml version='1.0' standalone = 'no'?>
<!-- bookstore.2.xml example document--!>
<bookstore specialty='novel'>
  <book style='compbook'>
    <title>Modern Database Management</title>
    <author>
      <first-name>Jeffrey</first-name>
      <last-name>Hoffer</last-name>
    </author>
    <price>112.00</price>
  </book>
</bookstore>
```

You can reference these XML documents with File System Access, using `create proxy table`.

The following code sample shows the use of `create proxy table`. The directory pathname in the `at` clause must reference a file system directory that Adaptive Server can both see and search. If you add an `;R` (indicating "Recursion") extension to the end of the pathname CIS extracts file information from every directory subordinate to the pathname.

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

<table>
<thead>
<tr>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>bookstore.1.xml</td>
</tr>
<tr>
<td>&lt;title&gt;Seven Years in Trenton&lt;/title&gt;</td>
</tr>
<tr>
<td>bookstore.2.xml</td>
</tr>
<tr>
<td>&lt;title&gt;Modern Database Management&lt;/title&gt;</td>
</tr>
</tbody>
</table>

(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 ("/").

```sql
insert into xmldoctab select filename,xmlcol=xmlextract("/",content) into from xmlxfsTab
```

(2 rows affected)

In this example, the datatype of the xmlxfsTab.content column is image, and the default datatype returned by the `xmlextract` built-in function is text. Therefore, specify the returns image clause in the `xmlextract` call to return the result as an image value.
To preserve a header, use xmlvalidate() instead of xmlextract():

```sql
insert into xmldoctab select filename,xmlvalidate(content)
from xmlxfsTab
-----------
(2 rows affected)
```

The following will create a new subdirectory, XmlDir:

```sql
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:

```sql
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.

```sql
insert xmlxfsTab(filename, content)
select 'parsed'+t.filename,xmlparse(t.content) from xmlxfsTab
-----------
(2 rows affected)
```

The following code sample queries the parsed documents stored in the XFS file system.

```sql
select filename, xmlextract("//book/title", content)
from xmlxfsTab
where filename like 'parsed%' and filetype = 'REG'
```

```
filename
-----------------
```

XML Services
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 nonxmlldoc.txt files along with bookstore1.xml and bookstore2.xml files:

    select filename from xmlxfsTab
    filename
                    -----------------------------------------
    picture.jpg     bookstore.1.xml
    bookstore.2.xml  nonxmlldoc.txt

    (4 rows affected)

The following code sample shows an XML query on both XML and non-XML data:
APPENDIX B  XML Services and External File System Access

XML Services

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>

nonxmldoc.txt
<xml_parse_error>Invalid document structure</xml_parse_error>
```

(4 rows affected)
Example: parsing XML and non-XML documents with the 'xmlerror=message' option

This code sample specifies the 'xmlerror=message' option in the xmlparse call. This will store the parsed XML for XML documents that are valid XML, and store a parsed XML error message element for documents that are not valid XML.

```sql
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.

```sql
select filename, xmlextract('/xml_parse_error', content)
from xmlxfsTab
where '/xml_parse_error' xmltest content and filename like 'ParsedDir%'
----------------
```

Or with `xmlrepresentation` builtin

```sql
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 xmlxfsTab
filename
----------------------------
picture.jpg NULL
bookstore.1.xml <title>Seven Years in Trenton</title>
bookstore.2.xml <title>Modern Database Management</title>
nonxmlldoc.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'
filename
----------------------------
picture.jpg
nonxmlldoc.txt

(2 rows affected)
```
Migrating Between the Java-based XQL Processor and the Native XML Processor

Introduction

The Java-based XQL processor and the native XML processor both implement query languages and return documents in parsed form, but they use different functions and methods.

- The native XML processor implements XML query language. It provides a built-in function, `xmlparse`, that returns, in parsed form, a document suitable for efficient processing with the `xmlextract` and `xmltext` built-in functions.

- The Java-based XQL processor is an earlier facility that implements the XQL query language. It provides a Java method, `com.sybase.xml.xql.Xql.parse`, that returns a parsed form of a document that is a `sybase.aseutils.SybXmlStream` object, suitable for processing with the `com.sybase.xml.xql.Xql.query` method.

If you want to migrate documents between the Java-based XQL processor and the native XML processor, you should be aware of the following possibilities and restrictions:

- Documents in text form can be processed directly by both the Java-based XQL processor and the native XML processor.

- The `sybase.aseutils.SybXmlStream` documents generated by `com.sybase.xml.xql.Xql.parse` can only be processed by the Java-based XQL processor. They cannot be processed by the built-in functions `xmlextract` or `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.

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:

```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 form of 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.xmlxql.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 xmltext built-in functions, or you can update the xmlindexed column for the native XML processor, as follows:

```sql
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:
Migrating documents and queries

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.aseutilities.SybXmlStream form of the documents, generated by the
com.sybase.xml.xql.Xql.parse method. After the update statement, that column
contains the parsed form of the documents, suitable for processing with the
xmlextract and xmlparse methods.

Regenerating text documents from the native XML processor

This section demonstrates regenerating the text form of the documents from the
form generated for the native XML processor.

If the xmlindexed column contains data generated by the xmlparse function, you
can regenerate the text form of the document in the new xmlsource column with
the following SQL statement:

update xmltab
set xmlsource = xmlextract("/", xmlindexed)

You can then

• process the xmlsource column directly with the Java-based XQL
  processor, using com.sybase.xml.xql.Xql.query, OR
• update the xmlindexed column with the parsed form suitable for processing
  with the Java-based XQL processor, using the following statement:

update xmltab
set xmlindexed = com.sybase.xml.xql.Xql.parse(xmlsource)

If you don't want to add the xmlsource column, you can combine these steps, as
in the following SQL statement:

update xmltab
set xmlindexed = com.sybase.xml.xql.Xql.parse
  (xmlextract("/", xmlindexed))
Before this update statement is executed, the xmlindexed column contains the parsed form of the documents, generated by the xmlparse built-in function. After the update statement, that column contains the parsed form of the documents, generated by com.sybase.xml.xml.Xql.parse, suitable for processing with com.sybase.xml.xml.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.
Sample Application for
xmlns:table()
<emps>
  <emp_id>E345</emp_id>
  <emp_name>Don Davis</emp_name>
  <salary>945.67</salary>
  <phones>
    <phone><phone_no>650.555.5001</phone_no></phone>
  </phones>
</emp>
<emp>
  <emp_id>E345</emp_id>
  <emp_name>Earl Evans</emp_name>
  <phones>
    <phone><phone_no>650.555.5001</phone_no></phone>
  </phones>
</emp>
</emps>
</dept>
</dept>
<dept_id>D234</dept_id>
<dept_name>Auxiliary</dept_name>
<emps>
  <emp_id>E345</emp_id>
  <emp_name>Don Davis</emp_name>
  <salary>945.67</salary>
  <phones>
    <phone><phone_no>650.555.5001</phone_no></phone>
  </phones>
</emp>
<emp>
  <emp_id>E345</emp_id>
  <emp_name>Earl Evans</emp_name>
  <phones>
    <phone><phone_no>650.555.5001</phone_no></phone>
  </phones>
</emp>
</emps>
</dept>
</dept>

Sample table

<emps>
  <emp_id>E345</emp_id>
  <emp_name>Don Davis</emp_name>
  <salary>945.67</salary>
  <phones>
    <phone><phone_no>650.555.5001</phone_no></phone>
  </phones>
</emp>
<emp>
  <emp_id>E345</emp_id>
  <emp_name>Earl Evans</emp_name>
  <phones>
    <phone><phone_no>650.555.5001</phone_no></phone>
  </phones>
</emp>
</emps>

<emps_summary>
  <salary_summary>
    <max_salary>934.56</max_salary>
    <total_salary>2770.35</total_salary>
  </salary_summary>
</emps_summary>

<projects>
  <project>
    <project_id>PABC</project_id>
    <budget>598.65</budget>
  </project>
  <project>
    <project_id>PBDC</project_id>
    <budget>587.65</budget>
  </project>
  <project>
    <project_id>PCDE</project_id>
    <budget>576.54</budget>
  </project>
</projects>

<projects_summary>
  <budget_summary>
    <max_budget>598.76</max_budget>
    <total_budget>1762.95</total_budget>
  </budget_summary>
</projects_summary>

<dept_id>D234</dept_id>
<dept_name>Auxiliary</dept_name>
<emps>
  <emp_id>E345</emp_id>
  <emp_name>Don Davis</emp_name>
  <salary>945.67</salary>
  <phones>
    <phone><phone_no>650.555.5001</phone_no></phone>
  </phones>
</emp>
<emp>
  <emp_id>E345</emp_id>
  <emp_name>Earl Evans</emp_name>
  <phones>
    <phone><phone_no>650.555.5001</phone_no></phone>
  </phones>
</emp>
</emps>
<phones/>

</emp>
</emps>
<emps_summary>
<salary_summary>
  <max_salary>945.67</max_salary>
  <total_salary>945.67</total_salary>
</salary_summary>
</emps_summary>

</emps_summary>
<projects>
  <project>
    <project_id>PDEF</project_id>
  </project>
  <project>
    <project_id>PEFG</project_id>
    <budget>554.32</budget>
  </project>
</projects>
<projects_summary>
  <budget_summary>
    <max_budget>554.32</max_budget>
    <total_budget>554.32</total_budget>
  </budget_summary>
</projects_summary>
</dept>
</dept>
<dept>
  <dept_id>D345</dept_id>
  <dept_name>Repair</dept_name>
  <emps>
    <emp>
      <emp_id>E678</emp_id>
      <emp_name>Fred Frank</emp_name>
      <salary>967.89</salary>
      <phones>
        <phone>408.555.6111</phone>
      </phones>
    </emp>
    <emp>
      <emp_id>E789</emp_id>
      <emp_name>George Gordon</emp_name>
    </emp>
  </emps>
</dept>
<salary>978.90</salary>
<phones>
    <phone><phone_no>510.555.7654</phone_no></phone>
</phones>
</emp>
<emp>
    <emp_id>E901</emp_id>
    <emp_name>Hank Hartley</emp_name>
    <salary>990.12</salary>
    <phones/>
</emp>
<emp>
    <emp_id>E678</emp_id>
    <emp_name>Isaak Idle</emp_name>
    <salary>990.12</salary>
    <phones>
        <phone><phone_no>925.555.9991</phone_no></phone>
        <phone><phone_no>650.555.9992</phone_no></phone>
        <phone><phone_no>415.555.9993</phone_no></phone>
    </phones>
</emp>
<emps>
    <emps_summary>
        <salary_summary>
            <max_salary>990.12</max_salary>
            <total_salary>2936.91</total_salary>
        </salary_summary>
        <projects>
            <project>
                <project_id>PFGH</project_id>
                <budget>543.21</budget>
            </project>
            <project>
                <project_id>PGHI</project_id>
            </project>
            <project>
                <project_id>PHIJ</project_id>
                <budget>521.09</budget>
            </project>
        </projects>
        <projects_summary>
            <budget_summary>
                <max_budget>543.21</max_budget>
                <total_budget>1064.30</total_budget>
            </budget_summary>
        </projects_summary>
    </emps_summary>
</emps>
Using the *depts* document

The *depts* document is stored in a new row of the *sample_docs* table in Appendix A, “The sample_docs Example Table.” To reference this document in examples:

```sql
declare @dept_doc xml
select @dept_doc from sample_docs where name_doc='depts'
```

The *depts* document structure

The structure of the *depts* document is as follows:

```xml
<depts>
  <dept>
    <emps> - repeats under <depts>
    <emp> - one for each <dept>
      <emp_id> - one for each <emp>
      <emp_name> - one for each <emp>
      <phones> - one for each <emp>
        <phone> - repeats under <phones>
        <phone_no> - one for each <phone>
    <projects> - one for each <dept>
    <project> - repeats for each under <projects>
      <project_id> - one for each <project>
      <dept_id> - one for each <project>
  </dept>
</depts>
```

Creating SQL tables from the *depts* document

Normalizing the data from the *depts* document

You can normalize this data into SQL tables. For example:
Using the depts document

- **depts** – A row for each `<dept>` element, containing the `<dept_id>` and `<dept_name>`.
- **emps** – A row for each `<emp>` element, containing the `<emp_id>` and `<emp_name>`, as well as the `<dept_id>` element, which contains them.
- **emp_phones** – A row for each `<phone>`, containing the `<phone_no>` and the `<dept_id>` element, which contains all these elements.
- **projects** – A row for each `<project>`, containing the `<project_id>` and `<budget>` elements, and the containing `<dept_id>` element.

Generating tables using `select`

All the tables generated in this section, except the `depts` table, have a column pattern that uses the XPath ancestor notation to reference:

- Leaf elements, such as `<project>`, under `<projects>`, or `<salary>`, under `<emp>`.
- The element that contains the element defining the table, as `<emp>`, for example, contains `<emp_id>`.

This notation “flattens” nested data. For more information about flattening XML data, see Chapter 7, “xmltable()”.

**emps table**

In this select statement, the column pattern for the `<dept_id>` column references the `<dept_id>` element in the `<dept>`, which contains the current `<emp>`.

```sql
declare @dept_doc xml
select @dept_doc = doc from sample_docs where name_doc = 'depts'
select * into emps from xmltable('//emp' passing @dept_doc
    columns emp_id char(4),
    emp_name varchar(50),
    salary money,
    dept_id char(4) pattern './/dept_id') as dept_extract
select * from emps
```

<table>
<thead>
<tr>
<th>emp_id</th>
<th>emp_name</th>
<th>salary</th>
<th>dept_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>E123</td>
<td>Alex Allen</td>
<td>912.34</td>
<td>D123</td>
</tr>
<tr>
<td>E234</td>
<td>Bruce Baker</td>
<td>923.45</td>
<td>D123</td>
</tr>
<tr>
<td>E345</td>
<td>Carl Curtis</td>
<td>934.56</td>
<td>D123</td>
</tr>
<tr>
<td>E456</td>
<td>Don Davis</td>
<td>945.67</td>
<td>D234</td>
</tr>
<tr>
<td>E567</td>
<td>Earl Evans</td>
<td>956.78</td>
<td>D234</td>
</tr>
</tbody>
</table>
APPENDIX D  Sample Application for xmltable()

phones table

In the phones table, the column pattern for the emp_id column references the <emp_id> element in <emp>, which contains the current element <phone>.

```sql
declare @ dept_doc xml
select @ dept_doc
    = doc from sample_docs where name_doc='depts'
select * into phones
from xmltable('//phone' passing @ dept_doc
    columns emp_id char(4),'/emp_id'
    phone_no varchar(50)) as dept_extract
select * from phones
```

<table>
<thead>
<tr>
<th>emp_id</th>
<th>phone_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>E123</td>
<td>510.555.1987</td>
</tr>
<tr>
<td>E123</td>
<td>510.555.1876</td>
</tr>
<tr>
<td>E234</td>
<td>203.555.2333</td>
</tr>
<tr>
<td>E345</td>
<td>408.555.3123</td>
</tr>
<tr>
<td>E345</td>
<td>415.555.3987</td>
</tr>
<tr>
<td>E345</td>
<td>650.555.3777</td>
</tr>
<tr>
<td>E567</td>
<td>650.555.5001</td>
</tr>
<tr>
<td>E678</td>
<td>408.555.6111</td>
</tr>
<tr>
<td>E678</td>
<td>408.555.6222</td>
</tr>
<tr>
<td>E789</td>
<td>510.555.7654</td>
</tr>
<tr>
<td>E901</td>
<td>925.555.9991</td>
</tr>
<tr>
<td>E901</td>
<td>650.555.9992</td>
</tr>
<tr>
<td>E901</td>
<td>415.555.9993</td>
</tr>
</tbody>
</table>

projects table

In the projects table, the column pattern for the dept_id column references the <dept_id> element, in the <dept> that contains the current <project>.

```sql
declare @ dept_doc xml
select @ dept_doc
    = doc from sample_docs where name_doc='depts'
select * into projects
from xmltable('//project' passing @ dept_doc
    columns project_id char(4),
    dept_id char(4)) as dept_extract
```

<table>
<thead>
<tr>
<th>project_id</th>
<th>dept_id</th>
<th>phone_no</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XML Services 155
Using the depts document

```sql
-- Using the depts document

budget money,
    dept_id char(4) pattern '.../..dept_id')
as dept_extract
select * from projects
---------------
project_id     budget     dept_id
----------     -------     -------
PABC           598.76      D123
PBCD           587.65      D123
PCDE           576.54      D123
PDEF           565.43      D234
PEFG           554.32      D234
PFGH           543.21      D345
PGHI           NULL        D345
PHIJ           521.09      D345
```

```
depts table

declare @ dept_doc xml
select @ dept_doc = doc from sample_docs where name_doc='depts'
select * into depts
from xmltable('//dept' passing @ dept_doc
    columns dept_id char(4),
    dept_name char(4)) as dept_extract
select * from depts
---------------
dept_id     dept_name
--------     -----------
D123        Main
D234        Auxiliary
D345        Repair
```

---

Adaptive Server Enterprise
Index

Symbols
@@error
error number of last error 15
global variable 15

Numerics
16-bit values, surrogate pairs 92

A
ample_docs table
titles table, XML representation 127
any/all quantified predicate subquery, cannot use for
xml subquery as 64
at clause command 135
attributes, embedded in element tags 4

B
base64, SQLX option 73
basic operators, XPath, supported 47
BCP, for transferring data 92
binary
datatype 73, 89
option 73
SQLX option 73
values 89
binary option 40
binary SQLX option 72
bookstore, sample doc 124

C
char datatype 2, 88
character encoding. See character sets
color literals 44
color set conversions 134
color set support 41
color set transfer, Java 93
color sets
and XML data 93
color set transfer, Java 93
declared matching actual 4
default UTF8 4
specifying 4
translations, bypassed 4
XML 4
color value, example 89
CIS (Component Integration Services) 133
color set, color sets 93
color set transfer, Java 93
color-server data
transferring 92
using Java 92
code samples
HTML, Order example 6
XML, Info example 4
XML, Item example 5
column
image, parsed XML 3
column string/unicode 92
columnstyle option 40
columnstyle SQLX option 72
columnstyle, SQLX option 72, 82
command
at clause 135
create proxy table 135
select 76
xmlparse xmltest 1
xmltest 1
xmltextextract 1
xmlvalidate 1, 29
command create proxy table 135
commands
Index

xmlvalidate 29
comparison operators in XPath 49
Component Integration Services (CIS) 133
concat string function 49, 54
concat, function 54
concat, XPath string function 50
content columns, Unicode, rules for assigning 134
correlated subquery, cannot be for xml clause 64
create proxy table command 135
create view command, cannot use for xml subquery in 64
CTLIB, for transferring data 92
customizing elements 4

datatype
char 2
image 2
text 2
datatypes
binary 73, 89
categories in Unicode 91
char 88
image 73, 89
numeric 88
text 88
varbinary 73, 89
varchar 88
datatypes for storing XML 1
datatypes, Unicode
char 91
text 91
unitext 91
univarchar 91
varchar 91
datatype Unicode, Unicode
datatype 92
declare cursor command, cannot use for xml subquery in 64
default ncr option 94
default xml sort order, in sp_configure 100
default xml sort order, modifying 101
derived table syntax 104
derived table syntax, in xmltable 103
description
subqueries, for xml 63
disable_unichar_sending property 93
document
and queries, migrating 144
migrating between Java and native processors 144
Document Type Definition. See DTD
DTD 7
#IMPLIED 8
#PCDATA DTD elements 8
asterisk (*) 8
ATTLIST 8
ELEMENT 8
embedded 8
for valid XML document 9
not required in all documents 9
plus sign (+) 8
question mark (?) 8
DTDS and schemas, xmlvalidate 33
dtdvalidate option 40
dtdvalidate,validation option 30
duplicate column names, mapping 82

d
EFS
examples 134
EFS access 133
element tags
customizing 4
embedded attributes 4
HTML 6
strict nesting 4
user-created 4
elements, empty 45
embedded DTD 8
embedding DTD in XML 8
empty elements 45
encoding option, xmlparse 99
entities
predefined, in XML language 42
predefined, in XML query language 44
entitize
option 40, 72
SQLX option 74
Euro symbol, € 92
example
parsing XML documents, using xmlerror=message with XFS 140
specifying xmlerror=message option, with XFS 139
storing parsed XML documents with external file system 137
using xmlerror=null 141
xmlerror option, with XFS 138
example table
sample_docs 123
Unicode examples 95
examples
for xml clause 62, 95
subqueries, for xml 64
examples of XML documents 15
exceptions
causes to raise 14
for xml clause 62
subqueries, for xml 64
events not exists quantified predicate subquery, cannot use for xml subquery as 64
events not exists, quantified predicate subquery 64
expressions, parenthesized in XPath 54
Extensible Style Language. See XSL
extensions
for xml clause 65
for xml clause, descriptions 65
for xml clause, examples 65
for xml clause, exceptions 65
External Directory Recursive Access, maps proxy table 133
External File System access, in XML 133
external file systems, character set conversions 134
external file systems, Unicode columns 134

F
File System Access, feature 133
for processing XML documents containing non-ASCII data 91
for xml
clause 71, 76
clause, syntax and examples of 59
description of 60
in data mapping 82
SQLX-XML format 82
for xml all 65
for xml clause
cannot be correlated subquery 64
cannot be used in nested scalar subquery 64
cannot use as quantified predicate subquery 64
cannot use in commands 64
extamples 62, 95
exceptions 62
extensions 65
extensions description 65
extensions examples 65
generating non-ASCII data 91
options 62
syntax 59
using with isql 71
for xml clause extensions exceptions 65
for xml header option 95
for xml schema 65
for xml select command, cannot use for xml subquery in 64
for xml subqueries 63
description 63
examples 64
exceptions 64
for xml, Unicode 93
format [yes | no] SQLX option 75
format option 40
format SQLX option 72
formatting
instructions, provided in XSL 2
SQLX-XML 71
forxml function 76
function
concat 49, 54
forxml 76
xmlextract 12
xmlparse 23
xmlrepresentation 27
xmltable 103, 104
function concat, XPath string function 50
function normalize-space, XPath string function 50
function predicate xmltest 19
Index

function tolower, XPath string function 49
function toupper, XPath string function 49
functions
  concat 54
  mapping 59
  normalize-space 53
tolower, toupper 53
XPath 47
functions, XML, documented 53

G

generating XML with or xml 1
global variables
  @@error 15
guidelines, for XPath functions 50

H

handling non-ASCII data with for xml clause 93
header
  SQLX option 72, 75
header option 40
hex, SQLX option 73
HTML
  display of Order data 5
  DTD elements 7
  element bracketing inconsistent 6
  inconsistent element tags 6
  limitations 6
  Order code sample 5, 6

I

€, Euro 92
I18N
  examples 95
  non-ASCII data 91
image column 3
image datatype 73, 89
image, datatype 101
image_doc, sample_docs table columns 123

in/not in quantified predicate subquery, cannot use for
xml subquery as 64
inconsistent element tags, HTML 6
incremental SQLX option 72, 76
info, XML code example 4
ISQL, for transferring data 92
isql, using with for xml clause 71

J

Java
  sample directory for character set of client 93
  Java processor 1
  java.lang.String datatype 2
  java.lang.String, datatype 101
Java-based processor
  migrating to native XML processor 143, 149
  migrating with native XML processor 144
  regenerating text documents 145
JDBC code
  use property disable_unicharSending 93

L

languages
  XML and XML query 41
  XPath 41
  XQL 41

M

mapping
  duplicate column names 82
  duplicate column names, example 82
  SQL names, objectives 85
  SQL values 88
  SQL values, examples 88
  to XML names 85
to XML values 88
  unnamed columns 82
migrating
  between Java-based and native processors 143, 149
Index

supported 14
option syntax, xmlvalidate 30
option values, query functions 40, 44
option xsidecl={yes | no} 81
option_string values, table of 40
option_strings
  description 39
  description and examples 39
  discussion 39
  general format 39
  parameter 11
  syntax 39
options
  for xml clause 62
  SQLX 71, 72
  SQLX, definitions 72
options for xml all 77
options, validation 30
Order DTD, sample code 7
Order sample
  HTML 6
  XML code 3

P
parenthesized expressions
  and subscripts 55
  and unions 56
parsed XML 3
plain characters or predefined entities 44
plus sign (+) in XML document type definition 8
predefined entities 42, 44
prefix option 40
prefix SQLX option 72, 87
preserved white space 44
processing non-ASCII data 99
proxy table, created by File System Access feature 133
publishers, pubs2 database table 124

Q
queries
  migrating 144
  migrating between Java-based and native processors

147
query functions
  syntax and examples 12
  table of 12
  XML 11
  xmlextract 12
  xmlrepresentation 12
  xmltest 12
  xmlvalidate 12
query functions, option values for 40, 44
query functions, SQL extensions 11
querying XML
  with xmltest, and xmlextract 1

R
referencing, XML DTD externally 8
regenerating
text documents, from Java-based processor 145
text documents, from native XML processor 146
relative function call 50
restrictions, in migrating documents 143
root option 40
root SQLX option 72, 78
rowname option 40
rowname, SQLX option 72, 78, 87
rules for Unicode content columns 134

S
sample code
  DTD, Order example 7
  HTML, Order sample 6
  XML, Info example 4
  XML, Order example DTD 7
sample directory for character set transfer, Java 93
sample table, sample_docs 123
sample_doc table
  rows 124
sample_docs table
  columns 123
  publishers and titles tables 125
  publishers table, XML representation 126
  structure 123
Index

XML Services 163

table script (publishers) 126
sample_docs table columns
  image_doc 123
  name_doc sample_docs table columns 123
  text_doc 123
schema support 42
schemaloc SQLX option 72, 79
schemalocation option 40
schemavalidate option 40
searching, XML documents stored on Web 2
select command 76
select into command, cannot use for xml subquery in 64
select statement
  ncr option 94
server character set, different from client 93
SGML, Standardized General Markup Language 2
shredding XML, with xmltest, and xmlextract 1
sort order
  xmlparse 99
sort ordering
  xmlextract 100
sort ordering in XML Services 101
sp_configure
  enabling XML Services and External File System Access 135
  sp_configure option default xml sort order 100
  sp_configure, enabling XML Services and External File System Access 133
space, white, preserved 44
specifying character set 4
SQL extensions, query functions as 11
SQL names, examples 87
sql_name, SQLX option 78
SQLX
  data 82
  data mapping 82
  options, definitions 72
  options, table 72
SQLX options 71
  base64 73
  binary 72, 73
  columnstyle 72, 73, 82
  entitize 74
  format 72
  format={yes | no} 75
header 72, 75
hex 73
incremental 72, 76
nullstyle 72, 77
prefix 72, 87
root 72, 78
rowname 72, 78, 87
schemaloc 72, 79
sql_name 78
statement 72, 80
tablename 72, 80, 87
tablename=sqlname 80
targetns 72, 81
targetns=url 81
SQLX-XML
  format 71
SQLX-XML format 82
Standardized General Markup Language (SGML) 2
statement option 40
statement SQLX option 72, 80
storing XML
  as datatypes 2
    parsed in image column 2
storing XML with datatypes, xmlparse, and xmlvalidate 1
string, datatype 101
string/ unicode columns 92
subqueries, for xml 63
subscripts, in parenthesized expressions 55
surrogate pairs, 16-bit value 92
surrogate pairs, Unicode 92
syntax
  examples, xmlextract 12
  for xml clause 59
  for xml clause, description 59
  option_strings 39
  subqueries, for xml 63
  xmlparse 23
  xmltest 19
  XPath tokens 45
syntax, xmlrepresentation 27

T

table
values
  binary 89
  character 88
  numeric 88
varbinary datatype 89
varchar datatype 2, 88
varchar univarchar column 95

variables, HTML
  <b>...</b> 6
  <table>...</table>, layout 6
  bgcolor, color 6
  CustomerID 6
CustomerName 6
  Data 6
  ItemID 5
  ItemName 5
  order 5
  Quantities 6
  Quantity 5
  units 5

variables, XML tag 4

W
  well-formed XML document 4
  white space, preserved 44

X
  XFS
    specifying xmlerror=message 139, 140
XML
  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 7
  DTD not required in all documents 9
  DTD sample code, embedding 8
  DTD sample code, referencing externally 8
  DTD, instruction 7
  mapping functions 59
  mappings 71

names, mapping to 85
parsed 3
query functions 11
query language 41
read by HTML browsers and processors 2
representation of publishers table 126
representation of titles table 127
sample document 3
schema declarations 42
strict phrase structure 2
subset of SGML 2
suitable for data interchange 2
values, mapping to 88
xml
  sample document 124
xml all options 77
XML document
  as character data 4
  creating proxy table, sample code 135
  DTD sample code 7
  example, info 4
  extracting book title from, sample code 136
  generating from Adaptive Server 2
  importing to ASE table, sample code 136
  namespace support, XML documents 42
  nested markup tags 4
  no formatting instructions 4
  parts of 4
  querying from subdirectory, sample code 137
  sample code, Order 3
  searching on Web 2
  storing in Adaptive Server 2
  strict phrase structure 2
  tags 2
  valid with DTD 9
  well-formed 4
XML documents, examples of 15
XML EFS access 133
XML functions 53
XML query language, subset of XPath 45
XML Services
  using sp_configure 133
  with External File System Access 133
    with External File System Access, using
      sp_configure 135
XML services

XML Services 165
Index

parenthesized expressions 54
XPath string functions 49
xmlerror option 40
xmlerror, supported option string 14
xmlexparse
  options 24
xmlrepresentation
  description 27
xmlextract 91, 99, 101
  built-in query function 12
  description 13
  examples 15
  exceptions 14
  extracts XML 1
  function description 12
  namespace prefixes 42
  ncr option 100
  sort ordering 100
  support for XQuery language subset 42
  syntax 12
xmlextract command 1
xmlextract, query function 12
xmlparse 99, 100
  built-in function 12
  description 24
  examples 25
  exceptions 25
  function description 23
  sort order 99
  storing non-ASCII data 91
  syntax 23
Unicode 99
  user modification of option 101
xmlparse command 1
xmlparse, encoding option 99
xmlrepresentation
  examples 27
  function description 27
xmlrepresentation, query function, determines parsed image
  column 12
xmltable
  description, syntax 104
  examples 104
  function 103, 104
  ordonality columns example 110
  processing tables of documents example 115
  see also 121
  usage 118
xmltable, with derived table syntax 103
xmltest 91
  command 1
  description 19, 20
  examples 21
  function 19
  function predicate 19
  options 21
  queries XML 1
  query function 12
  syntax and description 19
xmltext
  namespace prefixes 42
  SQL predicate, returns boolean result 12
xmlvalid option 40
xmlvalidate
  command 29
  description 29, 30
  examples 33
  exceptions 32
  options 30
  options description 30
  query function 12
  syntax 29
xmlvalidate command 1
xmlvalidate, command 29
xmlvalidate, supports NCR option 101
xmlvalidate, supports Unicode 101
XPath
  basic operators, table 48
  comparison operators 49
  comparison operators, table 49
  examples 50
  general guidelines 50
  language 1
  language subset 45
  operators and functions 47
  parenthesized expressions 54
  set operators, table 48
  supported tokens 46
  syntax and tokens 45
  XPath 1.0 41
  XPath string functions 49
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>examples</td>
<td>50</td>
</tr>
<tr>
<td>XQL processor, migrating documents</td>
<td>144</td>
</tr>
<tr>
<td>XQuery language, supported by xmlextact and xmltest</td>
<td>42</td>
</tr>
<tr>
<td>xscdecl option</td>
<td>40</td>
</tr>
<tr>
<td>xsidclem option</td>
<td>72</td>
</tr>
<tr>
<td>xsidclem={yes</td>
<td>no} option</td>
</tr>
<tr>
<td>XSL, Extensible Style Language</td>
<td>2</td>
</tr>
</tbody>
</table>
Index