Db2 11 for z/OS

pureXML Guide



SC19-4064-06

Notes

Before using this information and the product it supports, be sure to read the general information under "Notices" at the end of this information.

Subsequent editions of this PDF will not be delivered in IBM Publications Center. Always download the latest edition from PDF format manuals for Db2 11 for z/OS (Db2 Home).

2020-07-30 edition

This edition applies to Db2® 11 for z/OS® (product number 5615-DB2), Db2 11 for z/OS Value Unit Edition (product number 5697-P43), and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Specific changes are indicated by a vertical bar to the left of a change. A vertical bar to the left of a figure caption indicates that the figure has changed. Editorial changes that have no technical significance are not noted.

© Copyright International Business Machines Corporation 2007, 2020.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this information..... ix
Who should read this information..... ix
Db2 Utilities Suite for z/OS..... ix
Terminology and citations..... x
Accessibility features for Db2 11 for z/OS..... x
How to send your comments..... xi
How to read syntax diagrams..... xi

Chapter 1. Overview of pureXML..... 1
pureXML data model..... 2
Sequences and items..... 2
Atomic values..... 3
Nodes..... 3
Data model generation in XQuery..... 11
Comparison of the XML model and the relational model..... 12
XML data type..... 14
Tutorial: Working with XML data..... 15
Prerequisites for using pureXML..... 17
Setting up the XML schema repository..... 18

Chapter 2. Working with XML data..... 27
Creation of tables with XML columns..... 27
Altering tables with XML columns..... 27
Storage structure for XML data..... 29
Limitation of XML virtual storage usage..... 32
Insertion of rows with XML column values..... 33
Updates of XML columns..... 35
Updates of entire XML documents..... 35
Partial updates of XML documents..... 36
Deletion of rows with XML documents from tables..... 38
XML versions..... 39
XML support in triggers..... 41
XML parsing..... 42
XML parsing and whitespace handling..... 42
XML parsing and DTDs..... 43
XML schema validation..... 45
XML schema validation and ignorable whitespace..... 45
XML schema validation with an XML type modifier..... 46
XML schema validation with DSN_XMLVALIDATE..... 53
How to determine whether an XML document has been validated..... 56
Casts between XML data types and SQL data types..... 57
Examples of casts from XML schema data types to SQL data types..... 57
Retrieving XML data..... 60
Retrieval of an entire XML document from an XML column..... 60
XMLQUERY function for retrieval of portions of an XML document..... 61
XMLEXISTS predicate for querying XML data..... 63
Constant and parameter marker passing to XMLEXISTS and XMLQUERY..... 64
XMLTABLE function for returning XQuery results as a table..... 65
XML support in native SQL routines..... 70
Requests for data in XML columns by earlier Db2 clients..... 72

Functions for constructing XML values ... 72
Special character handling in SQL/XML publishing functions ... 76
XML serialization ... 77
Differences in an XML document after storage and retrieval ... 79
Transforming an XML document with XSLTRANSFORM ... 80

Chapter 3. XML data indexing ... 81
Pattern expressions ... 82
Namespace declarations in XML index definitions ... 83
Data types associated with pattern expressions ... 84
XML schemas and XML indexes ... 84
The UNIQUE keyword in an XML index definition ... 85
Access methods with XML indexes ... 86
Example of DOCID ANDing access (ACCESSTYPE='DI') ... 87
Example of DOCID ORing access (ACCESSTYPE='DU') ... 88
Examples of index definitions and queries that use them ... 89
Examples of XML index usage by equal predicates ... 89
Examples of XML index usage by predicates that test for node existence ... 90
Example of XML index usage by predicates with case-insensitive comparisons ... 91
Example of index usage for an XMLEXISTS predicate with the fn:starts-with function ... 91
Example of index usage for an XMLEXISTS predicate with the fn:substring function ... 91
Example of XML index usage by join predicates ... 92
Example of XML index usage by queries with XMLTABLE ... 92

Chapter 4. XML support in Db2 utilities ... 95

Chapter 5. XML schema management with the XML schema repository (XSR) ... 103
Procedures for XML schema registration and removal that are supplied with Db2 ... 104
Example of XML schema registration and removal using stored procedures ... 104

Chapter 6. Db2 application programming language support for XML ... 107
XML data in Java applications ... 107
XML data in embedded SQL applications ... 108
Host variable data types for XML data in embedded SQL applications ... 108
XML column updates in embedded SQL applications ... 113
XML data retrieval in embedded SQL applications ... 115
XML data in ODBC applications ... 118
XML column updates in ODBC applications ... 118
XML data retrieval in ODBC applications ... 120
Data types for archiving XML documents ... 122

Chapter 7. XML data encoding ... 123
Background information on XML internal encoding ... 123
XML encoding considerations ... 124
Encoding considerations for input of XML data to a Db2 table ... 124
Encoding considerations for retrieval of XML data from a Db2 table ... 125
XML data encoding in JDBC and SQLJ applications ... 125
XML encoding scenarios ... 126
Encoding scenarios for input of internally encoded XML data to a Db2 table ... 126
Encoding scenarios for input of externally encoded XML data to a database ... 128
Encoding scenarios for retrieval of XML data with implicit serialization ... 130
Encoding scenarios for retrieval of XML data with explicit XMLSERIALIZE ... 132
Mappings of encoding names to effective CCSIDs for stored XML data ... 135
Mappings of CCSIDs to encoding names for textual XML output data ... 148

Chapter 8. Overview of XQuery ... 155

Best applications for XQuery or XPath....157
XML namespaces and qualified names in XQuery....158
Case sensitivity in XQuery....159
Whitespace in XQuery....160
Comments in XQuery....160

Chapter 9. XQuery type system....163
Overview of the type system....163
Constructor functions for built-in data types....163
Generic data types....164
xs:anyType....164
xs:anySimpleType....164
xs:anyAtomicType....165
Data types for untyped data....165
xs:untyped....165
xs:untypedAtomic....165
xs:string....166
Numeric data types....166
xs:decimal....166
xs:double....167
xs:integer....167
Range limits for numeric types....168
xs:boolean....168
Date and time data types....169
xs:date....169
xs:dateTime....170
xs:dayTimeDuration....171
xs:duration....172
xs:time....173
xs:yearMonthDuration....174
Casts between XML schema data types....175

Chapter 10. XQuery prologs and expressions....179
Prologs....179
Boundary-space declaration....180
Copy-namespaces declaration....180
Namespace declarations....181
Default namespace declarations....182
Expressions....183
Expression evaluation and processing....183
Primary expressions....184
Path expressions....188
Sequence expressions....196
Filter expressions....196
Arithmetic expressions....197
Comparison expressions....200
Logical expressions....205
XQuery constructors....206
FLWOR expressions....217
Conditional expressions....228
Basic updating expressions....230
Castable expressions....237
Regular expressions....239

Chapter 11. Descriptions of XQuery functions....243
fn:abs function....243
fn:adjust-date-to-timezone function....244

fn:adjust-dateTime-to-timezone function....245
fn:adjust-time-to-timezone function....247
fn:avg function....248
fn:boolean function....249
fn:compare function....250
fn:concat function....251
fn:contains function....251
fn:count function....252
fn:current-date function....252
fn:current-dateTime function....252
fn:current-time function....253
fn:data function....253
fn:dateTime function....254
fn:day-from-date function....254
fn:day-from-dateTime function....255
fn:days-from-duration function....255
fn:distinct-values function....256
fn:hours-from-dateTime function....257
fn:hours-from-duration function....257
fn:hours-from-time function....258
fn:implicit-timezone function....258
fn:minutes-from-dateTime function....259
fn:minutes-from-duration function....259
fn:minutes-from-time function....260
fn:month-from-date function....260
fn:month-from-dateTime function....261
fn:months-from-duration function....261
fn:normalize-space function....262
fn:last function....263
fn:local-name function....264
fn:lower-case function....265
fn:matches function....265
fn:max function....267
fn:min function....268
fn:name function....269
fn:not function....270
fn:position function....270
fn:replace function....271
fn:round function....272
fn:seconds-from-datetime function....273
fn:seconds-from-duration function....274
fn:seconds-from-time function....274
fn:starts-with function....275
fn:string function....275
fn:string-length function....276
fn:substring function....277
fn:sum function....277
fn:timezone-from-date function....278
fn:timezone-from-dateTime function....278
fn:timezone-from-time function....279
fn:tokenize function....280
fn:translate function....281
fn:upper-case function....282
fn:year-from-date function....283
fn:year-from-datetime function....283
fn:years-from-duration function....284

Information resources for Db2 11 for z/OS and related products..... 285

Notices.....287
General-use Programming Interface and Associated Guidance Information..... 288
Trademarks..... 288
Terms and conditions for product documentation..... 289
Privacy policy considerations..... 289

Glossary..... 291

Index..... 293

About this information

This information describes pureXML® (Db2 for z/OS support for XML). This support lets you store XML data in Db2 databases and retrieve XML data from Db2 databases.

Throughout this information, "Db2" means "Db2 11 for z/OS". References to other Db2 products use complete names or specific abbreviations.

Important: To find the most up to date content for Db2 11 for z/OS, always use IBM® Knowledge Center or download the latest PDF file from PDF format manuals for Db2 11 for z/OS (Db2 Home).

This information assumes that Db2 11 is running in new-function mode, and that your application is running with the application compatibility value of 'V11R1'.

Availability of new function in Db2 11

The behavior of data definition statements such as CREATE, ALTER, and DROP, which embed data manipulation SQL statements that contain new capabilities, depends on the application compatibility value that is in effect for the application. An application compatibility value of 'V11R1' must be in effect for applications to use new capability in embedded statements such as SELECT, INSERT, UPDATE, DELETE, MERGE, CALL, and SET *assignment-statement*. Otherwise, an application compatibility value of 'V10R1' can be used for data definition statements.

Generally, new SQL capabilities, including changes to existing language elements, functions, data manipulation statements, and limits, are available only in new-function mode with applications set to an application compatibility value of 'V11R1'.

Optimization and virtual storage enhancements are available in conversion mode unless stated otherwise.

SQL statements can continue to run with the same expected behavior as in DB2® 10 new-function mode with an application compatibility value of 'V10R1'.

Who should read this information

This information is for the following users:

- Db2 for z/OS application developers who are familiar with Structured Query Language (SQL) and who are familiar with XML.
- Db2 for z/OS database managers who are familiar with XML.

Db2 Utilities Suite for z/OS

Important: In Db2 11, the Db2 Utilities Suite for z/OS is available as an optional product. You must separately order and purchase a license to such utilities, and discussion of those utility functions in this publication is not intended to otherwise imply that you have a license to them.

Db2 11 utilities can use the DFSORT program regardless of whether you purchased a license for DFSORT on your system. For more information, see the following informational APARs:

- II14047
- II14213
- II13495

Db2 utilities can use IBM Db2 Sort for z/OS (5655-W42) as an alternative to DFSORT for utility SORT and MERGE functions. Use of Db2 Sort for z/OS requires the purchase of a Db2 Sort for z/OS license. For more information about Db2 Sort for z/OS, see Db2 Sort for z/OS.

Related concepts

Db2 utilities packaging (Db2 Utilities)

© Copyright IBM Corp. 2007, 2020

Terminology and citations

When referring to a Db2 product other than Db2 for z/OS, this information uses the product's full name to avoid ambiguity.

About the Db2 brand change: IBM has re-branded DB2 to Db2, and Db2 for z/OS is now the name of the offering previously know as "DB2 for z/OS". During and after implementation of this change, you might sometimes still see references to the original names, such as "DB2 for z/OS" and "DB2", in different IBM web pages and documents. However, assume that these refer to the same "Db2 for z/OS" product when the PID, Entitlement Entity, version, modification, and release information match. For more information, see Revised naming for IBM Db2 family products on IBM z/OS platform.

The following terms are used as indicated:

Db2
Represents either the Db2 licensed program or a particular Db2 subsystem.

Tivoli® OMEGAMON® XE for Db2 Performance Expert on z/OS
Refers to any of the following products:

- IBM Tivoli OMEGAMON XE for Db2 Performance Expert on z/OS
- IBM Db2 Performance Monitor on z/OS
- IBM Db2 Performance Expert for Multiplatforms and Workgroups
- IBM Db2 Buffer Pool Analyzer for z/OS

C, C++, and C language
Represent the C or C++ programming language.

CICS®
Represents CICS Transaction Server for z/OS.

IMS
Represents the IMS Database Manager or IMS Transaction Manager.

MVS™
Represents the MVS element of the z/OS operating system, which is equivalent to the Base Control Program (BCP) component of the z/OS operating system.

RACF®
Represents the functions that are provided by the RACF component of the z/OS Security Server.

Accessibility features for Db2 11 for z/OS

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in z/OS products, including Db2 11 for z/OS. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers and screen magnifiers.
- Customization of display attributes such as color, contrast, and font size

Tip: The IBM Knowledge Center (which includes information for Db2 for z/OS) and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard navigation

For information about navigating the Db2 for z/OS ISPF panels using TSO/E or ISPF, refer to the *z/OS TSO/E Primer*, the *z/OS TSO/E User's Guide*, and the *z/OS ISPF User's Guide*. These guides describe how to

navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other Db2 for z/OS documentation.

Send your comments by email to db2zinfo@us.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title or a help topic title).

How to read syntax diagrams

Certain conventions apply to the syntax diagrams that are used in IBM documentation.

Apply the following rules when reading the syntax diagrams that are used in Db2 for z/OS documentation:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►—— symbol indicates the beginning of a statement.

  The ——► symbol indicates that the statement syntax is continued on the next line.

  The ►—— symbol indicates that a statement is continued from the previous line.

  The ——►◄ symbol indicates the end of a statement.
- Required items appear on the horizontal line (the main path).

  ►► *required_item* ►◄
- Optional items appear below the main path.

  ►► *required_item* ┬──────────┬►◄
  └ *optional_item* ┘

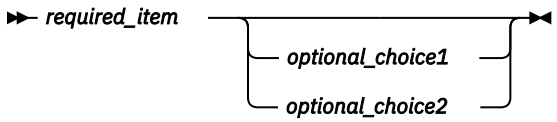
  If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

  ┌ *optional_item* ┐
  ►► *required_item* ┴──────────┴►◄
- If you can choose from two or more items, they appear vertically, in a stack.

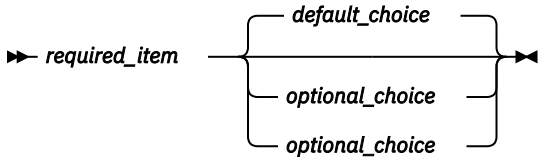
  If you *must* choose one of the items, one item of the stack appears on the main path.

  ►► *required_item* ┬ *required_choice1* ┬►◄
  └ *required_choice2* ┘

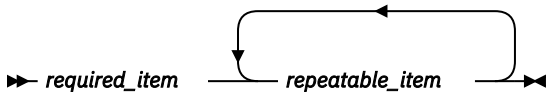
  If choosing one of the items is optional, the entire stack appears below the main path.



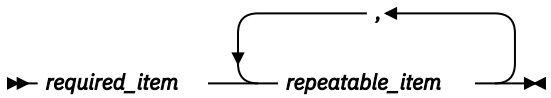
If one of the items is the default, it appears above the main path and the remaining choices are shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.

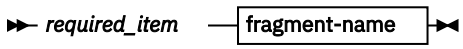


If the repeat arrow contains a comma, you must separate repeated items with a comma.

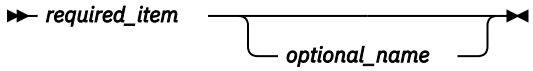


A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.



fragment-name



- With the exception of XPath keywords, keywords appear in uppercase (for example, FROM). Keywords must be spelled exactly as shown. XPath keywords are defined as lowercase names, and must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Related concepts

Commands in Db2 (Db2 Commands)

Db2 online utilities (Db2 Utilities)

Db2 stand-alone utilities (Db2 Utilities)

Chapter 1. Overview of pureXML

pureXML is Db2 for z/OS support for XML. pureXML lets your client applications manage XML data in Db2 tables.

You can store well-formed XML documents in their hierarchical form and retrieve all or portions of those documents.

Because the stored XML data is fully integrated into the Db2 database system, you can access and manage the XML data by leveraging Db2 functions and capabilities.

To efficiently manage traditional SQL data types and XML data, Db2 stores XML data in separate table spaces from the tables that contain XML columns. However, the underlying storage mechanism that is used for XML data is transparent to the application. The application does not need to explicitly specify which XML table spaces to use, or to manage the physical storage for XML and non-XML objects.

XML document storage: The XML column data type is provided for storage of XML data in Db2 tables. Most SQL statements support the XML data type. This enables you to perform many common database operations with XML data, such as creating tables with XML columns, adding XML columns to existing tables, creating indexes over XML columns, creating triggers on tables with XML columns, and inserting, updating, or deleting XML documents. You can update entire XML documents in an XML column, or update only portions of XML documents.

Alternatively, you can extract data items from an XML document and store those data items in columns of relational tables, using the SQL XMLTABLE built-in function in the INSERT via SELECT form of an INSERT statement.

XML document retrieval: You can use SQL to retrieve entire documents from XML columns, just as you retrieve data from any other type of column. When you need to retrieve portions of documents, you can specify XQuery expressions, through SQL with XML extensions (SQL/XML).

XML schema validation: XML schema validation is the process of determining whether the structure, content, and data types of an XML document are valid according to an XML schema. You can perform XML schema validation explicitly, by using the DSN_XMLVALIDATE function, or implicitly, if the XML column into which you insert XML documents has an XML type modifier.

Application development: Application development support of XML enables applications to combine XML and relational data access and storage. The following programming languages support the XML data type:

- Assembler
- C or C++ (embedded SQL or Db2 ODBC)
- COBOL
- Java™ (JDBC or SQLJ)
- PL/I

Database administration: Db2 for z/OS database administration support for pureXML includes the following items:

XML schema repository (XSR)
The XML schema repository (XSR) is a repository for all XML schemas that are required to validate and process XML documents that are stored in XML columns.

Utility support
Db2 for z/OS utilities support the XML data type. The storage structure for XML data and indexes is similar to the storage structure for LOB data and indexes. As with LOB data, XML data is not stored in the base table space, but it is stored in separate table spaces that contain only XML data. The XML table spaces also have their own index spaces. Therefore, the implications of using utilities for manipulating, backing up, and restoring XML data and LOB data are similar.

Performance: Indexing support is available for data stored in XML columns. The use of indexes over XML data can improve the efficiency of queries that you issue against XML documents. An XML index differs

© Copyright IBM Corp. 2007, 2020

from a relational index in that a relational index applies to an entire column, whereas an XML index applies to part of the data in a column. You indicate which parts of an XML column are indexed by specifying an XML pattern, which is a limited XPath expression.

pureXML data model

The pureXML data model follows the XPath 2.0 and the XQuery 1.0 data model. This data model provides an abstract representation of one or more XML documents or fragments.

The purpose of the data model is to define all permissible values of expressions in XQuery, including values that are used during intermediate calculations. Every XQuery expression takes as its input an instance of the data model and returns an instance of the data model. The pureXML data model is described in terms of sequences and items, atomic values, and nodes.

Related concepts

Overview of XQuery
XQuery is a functional programming language that was designed by the World Wide Web Consortium (W3C) to meet specific requirements for querying and modifying XML data.

Sequences and items

The XQuery data model is based on the notion of a sequence. The value of an XQuery expression is always a sequence. A *sequence* is an ordered collection of zero or more items. An *item* is either an atomic value or a node.

A sequence can contain nodes, atomic values, or any mixture of nodes and atomic values. For example, each of the following values can each be represented as a single sequence:

- `36`
- `<dog/>`
- `(2, 3, 4)`
- `(36, <dog/>, "cat")`
- `()`
- An XML document

A node can occur in more than one sequence, and a sequence can contain duplicate items. A sequence cannot be a member of another sequence. In other words, sequences cannot be nested. When two sequences are combined, the result is always a flattened sequence of nodes and atomic values. For example, appending the sequence (2, 3) to the sequence (3, 5, 6) results in the single sequence (3, 5, 6, 2, 3). Combining these sequences does not produce the sequence (3, 5, 6, (2, 3)) because nested sequences never occur.

A single item that appears on its own is modeled as a sequence that contains one item. For example, there is no distinction between the sequence (2) and the atomic value 2.

A sequence that contains zero items is called an *empty sequence*. Empty sequences can be used to represent missing or unknown information.

Related concepts

Atomic values
An *atomic value* is an instance of one of the built-in atomic data types that are defined by XML Schema.

Nodes
A *node* conforms to one of the types of nodes that are defined for XQuery. These node types include: document, element, attribute, text, processing instruction, comment, and namespace nodes.

Data model generation in XQuery

Before an XQuery expression can be processed, the input documents must be represented in the pureXML data model.

Atomic values

An *atomic value* is an instance of one of the built-in atomic data types that are defined by XML Schema.

These data types include strings, integers, decimals, dates, and other atomic types. These types are described as "atomic" because they cannot be subdivided. Some atomic types have literal values. For example, the following literals are atomic values:

- "this is a string"
- 45
- 1.44

Other atomic types have constructor functions to build atomic values out of strings. For example, the following constructor function builds a value of type xs:decimal out of the string "12.34":

```
xs:decimal("12.34")
```

Related concepts

Sequences and items
The XQuery data model is based on the notion of a sequence. The value of an XQuery expression is always a sequence. A *sequence* is an ordered collection of zero or more items. An *item* is either an atomic value or a node.

Nodes
A *node* conforms to one of the types of nodes that are defined for XQuery. These node types include: document, element, attribute, text, processing instruction, comment, and namespace nodes.

Data model generation in XQuery
Before an XQuery expression can be processed, the input documents must be represented in the pureXML data model.

Related reference

xs:decimal
The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

Nodes

A *node* conforms to one of the types of nodes that are defined for XQuery. These node types include: document, element, attribute, text, processing instruction, comment, and namespace nodes.

The nodes of a sequence form one or more trees that consist of a document node and all of the nodes that are reachable directly or indirectly from the document node. Every node belongs to exactly one tree, and every tree has exactly one document node. A tree whose root node is a document node is referred to as a *document*. A tree whose root node is not a document node is referred to as a *fragment*.

The following XML document includes a document element, named `product`, which contains a `description` element. The `product` element has an attribute named `pid` (purchase order ID). The `description` element contains elements named `name`, `details`, `price`, and `weight`.

```
<product xmlns="http://posample.org" pid="100-101-01">
 <description>
   <name>Snow Shovel, Deluxe 24"</name>
   <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic
curved handle with D-Grip</details>
   <price>19.99</price>
   <weight>2 kg</weight>
 </description>
</product>
```

The following figure shows a simplified representation of the data model for the previously described document. The figure includes a document node, element nodes, attribute nodes, and text nodes.

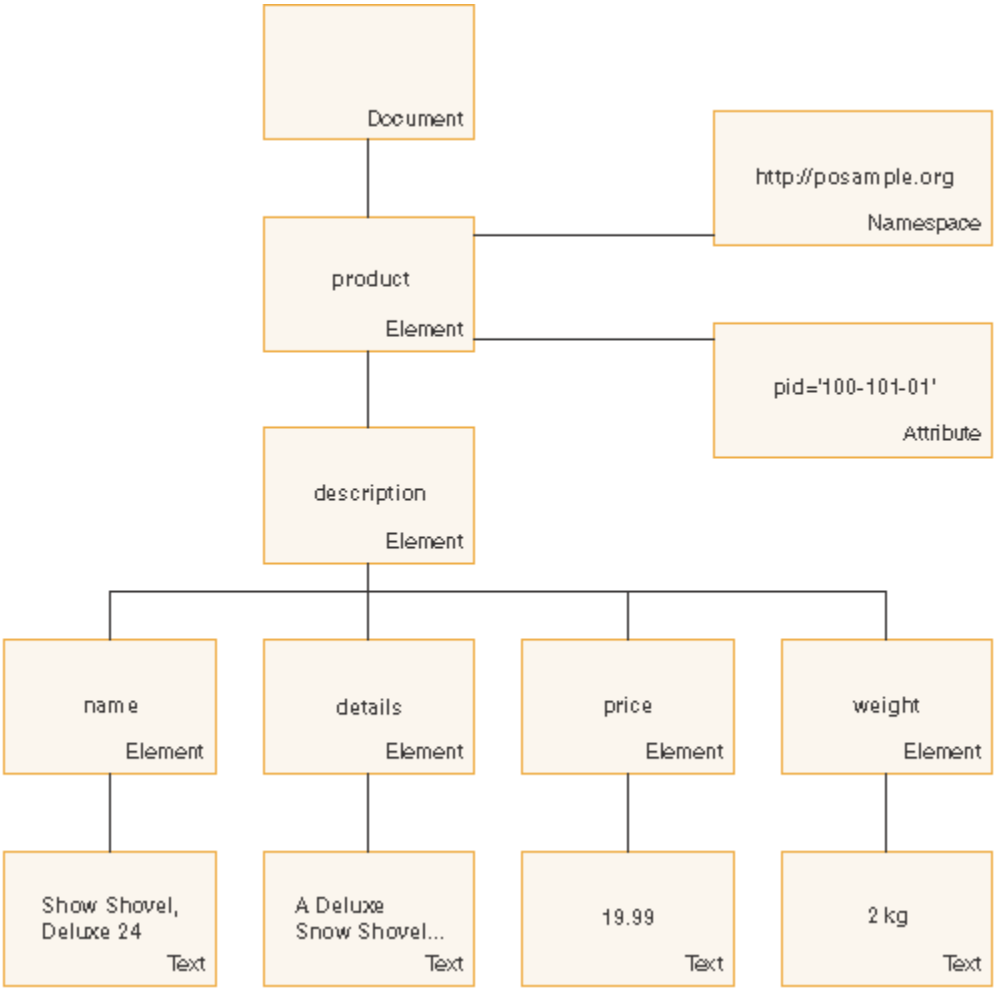


Figure 1. Data model diagram for document for a product

As the example illustrates, a node can have other nodes as children, thus forming one or more *node hierarchies*.

Node identity

Each node has a unique identity. This means that two nodes are distinguishable even though their names and values might be the same. In contrast, atomic values do not have an identity. Every instance of an atomic value (for example, the integer 7) is identical to every other instance of that value.

Document order

Among all of the nodes in a hierarchy, there is a total ordering called *document order*, in which each node appears before its children. Document order corresponds to the order in which the nodes appear when the node hierarchy is represented in XML format:

- The document node is the first node.
- Element nodes occur before their children.
- Namespace nodes immediately follow the element node with which they are associated.
- Attribute nodes occur after namespace nodes, or their associated element node, if no namespace nodes exist.

  Attribute nodes and namespace nodes are not children of an element node, but the associated element node is their parent node.

The relative order of attribute nodes is arbitrary, but this order does not change during the processing of an XQuery expression.

- Element nodes, text nodes, processing instruction nodes, and comment nodes can be children of an element node or a document node.
- The relative order of siblings is determined by their order in the node hierarchy.
- Children and descendants of a node occur before siblings that follow the node.

Node properties

Each node has *properties* that describe characteristics of that node. For example, a node's properties might include the name of the node, its children, its parent, its attributes, and other information that describes the node. The node kind determines which properties are present for specific nodes.

A node can have one or more of the following properties:

node name
The name of the node (expressed as a QName).

parent
The node that is the parent of the current node.

type name
The dynamic (run time) type of the node.

children
The sequence of nodes that are *children* of the current node.

attributes
The set of attribute nodes that belong to the current node.

string value
A string value that can be extracted from the node.

typed value
A sequence of zero or more atomic values that can be extracted from the node.

target
Identifies the application to which a processing instruction is directed. The target is an NCName (local name with no colons).

content
The content of a processing instruction, text node, or comment node.

Related concepts

Sequences and items
The XQuery data model is based on the notion of a sequence. The value of an XQuery expression is always a sequence. A *sequence* is an ordered collection of zero or more items. An *item* is either an atomic value or a node.

Atomic values
An *atomic value* is an instance of one of the built-in atomic data types that are defined by XML Schema.

Data model generation in XQuery
Before an XQuery expression can be processed, the input documents must be represented in the pureXML data model.

Document nodes

A document node encapsulates an XML document.

A document node cannot have parent nodes and can have zero or more child nodes. The child nodes can include element nodes, text nodes, processing instruction nodes, or comment nodes. To be a well-formed document, the document node must have exactly one child element node and no child text nodes.

A document node has the following node properties:

- children

- string value
- typed value

For a document node, the string value is the concatenation of all of the string values of all of its descendent text nodes, in document order, and the typed value is the same as the string value of type xs:untypedAtomic.

For example, suppose that a document has the following textual representation:

```
<product xmlns="http://posample.org" pid="100-101-01">
 <description>
   <name>Snow Shovel, Deluxe 24"</name>
   <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic
curved handle with D-Grip</details>
   <price>19.99</price>
   <weight>2 kg</weight>
 </description>
</product>
```

The document node has the following property values:

Table 1. Properties of the document node

Node property	Value	Value type
children	product node	
string value	"Snow Shovel, Deluxe 24"A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with D-Grip19.992 kg"	
typed value	"Snow Shovel, Deluxe 24"A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with D-Grip19.992 kg"	xs:untypedAtomic

Related concepts

Element nodes
An element node encapsulates an XML element.

Attribute nodes
An *attribute node* represents an XML attribute.

Text nodes
A text node encapsulates XML character content.

Processing instruction nodes
A processing instruction node encapsulates an XML processing instruction.

Comment nodes
A comment node encapsulates XML comments.

Element nodes

An element node encapsulates an XML element.

An element can have zero or one parent, and zero or more children. The children can include element nodes, processing instruction nodes, comment nodes, and text nodes. Document and attribute nodes are never children of element nodes. However, an element node is considered to be the parent of its attributes. The attributes of an element node must have unique QNames.

An element node has the following node properties:

- node name
- parent
- type name (The type name of an element node in Db2 is always xs:untyped.)

- children
- attributes
- string value
- typed value
- in-scope namespaces

For an element node, the string value is the concatenation of the string values of all of its text node descendants in document order. If the element is empty, the string value is the empty string "". The typed value of an element is one of the following values:

- If the element can be null, the typed value is ().
- If the element is empty, the typed value is the empty sequence ().
- Otherwise, the typed value is its string value as type xs:untypedAtomic.

For example, suppose that a document has the following textual representation:

```
<product xmlns="http://posample.org" pid="100-101-01">
 <description>
   <name>Snow Shovel, Deluxe 24"</name>
   <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic
curved handle with D-Grip</details>
   <price>19.99</price>
   <weight>2 kg</weight>
 </description>
</product>
```

The product element node has the following property values:

Table 2. Properties of the product node

Node property	Value	Value type
node name	product	
parent	document node	
type name	xs:untyped	
children	description node	
attributes	pid	
string value	"Snow Shovel, Deluxe 24"A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with D-Grip19.992 kg"	
typed value	"Snow Shovel, Deluxe 24"A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with D-Grip19.992 kg"	xs:untypedAtomic
in-scope namespaces	(default, http://posample.org)	

The name element node has the following property values:

Table 3. Properties of the name node

Node property	Value	Value type
node name	name	
parent	description node	
type name	xs:untyped	

Table 3. Properties of the name node (continued)

Node property	Value	Value type
children	text node "Snow Shovel, Deluxe 24" "	
attributes	none	
string value	"Snow Shovel, Deluxe 24" "	
typed value	"Snow Shovel, Deluxe 24" "	xs:untypedAtomic
in-scope namespaces	(default, http://posample.org)	

Related concepts

Document nodes
A document node encapsulates an XML document.

Attribute nodes
An *attribute node* represents an XML attribute.

Text nodes
A text node encapsulates XML character content.

Processing instruction nodes
A processing instruction node encapsulates an XML processing instruction.

Comment nodes
A comment node encapsulates XML comments.

Attribute nodes

An *attribute node* represents an XML attribute.

An attribute node can have zero or one parent. The element node that owns an attribute is considered to be its parent, even though an attribute node is not a child of its parent element.

An attribute node has the following node properties:

- node name
- parent
- type name (The type name of an attribute node in Db2 is always xs:untypedAtomic.)
- string value
- typed value

For an attribute node, the string value is the normalized value of the attribute or schema normalized value of the attribute if the attribute was validated with a schema. The typed value is the same as the string value of type xs:untypedAtomic.

For example, suppose that a document has the following textual representation:

```
<product xmlns="http://posample.org" pid="100-101-01">
 <description>
   <name>Snow Shovel, Deluxe 24"</name>
   <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic
curved handle with D-Grip</details>
   <price>19.99</price>
   <weight>2 kg</weight>
 </description>
</product>
```

The pid attribute has the following property values:

Table 4. Properties of the pid attribute node

Node property	Value	Value type
node name	pid	
parent	product node	
type name	xs:untypedAtomic	
string value	"100-101-01"	
typed value	100-101-01"	xs:untypedAtomic

Related concepts

Document nodes
A document node encapsulates an XML document.

Element nodes
An element node encapsulates an XML element.

Text nodes
A text node encapsulates XML character content.

Processing instruction nodes
A processing instruction node encapsulates an XML processing instruction.

Comment nodes
A comment node encapsulates XML comments.

Data model generation in XQuery
Before an XQuery expression can be processed, the input documents must be represented in the pureXML data model.

Text nodes

A text node encapsulates XML character content.

A text node can have zero or one parent. The content of a text node can be empty. However, unless the parent of a text node is empty, the content of the text node cannot be an empty string. Text nodes that are children of a document or element node never appear as adjacent siblings. During document or element node construction, any adjacent siblings are combined into a single text node. If the resulting text node is empty, it is discarded.

Text nodes have the following node properties:

- content
- parent

For example, suppose that a document has the following textual XML representation:

```
<product xmlns="http://posample.org" pid="100-101-01">
 <description>
   <name>Snow Shovel, Deluxe 24"</name>
   <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic
curved handle with D-Grip</details>
   <price>19.99</price>
   <weight>2 kg</weight>
 </description>
</product>
```

The text node beneath the name element node has the following property values:

Table 5. Properties of the name text node

Node property	Value
content	Snow Shovel, Deluxe 24"

Table 5. Properties of the name text node (continued)

Node property	Value
parent	name

The string value of a text node is the content of the node, which in the preceding example is " Snow Shovel, Deluxe 24" ." The typed value of a text node is the same value as type xs:untypedAtomic.

Related concepts

Document nodes
A document node encapsulates an XML document.

Element nodes
An element node encapsulates an XML element.

Attribute nodes
An *attribute node* represents an XML attribute.

Processing instruction nodes
A processing instruction node encapsulates an XML processing instruction.

Comment nodes
A comment node encapsulates XML comments.

Related reference

xs:untypedAtomic
The data type `xs:untypedAtomic` serves as a special type annotation to indicate atomic values that have not been validated by an XML schema or a DTD.

Processing instruction nodes

A processing instruction node encapsulates an XML processing instruction.

A processing instruction node can have zero or one parent. The target of a processing instruction must be an NCName (a local name with no colons).

A processing instruction node has the following node properties:

- target
- content
- parent

For example, consider the following processing instruction:

```
<?xml-stylesheet href="book.css" type="text/css"?>
```

This processing instruction has the following property values:

Table 6. Properties of the processing instruction node

Node property	Value
target	xml-stylesheet
content	href="book.css" type="text/css"
parent	document node

The string value of a processing instruction node is the content of the node, which in this case is `href="book.css" type="text/css"`. The typed value is the same value as type xs:string.

Related concepts

Document nodes

A document node encapsulates an XML document.

Element nodes
An element node encapsulates an XML element.

Attribute nodes
An *attribute node* represents an XML attribute.

Text nodes
A text node encapsulates XML character content.

Comment nodes
A comment node encapsulates XML comments.

Comment nodes
A comment node encapsulates XML comments.

A comment node can have zero or one parent.

A comment node has the following node properties:

- content
- parent

For example, consider the following comment:

```
<ID>
<!-- This element contains an ID number. -->
101
</ID>
```

This comment has the following property values:

Table 7. Properties of the comment node

Node property	Value
content	This element contains an ID number.
parent	ID node

The string value of a comment node is the content of the node, which in the case of the preceding example is `This element contains an ID number.` The typed value is the same value as type xs:string.

Related concepts
Document nodes
A document node encapsulates an XML document.

Element nodes
An element node encapsulates an XML element.

Attribute nodes
An *attribute node* represents an XML attribute.

Text nodes
A text node encapsulates XML character content.

Processing instruction nodes
A processing instruction node encapsulates an XML processing instruction.

Data model generation in XQuery

Before an XQuery expression can be processed, the input documents must be represented in the pureXML data model.

An input XML document is transformed into an instance of the pureXML data model through a process called *XML parsing*. Alternatively, you can generate an instance of the pureXML data model by using SQL

XML constructors, such as XMLELEMENT and XMLATTRIBUTES. These built-in functions enable you to generate XML data from relational data. Likewise, the result of an XQuery expression (an instance of the pureXML data model) can be transformed into an XML representation through a process called *XML serialization*.

- During *XML parsing*, the string representation of an XML document is transformed into an instance of the XQuery model. Optionally, the XML document can be validated against a specific schema. The parsed data is represented as a hierarchy of nodes and atomic values. Each atomic value, element node, and attribute node in the XQuery data model is annotated with a dynamic type. The dynamic type specifies a range of values. For example, an attribute named `version` might have the dynamic type `xs:decimal` to indicate that the attribute contains a decimal value.

  Restriction: If the XML document is validated against a schema, Db2 does not keep the type annotation for each node. The data is stored as untyped.

  The value of an attribute is represented directly within the attribute node. An attribute node of unknown type is annotated with the dynamic type `xs:untypedAtomic`.

  The value of an element is represented by the children of the element node, which might include text nodes and other element nodes. The dynamic type of an element node indicates how the values in the child text nodes are to be interpreted. All element nodes have the type `xs:untyped`.

  An atomic value of unknown type is annotated with the type `xs:untypedAtomic`.

  If an input document has no schema, the document is not validated. Db2 assigns nodes and atomic values as untyped (xs:untyped or xs:untypedAtomic).
- During *serialization*, the sequence of nodes and atomic values (the instance of the XQuery data model) is converted into its string representation. The result of serialization does not always represent a well-formed document. In fact, serialization can result in a single atomic value (for example, 17) or a sequence of elements that do not have a common parent.

Related concepts

Sequences and items
The XQuery data model is based on the notion of a sequence. The value of an XQuery expression is always a sequence. A *sequence* is an ordered collection of zero or more items. An *item* is either an atomic value or a node.

Atomic values
An *atomic value* is an instance of one of the built-in atomic data types that are defined by XML Schema.

Nodes
A *node* conforms to one of the types of nodes that are defined for XQuery. These node types include: document, element, attribute, text, processing instruction, comment, and namespace nodes.

XML parsing
XML parsing is the process of converting XML data from its textual XML format to its hierarchical format.

XML serialization
XML serialization is the process of converting XML data from its internal representation in a Db2 table to the textual XML format that it has in an application.

Comparison of the XML model and the relational model

When you design your databases, you need to decide whether your data is better suited to the XML model or the relational model.

This topic discusses some of the factors that you need to consider as you make this decision.

The major differences between XML data and relational data are:

- XML data is hierarchical; relational data has a flat structure.

  An XML document contains information about the relationship of data items to each other in the form of the hierarchy. With the relational model, the only types of relationships that can be defined are parent table and dependent table relationships.

- XML data is self-describing; relational data is not.

  An XML document contains not only the data, but also tagging for the data that explains what it is. A single document can have different types of data. With the relational model, the content of the data is defined by its column definition. All data in a column must have the same type of data.

- XML data has inherent ordering; relational data does not.

  For an XML document, the order in which data items are specified is assumed to be the order of the data in the document. There is often no other way to specify order within the document. For relational data, the order of the rows is not guaranteed unless you specify an ORDER BY clause on one or more columns.

Sometimes the nature of the data dictates the way in which you store it. For example, if the data is naturally hierarchical and self-describing, you might store it as XML data. However, other factors might influence your decision about which model to use.

Some of those factors are:

- Whether maximum flexibility of the data is needed

  Relational tables are fairly rigid. For example, normalizing one table into many or denormalizing many tables into one can be very difficult. If the data design changes often, representing it as XML data is a better choice.

- Whether maximum performance for data retrieval is needed

  Some expense is associated with serializing and interpreting XML data. Retrieval of a few items from a large XML document is relatively expensive, so performance might be better for data in a relational format. However, for retrieval of entire documents, XML data might be more efficient if a large number of relational joins are needed to retrieve equivalent data in a relational format.

- Whether the data is processed later as relational data

  If subsequent processing of the data depends on the data being stored in a relational database, it might be appropriate to store parts of the data as relational, using decomposition. An example of this situation is when online analytical processing (OLAP) is applied to the data in a data warehouse. Also, if other processing is required on the XML document as a whole, then storing some of the data as relational as well as storing the entire XML document might be a suitable approach in this case.

- Whether the data components have meaning outside a hierarchy

  Data might be inherently hierarchical in nature, but the child components do not need the parents to provide value. For example, a purchase order might contain part numbers. The purchase orders with the part numbers might be best represented as XML documents. However, each part number has a part description associated with it. It might be better to include the part descriptions in a relational table, because the relationship between the part numbers and the part descriptions is logically independent of the purchase orders in which the part numbers are used.

- Whether data attributes apply to all data, or to only a small subset of the data

  Some sets of data have a large number of possible attributes, but only a small number of those attributes apply to any particular data value. For example, in a retail catalog, there are many possible data attributes, such as size, color, weight, material, style, weave, power requirements, or fuel requirements. For any given item in the catalog, only a subset of those attributes is relevant: power requirements are meaningful for a table saw, but not for a coat. This type of data is difficult to represent and search with a relational model, but relatively easy to represent and search with an XML model.

- Whether referential integrity is required

  XML columns cannot be defined as part of referential constraints. Therefore, if values in XML documents need to participate in referential constraints, you should store the data as relational data.

- Whether the data needs to be updated often

  Currently, you can update XML data in an XML column only by replacing full documents. If you need to frequently update small fragments of very large documents for a large number of rows, it can be more

efficient to store the data in non-XML columns. If, however, you are updating small documents and only a few documents at a time, storing as XML can be efficient as well.

XML data type

The XML data type is used to define columns of a table that store XML values. This data type provides the ability to store well-formed XML documents in a database.

All XML data is stored in the database in an internal representation. Character data in this internal representation is in the UTF-8 encoding scheme. The internal representation of values in an XML column is not a string and not directly comparable to string values.

An XML value can be transformed into a textual XML value that represents the XML document in the following ways:

- By using the XMLSERIALIZE function
- By retrieving the value into an application variable of an XML, string, or binary type

Similarly, a textual XML value that represents an XML document can be transformed to an XML value by using the XMLPARSE function or by storing a value from a string, binary, or XML application data type in an XML column.

A binary XML value is a value that is in the Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format. This format is an external representation of an XML value that is only used for exchange with a Db2 client application or the UNLOAD or LOAD utilities. The binary representation provides more efficient XML parsing. An XML value can be transformed into a binary XML value that represents the XML document in the following ways:

- In a JDBC or SQLJ application, by retrieving the XML column value into an `java.sql.SQLXML` object, and then retrieving the data from the `java.sql.SQLXML` object as a binary data type, such as `InputStream`. JDBC 4.0 or later provides support for the `java.sql.SQLXML` data type.
- In an ODBC application, by binding the XML column to an application variable with the SQL_C_BINARYXML data type, and retrieving the XML value into that application variable.
- By running the UNLOAD utility, and using one of the following field specifications for the XML output:

```
CHAR BLOBF template-name BINARYXML
VARCHAR BLOBF template-name BINARYXML
XML BINARYXML
```

Similarly, a binary value that represents an XML document can be transformed to an XML value in the following ways:

- In a JDBC or SQLJ application, by assigning the input value to an `java.sql.SQLXML` object, and then inserting the data from the `java.sql.SQLXML` object into the XML column.
- In an ODBC application, by binding a parameter marker for input to an XML column to an application variable with the SQL_C_BINARYXML data type, and inserting the binary data into the XML column.
- By running the LOAD utility, and using one of the following field specifications for the XML input:

```
CHAR BLOBF BINARYXML
VARCHAR BLOBF BINARYXML
XML BINARYXML
```

The size of an XML value in a Db2 table has no architectural limit. However, textual XML data that is stored in or retrieved from an XML column is limited to 2 GB.

Validation of an XML document against an XML schema is supported. XML schema validation is typically performed during INSERT or UPDATE into an XML column. If an XML column has an XML type modifier, only documents that are valid according to the XML schema that is specified by the XML type modifier can be inserted into the column. If an XML column does not have an XML type modifier, validation is optional.

Related concepts

Creation of tables with XML columns

To create tables with XML columns, you specify columns with the XML data type in the CREATE TABLE statement. A table can have one or more XML columns.

XML parsing
XML parsing is the process of converting XML data from its textual XML format to its hierarchical format.

XML serialization
XML serialization is the process of converting XML data from its internal representation in a Db2 table to the textual XML format that it has in an application.

Tutorial: Working with XML data

pureXML lets you define table columns that store a single, well-formed XML document in each row. This tutorial demonstrates how to set up a Db2 database system to store XML data and how to perform basic operations with XML data.

Before you begin

- Start a SPUFI session, or create a DSNTEP2 job that you can use to issue the SQL statements in these exercises.
- Set the SQL terminator to a character other than a semicolon, such as the number sign (#), so that SQL statements can contain embedded semicolons.
- If you use SPUFI, also change the following settings:
  - Set CAPS OFF so that the ISPF editor does not change input to uppercase.
  - On the CURRENT SPUFI DEFAULTS panel, change MAX CHAR FIELD to 32767, so that you will be able to see complete XML documents.

Procedure

1. Create a table named MYCUSTOMER that contains an XML column:

```
CREATE TABLE MYCUSTOMER (Cid BIGINT, INFO XML)#
```

2. Create an index over XML data. For the purposes of this tutorial, all XML documents that you store in the INFO column have a root element named *customerinfo* with an attribute named *Cid*. Create a unique index on the *Cid* attribute:

```
CREATE UNIQUE INDEX MYCUT_CID_XMLIDX ON MYCUSTOMER(INFO)
GENERATE KEY USING XMLPATTERN
  'declare default element namespace "http://posample.org"; /customerinfo/@Cid'
AS SQL DECFLOAT#
```

   The XML pattern that defines the index is case-sensitive. The element and attribute names in the XML pattern must match the element and attribute names in the XML documents exactly. In this example, customerinfo is the element and Cid is the attribute.

3. Insert three XML documents into the MYCUSTOMER table that you created in step 1 by issuing the following INSERT statements:

```
INSERT INTO MYCUSTOMER (CID, INFO) VALUES (1000,
'<customerinfo xmlns="http://posample.org" Cid="1000">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>5 Rosewood</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>M6W 1E6</pcode-zip>
  </addr>
  <phone type="work">416-555-1358</phone>
</customerinfo>')#

INSERT INTO MYCUSTOMER (CID, INFO) VALUES (1002,
'<customerinfo xmlns="http://posample.org" Cid="1002">
  <name>Jim Noodle</name>
  <addr country="Canada">
```

```
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
</customerinfo>')#

INSERT INTO MYCUSTOMER (CID, INFO) VALUES (1003,
'<customerinfo xmlns="http://posample.org" Cid="1003">
  <name>Robert Shoemaker</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-2937</phone>
</customerinfo>')#
```

You can confirm that the records were successfully inserted by issuing the following query:

```
SELECT CID, INFO FROM MYCUSTOMER#
```

4. Update the XML documents that are stored in an XML column. Issue the following UPDATE statement to add a cell phone number to the XML document for which the CID column value is 1002. To change individual items in an XML column, you must replace the entire column.

```
UPDATE MYCUSTOMER SET INFO =
'<customerinfo xmlns="http://posample.org" Cid="1002">
  <name>Jim Noodle</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
  <phone type="cell">905-554-7254</phone>
 </customerinfo>'
WHERE CID=1002#
```

You can confirm that the XML document was updated by issuing the following query:

```
SELECT CID, INFO FROM MYCUSTOMER
 WHERE CID=1002#
```

5. Delete any rows from the MYCUSTOMER table for which the customer document in the INFO column contains a cell phone number. Issue the following DELETE statement, which contains an XMLEXISTS predicate with an XQuery expression, to specify the documents that you want to delete.

```
DELETE FROM MYCUSTOMER
WHERE XMLEXISTS (
 'declare default element namespace "http://posample.org";
 /customerinfo/phone[@type="cell"]' PASSING INFO)#
```

You can confirm that the XML document was deleted by issuing the following query:

```
SELECT COUNT(*) FROM MYCUSTOMER
WHERE CID = 1002#
```

Confirm that the resulting value is 0.

6. Query XML data.

You can retrieve an entire XML document, or you can retrieve a portion of an XML document.

- Issue the following SELECT statement to retrieve the entire XML document that has a CID value of 1000:

```
SELECT CID, INFO FROM MYCUSTOMER
 WHERE CID=1000#
```

- Issue the following SELECT statement with the XMLQUERY function to retrieve the name element from each XML document in the MYCUSTOMER table.

```
SELECT XMLQUERY (
  'declare default element namespace "http://posample.org";
   for $d in $doc/customerinfo
   return <out>{$d/name}</out>'
   passing INFO as "doc")
  FROM MYCUSTOMER as c
  WHERE XMLEXISTS ('declare default element namespace "http://posample.org";
   $i/customerinfo/addr[city="Toronto"]' passing c.INFO as
"i")#
```

The SELECT statement returns the following result:

```
<out xmlns="http://posample.org"><name>Kathy Smith</name></out>
```

7. Update part of an XML document. Issue the following UPDATE statement, which includes the XMLMODIFY function, to change the address from 5 Rosewood to 42 Rosedale for the customer with customer ID 1000 in the MYCUSTOMER table.

```
UPDATE MYCUSTOMER
 SET INFO = XMLMODIFY(
  'declare default element namespace "http://posample.org";
  replace value of node /customerinfo/addr/street
  with "42 Rosedale"')
 WHERE CID=1000#
```

You can confirm that the XML document was updated by issuing the following query:

```
SELECT CID, INFO FROM MYCUSTOMER
 WHERE CID = 1000#
```

Related information:

“Creation of tables with XML columns” on page 27
“Deletion of rows with XML documents from tables” on page 38
Chapter 3, “XML data indexing,” on page 81
“Insertion of rows with XML column values” on page 33
“Retrieving XML data” on page 60
“Updates of XML columns” on page 35
DSNTEP2 and DSNTEP4 (Db2 Application programming and SQL)
Executing SQL by using SPUFI (Db2 Application programming and SQL)

Prerequisites for using pureXML

If you do not do XML schema validation using Db2, use of XML requires no special prerequisites. However, if you do XML schema validation using Db2, you need to install extra software.

If you plan to validate XML documents against XML schemas by using the DSN_XMLVALIDATE function or by using an XML schema type modifier on XML columns, you need an XML schema repository.

Related tasks

Additional steps for enabling the stored procedures and objects for XML schema support (Db2 Installation and Migration)
Installing the IBM Data Server Driver for JDBC and SQLJ as part of a Db2 installation (Db2 Application Programming for Java)

Setting up the XML schema repository

Before you can do XML schema validation on your XML documents, you need to set up an XML schema repository.

Before you begin

Besides Db2 for z/OS, XML schema repository setup requires that the following software is installed and configured:

- Workload Manager for z/OS (WLM)
- z/OS XML System Services
- Java 2 Technology Edition, V5 or later, 31-bit version
- IBM Data Server Driver for JDBC and SQLJ

About this task

Setting up an XML schema repository involves defining a set of Db2 tables and indexes that store XML schema information, and setting up a set of stored procedures that operate on the XML schemas that are stored in the tables.

Procedure

To set up the XML schema repository:

1. Define the XML schema repository tables and indexes.

   The table spaces are:

   - SYSIBM.SYSXSR
   - SYSIBM.SYSXSRA1
   - SYSIBM.SYSXSRA2
   - SYSIBM.SYSXSRA3
   - SYSIBM.SYSXSRA4

   The tables are:

   - SYSIBM.XSRANNOTATIONINFO
   - SYSIBM.XSRCOMPONENT
   - SYSIBM.XSROBJECTCOMPONENTS
   - SYSIBM.XSROBJECTGRAMMAR
   - SYSIBM.XSROBJECTHIERARCHIES
   - SYSIBM.XSROBJECTPROPERTY
   - SYSIBM.XSROBJECTS
   - SYSIBM.XSRPROPERTY

   The indexes are:

   - SYSIBM.XSRANNINFOIDX
   - SYSIBM.XSRCOMP01
   - SYSIBM.XSRCOMP02
   - SYSIBM.XSRHIER01
   - SYSIBM.XSRHIER02
   - SYSIBM.XSROBJ01
   - SYSIBM.XSROBJ02
   - SYSIBM.XSROBJ03

- SYSIBM.XSROBJ04
- SYSIBM.XSRXCC01
- SYSIBM.XSRXCP01
- SYSIBM.XSRXOG01
- SYSIBM.XSRXOP01

Installation job DSNTIJRT invokes a program that executes the CREATE DATABASE, CREATE TABLESPACE, CREATE TABLE and CREATE INDEX statements for the XML schema repository tables and indexes. After the installation process customizes job DSNTIJRT, you can run DSNTIJRT without further modification to create those tables and indexes.

Important: Do not drop these objects after you begin to do XML schema validation. Doing so can cause unexpected behavior.

2. Define the WLM environment and startup procedure for the C language XML schema repository stored procedures.
3. Define the WLM environment and startup procedure for the Java language XML schema repository stored procedure.
4. Define the XML schema repository stored procedures to Db2.
5. Bind the packages for the XML schema repository stored procedures.

   Installation job DSNTIJRT invokes a program that binds the packages for the XML schema repository stored procedures. After the installation process customizes job DSNTIJRT, you can run DSNTIJRT without further modification to bind the packages.
6. Bind the packages for the IBM Data Server Driver for JDBC and SQLJ.
7. Test the XML schema repository setup.

Defining the WLM environment and JCL startup procedure for C language XML schema repository stored procedures

The XML schema validation stored procedures that are written in C require their own WLM environment and a JCL procedure for starting that WLM environment.

About this task

The C language stored procedures that use this WLM environment and JCL procedure are XSR_ADDSCHEMADOC, XSR_REGISTER, and XSR_REMOVE.

One of the tasks that installation job DSNTIJRW performs is to call a program that installs a WLM environment with the default name of DSNWLM_XML. The installation process configures DSNWLM_XML so that you can use it to run the XML schema repository stored procedures. One of the tasks that installation job DSNTIJMV performs is to install a WLM startup procedure named *ssnm*WLMX for that WLM environment.

Follow this process if the predefined settings do not work for you, and you need to make changes to the WLM environment or startup procedure.

Procedure

To define the WLM environment and JCL startup procedure for C language XML schema repository stored procedures:

1. In a TSO session, start the IWMARIN0 utility.

   For example, in the ISPF Command Shell, type:

   ```
   EXEC 'SYS1.SBLSCLI0(IWMARIN0)'
   ```
2. In the WLM ISPF Choose Service Definition menu, choose option 1 or option 2, depending on your current WLM service definition setup. See the WLM documentation on service definitions for details.
3. In the WLM ISPF Definition Menu panel, choose option 9: Application Environments.

4. In the Application Environment Selection List panel, type 3 (Modify) next to DSNWLM_XML.
5. In the Modify an Application Environment panel, you see values like these.

```
                    Modify an Application Environment
Command ===> ____________________________________________________________

Application Environment Name . : DSNWLM_XML
Description  . . . . . . . . . . DB2-SUPPLIED WLM ENVIRONMENT
Subsystem Type . . . . . . . . . DB2
Procedure Name . . . . . . . . . DSNWLMX
Start Parameters . . . . . . . . DB2SSN=&IWMSSNM,APPLENV='DSNWLM_XML'
                                 ______________________________________
                                 ___________________________________

Starting of server address spaces for a subsystem instance:
1   1.  Managed by WLM
    2.  Limited to a single address space per system
    3.  Limited to a single address space per sysplex
```

The meanings of the parameters are:

Subsystem Type
Specify DB2.

Procedure Name
Specify a name that matches the name of the JCL startup procedure for the stored procedure address spaces that are associated with this application environment.

Start Parameters
If the Db2 subsystem in which the stored procedure runs is not in a Sysplex, specify a DB2SSN value that matches the name of that Db2 subsystem. If the same JCL is used for multiple Db2 subsystems, specify DB2SSN=&IWMSSNM. Specify an APPLENV value that matches the value that you specify in the Application Environment Name field.

Starting of server address spaces for a subsystem instance
Specify 1 (Managed by WLM) or 2 (Limited to a single address space per system).

6. Modify the JCL startup procedure for the stored procedure address spaces that are associated with the WLM application environment.

The default JCL startup procedure for DSNWLM_XML looks like this one.

```
//DSNWLMX PROC APPLENV=DSNWLM_XML,                          1
// DB2SSN=DSN,RGN=0K,NUMTCB=40                              2
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
// PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD DISP=SHR,DSN=CEE.SCEERUN                       3
// DD DISP=SHR,DSN=DSNB10.SDSNEXIT
// DD DISP=SHR,DSN=DSNB10.SDSNLOAD
//CEEDUMP DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
```

Notes:

1 In this line, the procedure name must be the same as the Procedure Name value in the Create an Application Environment or Modify an Application Environment panel. The APPLENV value must be the same as the Application Environment Name value in the Create an Application Environment or Modify an Application Environment panel. If the Start Parameters field in the Create an Application Environment or Modify an Application Environment panel contains an APPLENV parameter, the APPLENV parameter value in the Start Parameters field value overrides the value in the JCL startup procedure. NUMTCB should be between 40 and 60.

2 In this line, the DB2SSN value must be the same as your Db2 for z/OS subsystem name. NUMTCB should be between 40 and 60.

3 STEPLIB specifies the data sets that are necessary for running the stored procedures. At a minimum, you need the Language Environment® run time data set, SCEERUN, and the SDSNLOAD data set, which contains the DSNX9WLM program and the load modules for the C language XML schema repository stored procedures.

Related tasks

Setting up the Db2 core WLM environments during installation (Db2 Installation and Migration)

Related information

Setting up a Service Definition (z/OS MVS Planning: Workload Management)

Defining the WLM environment and JCL startup procedure for the Java language XML schema repository stored procedure

The XML schema validation stored procedure, XSR_COMPLETE, which is written in Java, can share a WLM environment with other Java routines. You need a JCL procedure that is tailored for starting that WLM environment.

About this task

One of the tasks that installation job DSNTIJRW performs is to call a program that installs a WLM environment with the default name of DSNWLM_JAVA. The installation process configures DSNWLM_JAVA so that you can use it to run the XML schema repository stored procedure XSR_COMPLETE. One of the tasks that installation job DSNTIJMV performs is to install a WLM startup procedure named *ssnm*WLMJ for that WLM environment.

You only need to follow this process if the predefined settings do not work for you, and you need to make changes to the WLM environment or startup procedure.

Procedure

To set up the WLM environment and JCL startup procedure for XSR_COMPLETE, follow these steps:

1. In a TSO session, start the IWMARIN0 utility.

   For example, in the ISPF Command Shell, type:

   ```
   EXEC 'SYS1.SBLSCLI0(IWMARIN0)'
   ```

2. In the WLM ISPF Choose Service Definition menu, choose option 1 or option 2, depending on your current WLM service definition setup. See the WLM documentation on service definitions for details.
3. In the Application Environment Selection List panel, type 3 (Modify) next to DSNWLM_JAVA.
4. In the WLM ISPF Definition Menu panel, choose option 9: Application Environments.
5. In the Modify an Application Environment panel, you see values like these.

```
  Application-Environment  Notes  Options  Help
------------------------------------------------------------------------------
                    Modify an Application Environment
Command ===> ____________________________________________________________

Application Environment Name . : DSNWLM_JAVA
Description  . . . . . . . . . . Environment for Java procedures
Subsystem Type . . . . . . . . . DB2
Procedure Name . . . . . . . . . DSNWLMJ
Start Parameters . . . . . . . . DB2SSN=&IWMSSNM,APPLENV='DSNWLM_JAVA'
                                 ----------------------------------------

Starting of server address spaces for a subsystem instance:
1   1.  Managed by WLM
    2.  Limited to a single address space per system
    3.  Limited to a single address space per sysplex
```

Subsystem Type

Specify DB2.

Procedure Name
Specify a name that matches the name of the JCL startup procedure for the stored procedure address spaces that are associated with this application environment.

Start Parameters
If the Db2 subsystem in which the stored procedure runs is not in a Sysplex, specify a DB2SSN value that matches the name of that Db2 subsystem. If the same JCL is used for multiple Db2 subsystems, specify DB2SSN=&IWMSSNM. Specify an APPLENV value that matches the value that you specify in the Application Environment Name field.

Starting of server address spaces for a subsystem instance
Specify 1 (Managed by WLM) or 2 (Limited to a single address space per system).

6. Modify the pre-defined JCL startup procedure for the stored procedure address spaces that are associated with the WLM application environment.

   The installation process creates a JCL startup procedure similar to this one.

```
//DSNWLMJ PROC APPLENV=DSNWLM_JAVA,                                  1
// DB2SSN=DSN,RGN=0K,NUMTCB=5                                        2
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,                3
//   PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD DISP=SHR,DSN=CEE.SCEERUN                                4
// DD DISP=SHR,DSN=DSNB10.SDSNEXIT
// DD DISP=SHR,DSN=DSNB10.SDSNLOAD
// DD DISP=SHR,DSN=DSNB10.SDSNLOD2
//JAVAENV DD DISP=SHR,                                               5
//   DSN=DSNB10.DSNWLMJ.JAVAENV
//JSPDEBUG DD SYSOUT=*                                               6
//JAVAOUT  DD PATH='/tmp/javaout.txt',                               7
//   PATHOPTS=(ORDWR,OCREAT,OAPPEND),
//   PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
//JAVAERR  DD PATH='/tmp/javaerr.txt',
//   PATHOPTS=(ORDWR,OCREAT,OAPPEND),
//   PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
```

   Notes:

   1 In this line, the procedure name must be the same as the Procedure Name value in the Create an Application Environment or Modify an Application Environment panel. The APPLENV value must be the same as the Application Environment Name value in the Create an Application Environment or Modify an Application Environment panel. If the Start Parameters field in the Create an Application Environment or Modify an Application Environment panel contains an APPLENV parameter, the APPLENV parameter value in the Start Parameters field value overrides the value in the JCL startup procedure.

   2 In this line, the DB2SSN value must be the same as your Db2 for z/OS subsystem name. The maximum value of NUMTCB should be 5.

   3 The EXEC statement specifies program DSNX9WLM, which is the WLM address space initialization program for a 31-bit JVM. If your JAVA_HOME variable specifies a path to a 64-bit JVM, such as /usr/lpp/java/8.0_64, you need to change DSNX9WLM in the EXEC statement to DSNX9WJM.

   4 STEPLIB specifies the Db2 and Language Environment data sets that are necessary for running the stored procedures. At a minimum, you need the Language Environment run time data set, SCEERUN, and the SDSNLOAD data set, which contains the DSNX9WLM program. SDSNLOD2 contains the JDBC and SQLJ dynamic link libraries (DLLs).

   5 JAVAENV specifies a data set that contains Language Environment run time options for Java stored procedures. The presence of this DD statement indicates to Db2 that the WLM environment is for Java routines. This data set must contain the environment variable JAVA_HOME. This environment variable indicates to Db2 that the WLM environment is for Java routines. JAVA_HOME also specifies the highest-level directory in the set of directories that containing the SDK for Java.

6 Specifies the destination to which Db2 puts information that you can use to debug your stored procedure. The information that Db2 collects is for assistance in debugging setup problems, and should be used only under the direction of IBM Software Support. You should comment out this DD statement during production.

7 Specifies HFS files into which the Java run time environment puts information that you can use to debug your stored procedure. This information is for assistance in debugging setup problems. You should comment out these DD statements during production.

7. Modify the Language Environment run time options data set (JAVAENV data set), _CEE_ENVFILE file, and JVMPROPS file that the installation process created.

   The default names for those data sets are:

Data set type	Default name
JAVAENV	DSNB10.DSNWLMJ.JAVAENV
_CEE_ENVFILE	/usr/lpp/db2b10/base/classes/dsnenvfile.txt
JVMPROPS	/usr/lpp/java/properties/dsnjvmsp

   The name of the Language Environment run time options data set must be the same as the data set name in the JAVAENV DD statement in the JCL startup procedure for the stored procedure address spaces that are associated with the WLM application environment.

   The installation process uses JCL similar to this to allocate and populate the Language Environment run time options data set.

   Do not include sequence numbers in any of the input data sets. There should be no text after column 72.

```
//DSNTIJJ EXEC PGM=IEBGENER
//SYSIN    DD  DUMMY
//SYSPRINT DD  SYSOUT=*
//SYSUT2   DD  DSN=DSNB10.DSNWLMJ.JAVAENV,                              1
//             DISP=(,CATLG,DELETE),
//             UNIT=SYSDA,SPACE=(TRK,1),
//             DCB=(RECFM=VB,LRECL=255)
//SYSUT1   DD  *
ENVAR("_CEE_ENVFILE=/usr/lpp/db2b10/base/classes/dsnenvfile.txt", 2
      "DB2_BASE=/usr/lpp/db2b10/base",                             3
      "JCC_HOME=/usr/lpp/db2b10/jdbc",                              4
      "JAVA_HOME=/usr/lpp/java150/J5.0",                            5
      "JVMPROPS=/usr/lpp/java/properties/dsnjvmsp"),                6
MSGFILE(JSPDEBUG,,,,ENQ),                                           7
XPLINK(ON)                                                          8
```

   The installation process uses JCL similar to this to allocate and populate the _CEE_ENVFILE file.

```
//DSNTIJR EXEC PGM=IEBGENER
//SYSIN    DD  DUMMY
//SYSPRINT DD  SYSOUT=*
//SYSUT2   DD  PATH='/usr/lpp/db2b10/base/classes/dsnenvfile.txt', 9
//             FILEDATA=TEXT,
//             PATHDISP=(KEEP,DELETE),
//             PATHOPTS=(OWRONLY,OCREAT,OEXCL),
//             PATHMODE=(SIRWXU,SIRGRP)
//SYSUT1   DD  *

CLASSPATH=/usr/include/java_classes/gxljapi.jar                        10
LIBPATH=/usr/lib/java_runtime                                         11
STEPLIB=DSNB10.SDSNLOAD
12
```

   The installation process uses JCL similar to this to allocate and populate the JVMPROPS file.

```
//DSNTIJS EXEC PGM=IEBGENER
//SYSIN    DD  DUMMY
//SYSPRINT DD  SYSOUT=*
//SYSUT2   DD  PATH='/usr/lpp/java/properties/dsnjvmsp',               13
```

```
//             FILEDATA=TEXT,
//             PATHDISP=(KEEP,DELETE),
//             PATHOPTS=(OWRONLY,OCREAT,OEXCL),
//             PATHMODE=(SIRWXU,SIRGRP)
//SYSUT1   DD  *
# Sets the initial size of middleware heap within non-system heap
#-Xms64M
# Sets the maximum size of nonsystem heap
#-Xmx128M
```

Notes:

1. This is the name of the Language Environment run time options data set.
2. This line and the ones that follow it define the content of the Language Environment run time options data set. Change any data set names that differ from the ones in your environment.

   The content of the Language Environment run time options data set can have a length of no more than 245 bytes. If the length of the ENVAR options string makes the length of the content of the run time options data set greater than 245 bytes, you need to use a _CEE_ENVFILE file, whose name you specify in the ENVAR parameter. _CEE_ENVFILE identifies an additional file in which you can include more options. You can put all of your run time options in the _CEE_ENVFILE file, or put some options in the ENVAR parameter, and some in the _CEE_ENVFILE file.
3. The value of DB2_BASE is the highest-level directory in the set of HFS directories that contain Db2 for z/OS code.
4. The value of JCC_HOME is the highest-level directory in the set of directories that contain the JDBC driver.
5. The value of JAVA_HOME is the highest-level directory in the set of directories that contain the SDK for Java. The SYSPROC.XSR_COMPLETE stored procedure requires SDK for z/OS, Java 2 Technology Edition, V5 or later.
6. The value of JVMPROPS is the name of a z/OS UNIX System Services file that contains startup options for the JVM in which the stored procedure runs.
7. MSGFILE specifies the DD name of a data set in which Language Environment puts run time diagnostics. All subparameters in the MSGFILE parameter are optional. The first subparameter is the DD name.
8. XPLINK(ON) is required.
9. The value of PATH must be the same as the value of _CEE_ENVFILE.
10. CLASSPATH must include the path for the Java class that contains the JAR files for z/OS XML System Services XML schema registration (gxljapi.jar).
11. LIBPATH must include the path for the Java native libraries.
12. STEPLIB must include the Db2 for z/OS run time library (SDSNLOAD).
13. The value of PATH must be the same as the value of JVMPROPS.

Related concepts

WLM address space startup procedure for Java routines (Db2 Application Programming for Java)

Related information

Setting up a Service Definition (z/OS MVS Planning: Workload Management)

Defining the XML schema repository stored procedures to Db2

You execute CREATE PROCEDURE statements to define the XML schema repository stored procedures to Db2.

Before you begin

- Run job DSNTIJSG during installation to create the stored procedures.

About this task

The CREATE PROCEDURE statements for the XML schema repository stored procedures are in a program that installation job DSNTIJRT calls.

The C language stored procedures are:

- SYSPROC.XSR_REGISTER
- SYSPROC.XSR_ADDSCHEMADOC
- SYSPROC.XSR_REMOVE

The Java language stored procedure is SYSPROC.XSR_COMPLETE.

Binding the IBM Data Server Driver for JDBC and SQLJ packages for the XML schema repository

The XSR_COMPLETE XML schema repository stored procedure is a Java stored procedure. It requires the IBM Data Server Driver for JDBC and SQLJ packages.

About this task

You need to bind the following sets of IBM Data Server Driver for JDBC and SQLJ packages:

- The NULLID collection ID
- The SYSXSR collection ID

Procedure

To bind the IBM Data Server Driver for JDBC and SQLJ packages for the XML schema repository:

1. In z/OS UNIX System Services, execute the DB2Binder utility to bind the IBM Data Server Driver for JDBC and SQLJ under the SYSXSR collection ID.

   The DB2Binder command that you need to use looks like this:

   ```
   java com.ibm.db2.jcc.DB2Binder -url url
   -user user-id -collection SYSXSR -password password -action replace
   ```

   For example:

   ```
   java com.ibm.db2.jcc.DB2Binder -url myserver.svl.ibm.com:446/MYDB
   -user myid -collection SYSXSR -password mypass -action replace
   ```

   You need to make these substitutions when you run DB2Binder.

Value in example	Value to substitute
myserver.svl.ibm.com:446/MYDB	The server, port, and location name values for the Db2 subsystem on which you are setting up the XML schema repository.
myid	Your user ID
mypass	Your password

2. In z/OS UNIX System Services, execute the DB2Binder utility to bind the IBM Data Server Driver for JDBC and SQLJ under the NULLID collection ID.

The DB2Binder command looks like this.

```
java com.ibm.db2.jcc.DB2Binder -url jdbc:db2://myserver.svl.ibm.com:446/MYDB
-user myid -collection NULLID -password mypass -action replace
```

You need to make the same substitutions as in step 1.

Related reference

DB2Binder utility (Db2 Application Programming for Java)

Testing the XML schema repository setup

After you set up the XML schema repository, perform some tests to ensure that it is operating correctly. You need to test whether the WLM environment, the Java stored procedure environment, and the XML schema repository stored procedures are working correctly.

Procedure

Run installation job DSNTIJRV.

The following JCL uses program DSNTRVFY to check the XSR stored procedures:

```
//DSNTJVFY EXEC PGM=DSNTRVFY,
//         PARM='DB2SSN(DSN) ROUTINE(DD:SYSIN)'
//DBRMLIB  DD DSN=DSNB10.SDSNDBRM,DISP=SHR
//SYSUT1   DD UNIT=SYSDA,SPACE=(32000,(30,30)),DCB=(RECFM=VB,LRECL=133)
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
  SYSPROC.XSR_REGISTER
  SYSPROC.XSR_ADDSCHEMADOC
  SYSPROC.XSR_COMPLETE
  SYSPROC.XSR_REMOVE
```

In this example, the PARM passed to program DSNTRVFY directs it to verify the routines specified in ddname SYSIN.

In order to test your XML schema repository, job DSNTIJRV might generate error SQLCODEs in WLM task logs and trace records that can be ignored. Examine the ddname SYSPRINT summary reported by message DSNT040I to determine whether your XML schema repository is successfully set up. Reasons for failures can also be found in ddname SYSPRINT.

Related concepts

Job DSNTIJRV (Db2 Installation and Migration)

Related tasks

Setting up a WLM application environment for stored procedures during installation (Db2 Installation and Migration)

Related information

DSNT040I (Db2 Messages)

Chapter 2. Working with XML data

Db2 pureXML support lets you create tables, store and retrieve XML data in Db2 tables, and validate XML data.

Creation of tables with XML columns

To create tables with XML columns, you specify columns with the XML data type in the CREATE TABLE statement. A table can have one or more XML columns.

You do not specify a length when you define an XML column. There is no architectural limit on the size of an XML value in a database. However, textual XML data that is exchanged with a Db2 database is limited to 2 GB-1, so the effective limit of an XML column is 2 GB-1.

Like a LOB column, an XML column holds only a descriptor of the column. The data is stored separately.

When you define an XML column, you can add an *XML type modifier*. An XML type modifier associates a set of one or more XML schemas with the XML data type. You can use an XML type modifier to cause all XML documents that are stored in an XML column to be validated according to one of the XML schemas that is specified in the type modifier.

When you create a table with an XML column in a universal table space, Db2 maintains multiple versions of XML documents during update operations, to enhance concurrency and memory usage.

Example: A table for customer data contains two XML columns. The definition looks like this:

```
CREATE TABLE CUSTOMER (CID BIGINT NOT NULL PRIMARY KEY,
                       INFO XML,
                       HISTORY XML)
```

Example: A table for customer data contains an XML column named CONTENT. The documents in the XML column need to be validated according to XML schema SYSXSR.PO1, which has already been registered. The definition looks like this:

```
CREATE TABLE PURCHASEORDERS(
  ID INT NOT NULL,
  CONTENT XML(XMLSCHEMA ID SYSXSR.PO1))
```

Related concepts

XML data type
The XML data type is used to define columns of a table that store XML values. This data type provides the ability to store well-formed XML documents in a database.

XML versions
Multiple versions of an XML document can coexist in an XML table. The existence of multiple versions of an XML document can lead to improved concurrency through lock avoidance. In addition, multiple versions can save real storage by avoiding a copy of the old values in the document into memory in some cases.

Altering tables with XML columns

To add XML columns to existing tables, you specify columns with the XML data type in the ALTER TABLE statement with the ADD COLUMN clause. A table can have one or more XML columns.

To alter an existing XML column to include an XML type modifier or remove an XML type modifier, use ALTER TABLE.

When you add XML columns to a table, the Db2 database server implicitly creates a table space and table for each XML column. The data for an XML column is stored in the corresponding table.

An XML type modifier associates a set of one or more XML schemas with the XML data type. You can use an XML type modifier to cause all XML documents that are stored in an XML column to be validated

© Copyright IBM Corp. 2007, 2020

according to one of the XML schemas that is specified in the type modifier. When you perform either of the following actions, Db2 puts the XML table space that corresponds to the altered XML column in CHECK-pending status:

- Add an XML type modifier to an existing XML column that does not have a type modifier but contains XML data
- Remove an XML schema from an existing XML type modifier that has more than one XML schema

When the XML table space is in CHECK-pending status, you need to run CHECK DATA to validate the values of the altered XML column for existing rows. If you add an XML schema to the type modifier of an existing XML column, Db2 does not put the XML table space in CHECK-pending status, and values of the altered column in the existing rows are not revalidated.

When you add an XML column to a table that is in a universal table space, Db2 maintains multiple versions of XML documents during update operations, to enhance concurrency and memory usage.

Example: A table contains customer data that contains two XML columns. The definition looks like this:

```
CREATE TABLE Customer (Cid BIGINT NOT NULL PRIMARY KEY,
                       Info XML,
                       History XML)
```

Create a table named MyCustomer that is a copy of Customer, and add an XML column to describe customer preferences:

```
CREATE TABLE MyCustomer LIKE Customer;
ALTER TABLE MyCustomer ADD COLUMN Preferences XML;
```

Example: A table for customer data contains an XML column named CONTENT. The definition looks like this:

```
CREATE TABLE PURCHASEORDERS(
  ID INT NOT NULL,
  CONTENT XML)
```

The table contains several XML documents. The documents in the XML column need to be validated according to XML schema SYSXSR.PO1, which has already been registered. Alter the XML column to add an XML type modifier that specifies SYSXSR.PO1:

```
ALTER TABLE PURCHASEORDERS
  ALTER CONTENT
  SET DATA TYPE XML(XMLSCHEMA ID SYSXSR.PO1)
```

The table space that contains the XML documents for the CONTENT column is put in CHECK-pending status. You need to run CHECK DATA against the XML table space to remove the CHECK-pending status.

Related concepts

Creation of tables with XML columns
To create tables with XML columns, you specify columns with the XML data type in the CREATE TABLE statement. A table can have one or more XML columns.

XML schema validation with an XML type modifier
You can automate XML schema validation by adding an XML type modifier to an XML column definition.

Storage structure for XML data
The storage structure for XML data is similar to the storage structure for LOB data.

XML versions
Multiple versions of an XML document can coexist in an XML table. The existence of multiple versions of an XML document can lead to improved concurrency through lock avoidance. In addition, multiple

versions can save real storage by avoiding a copy of the old values in the document into memory in some cases.

Storage structure for XML data

The storage structure for XML data is similar to the storage structure for LOB data.

As with LOB data, the *base table* that contains an XML column exists in a different table space from the table that contains the XML data.

The storage structure for the XML data depends on the type of table space that contains the base table, as described in the following table.

Table 8. Organization of base table spaces and corresponding XML table spaces

Base table space organization	XML table space organization	Remarks
Partition-by-growth[1]	Partition-by-growth	An XML document can span more than one partition. The base table space and the XML table space grow independently.
partition-by-range [1]	partition-by-range	If a base table row moves to a new partition, the XML document also moves to a new partition.
Partitioned (non-UTS)	partition-by-range	If a base table row moves to a new partition, the XML document also moves to a new partition. This table space type is deprecated.
Segmented (non-UTS)	Partition-by-growth	This table space type is deprecated.
Simple	Partition-by-growth	This table space type is deprecated.

Note:

1. This table space organization supports XML versions.

The following figure demonstrates the relationship between segmented table spaces for base tables with XML columns and the corresponding XML table spaces and tables. The relationships are similar for simple base table spaces and partition-by-growth base table spaces. This figure represents XML columns that do not support XML versions.

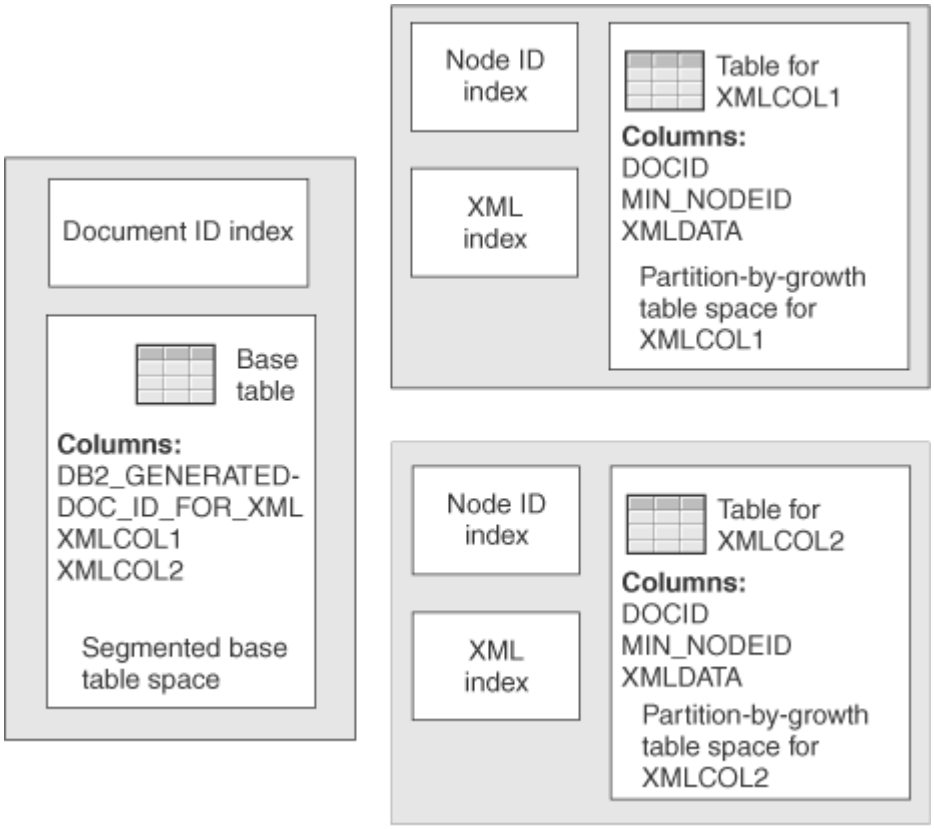


Figure 2. XML storage structure for a base table in a segmented table space

The following figure demonstrates the relationship between partitioned table spaces for base tables with XML columns and the corresponding XML table spaces and tables. The relationships are similar for partition-by-range base table spaces. This figure represents XML columns that do not support XML versions.

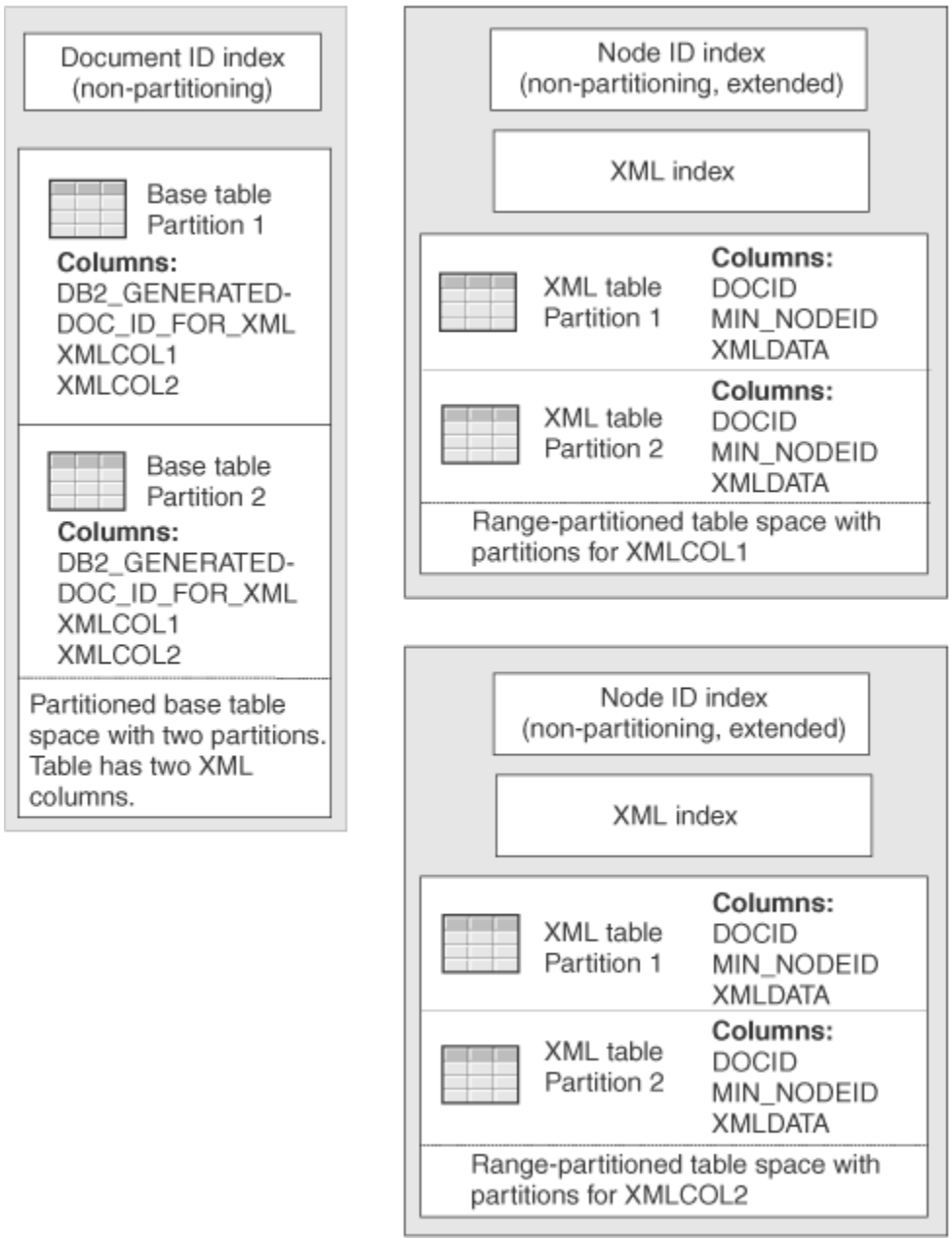


Figure 3. XML storage structure for a base table in a partitioned table space

When you create a table with XML columns or ALTER a table to add XML columns, the Db2 database server implicitly creates the following objects:

- A table space and table for each XML column. The data for an XML column is stored in the corresponding table.

  Db2 creates the XML table space and table in the same database as the base table that contains the XML column. The XML table space is in the Unicode UTF-8 encoding scheme.

  If the base table contains XML columns that support XML versions, each XML table contains two more columns than an XML table for an XML column that does not support XML versions. Those columns are named START_TS and END_TS, and they have the BINARY(8) data type if the page format is basic 6-byte format and BINARY(10) data type if the page format is extended 10-byte format. START_TS contains the RBA or LRSN of the logical creation of an XML record. END_TS contains the RBA or LRSN of

the logical deletion of an XML record. START_TS and END_TS identify the rows in the XML table that make up a version of an XML document.

- An document ID column in the base table, named DB2_GENERATED_DOCID_FOR_XML, with data type BIGINT.

  DB2_GENERATED_DOCID_FOR_XML holds a unique document identifier for the XML columns in a row. One DB2_GENERATED_DOCID_FOR_XML column is used for all XML columns.

  The DB2_GENERATED_DOCID_FOR_XML column has the GENERATED ALWAYS attribute. Therefore, a value in this column cannot be NULL.

  If the base table space supports XML versions, the length of the XML indicator column is eight bytes longer that the XML indicator column in a base table space that does not support XML versions.
- An index on the DB2_GENERATED_DOCID_FOR_XML column.

  This index is known as a document ID index.
- An index on each XML table that Db2 uses to maintain document order, and map logical node IDs to physical record IDs.

  This index is known as a node ID index. The node ID index is an extended, non-partitioning index.

  If the base table space supports XML versions, the index key for the node ID index contains two more columns than the index key for a node ID index for a base table space that does not support XML versions. Those columns are named START_TS and END_TS, and they have the BINARY(8) data type.

You can perform limited SQL operations,on the implicitly created objects, such as altering the following attributes of an XML table space:

- SEGSIZE
- BUFFERPOOL
- STOGROUP
- PCTFREE
- GBPCACHE

You can also the any except for the following attributes of the document ID index or node ID index:

- CLUSTER
- PADDED
- Number of columns (ADD COLUMN is not allowed)

For a complete list of operations that you can perform on these objects, see ALTER TABLESPACE (Db2 SQL), ALTER TABLE (Db2 SQL), and ALTER INDEX (Db2 SQL).

Related concepts

Access methods with XML indexes
Several data access methods use XML indexes.

XML versions
Multiple versions of an XML document can coexist in an XML table. The existence of multiple versions of an XML document can lead to improved concurrency through lock avoidance. In addition, multiple versions can save real storage by avoiding a copy of the old values in the document into memory in some cases.

XML table space implicit creation (Db2 Administration Guide)

Limitation of XML virtual storage usage

You can use the XMLVALA and XMLVALS subsystem parameters to limit the amount of Db2 virtual storage that is used for XML processing.

Because XML values are defined without a maximum size, Db2 cannot estimate the amount of memory that it needs for processing SQL/XML and XQuery queries before run time. Db2 allocates virtual storage at

run time based on the size of the XML data. For large XML data, the amount of virtual storage that Db2 requires can grow very large.

If your Db2 subsystem encounters storage constraints because XML values are using too much memory, set the XMLVALA and XMLVALS subsystem parameters:

- To specify the maximum amount of memory, in KB, that Db2 uses for storing XML values for each user, set XMLVALA. The default is 204800 KB.
- To specify the maximum amount of memory, in MB, that Db2 uses for storing XML values for the entire subsystem, set XMLVALS. The default is 10240 MB.

Related concepts

Storage structure for XML data
The storage structure for XML data is similar to the storage structure for LOB data.

Related reference

USER XML VALUE STG field (XMLVALA subsystem parameter) (Db2 Installation and Migration)
SYSTEM XML VALUE STG field (XMLVALS subsystem parameter) (Db2 Installation and Migration)

Insertion of rows with XML column values

To insert rows into a table that contains XML columns, you can use the SQL INSERT statement.

The documents that you insert into XML columns must be well-formed XML documents, as defined in the XML 1.0 specification. A document node will be created implicitly if one does not already exist. The application data type can be XML (XML AS BLOB, XML AS CLOB, XML AS DBCLOB), character, or binary.

Recommendation: Insert XML data from host variables, rather than literals, so that the Db2 database server can use the host variable data type to determine some of the encoding information.

XML data in an application can be in textual XML format or binary XML format (Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format). Binary XML format is valid only for JDBC, SQLJ, and ODBC applications. When you insert the data into an XML column, it must be converted to its XML hierarchical format. The Db2 database server performs this operation implicitly when XML data is inserted directly from a host variable into an XML column. Alternatively, you can invoke the XMLPARSE function explicitly when you perform the insert operation, to convert the data to the XML hierarchical format.

During document insertion, you can validate the XML document against a registered XML schema. If the XML column into which you are inserting a document has an XML schema modifier, validation occurs automatically. Otherwise, you can call the DSN_XMLVALIDATE function to do XML schema validation. You can perform validation during document insertion only if the document is in the textual XML format.

The following examples demonstrate how XML data can be inserted into XML columns. The examples use table MYCUSTOMER, which is a copy of the sample CUSTOMER table. The XML data that is to be inserted looks like this:

```
<customerinfo xmlns="http://posample.org" Cid="1015">
<name>Christine Haas</name>
<addr country="Canada">
<street>12 Topgrove</street>
<city>Toronto</city>
<prov-state>Ontario</prov-state>
<pcode-zip>N8X-7F8</pcode-zip>
</addr>
<phone type="work">905-555-5238</phone>
<phone type="home">416-555-2934</phone>
</customerinfo>
```

Example: In a JDBC application, read textual XML data from file c6.xml as binary data, and insert the data into an XML column:

```
PreparedStatement insertStmt = null;
String sqls = null;
int cid = 1015;
sqls = "INSERT INTO MyCustomer (Cid, Info) VALUES (?, ?)";
insertStmt = conn.prepareStatement(sqls);
insertStmt.setInt(1, cid);
```

```
File file = new File("c6.xml");
insertStmt.setBinaryStream(2, new FileInputStream(file), (int)file.length());
insertStmt.executeUpdate();
```

Example: Suppose that the data in file c7.xml contains the same XML document as in file c6.xml, but the data is stored in binary XML format. In a JDBC application, read the data from file c7.xml, and insert the data into an XML column:

```
…
SQLXML info = conn.createSQLXML();
OutputStream os = info.setBinaryStream();
FileInputStream fis = new FileInputStream("c7.xml");
int read;
while ((read = fis.read ()) != -1) {
  os.write (read);
}
PreparedStatement insertStmt = null;
String sqls = null;
int cid = 1015;
sqls = "INSERT INTO MyCustomer (Cid, Info) VALUES (?, ?)";
insertStmt = conn.prepareStatement(sqls);
insertStmt.setInt(1, cid);
insertStmt.setSQLXML(2, info);
insertStmt.executeUpdate();
```

Example: In a static embedded C application, insert data from an XML AS BLOB host variable into an XML column:

```
EXEC SQL BEGIN DECLARE SECTION;
   sqlint64 cid;
   SQL TYPE IS XML AS BLOB (10K) xml_hostvar;
EXEC SQL END DECLARE SECTION;
…
cid=1015;
/* Read data from file c6.xml into xml_hostvar */
…
EXEC SQL INSERT INTO MyCustomer (Cid,Info) VALUES (:cid, :xml_hostvar);
```

Example: In a static embedded COBOL application, insert data from a character XML host variable into an XML column:

```
…
 WORKING-STORAGE SECTION.
…
* XML HOST VARIABLES
 01  CLOB-XML-IN USAGE IS SQL TYPE IS XML AS CLOB(10K).
 01  CLOB-XML-OUT USAGE IS SQL TYPE IS XML AS CLOB(10K).
* VARIABLE USED FOR DISPLAY OF THE RETRIEVED VALUE
 01  CLOB-XML-OUT-DISPLAY.
   02  CLOB-XML-OUT-DISPLAY-LENGTH
       PIC 9(9) COMP.
   02  CLOB-XML-OUT-DISPLAY-DATA.
       49  FILLER PIC X(10240).
*************************************************
* SQL INCLUDE FOR SQLCA                         *
*************************************************
      EXEC SQL INCLUDE SQLCA  END-EXEC.
 PROCEDURE DIVISION.
* USE XMLPARSE TO CONVERT THE INPUT DATA TO THE XML TYPE.
      EXEC SQL SET :CLOB-XML-IN=
       XMLPARSE(DOCUMENT
       '<customerinfo xmlns="http://posample.org" Cid="1015">' ||
       '<name>Christine Haas</name>'                            ||
       '<addr country="Canada">'                                ||
       '<street>12 Topgrove</street>'                           ||
       '<city>Toronto</city>'                                   ||
       '<prov-state>Ontario</prov-state>'                       ||
       '<pcode-zip>N8X-7F8</pcode-zip>'                         ||
       '</addr>'                                                ||
       '<phone type="work">905-555-5238</phone>'                ||
       '<phone type="home">416-555-2934</phone>'                ||
       '</customerinfo>')
      END-EXEC.
* INSERT THE DATA.
      EXEC SQL INSERT INTO CUSTOMER(CID, INFO)
       VALUES (1015,:CLOB-XML-IN)
      END-EXEC.
```

```
* CHECK THE VALUE THAT YOU INSERTED.
      EXEC SQL SELECT INFO
       INTO :CLOB-XML-OUT FROM CUSTOMER
       WHERE CID=1015
      END-EXEC.
      MOVE CLOB-XML-OUT TO CLOB-XML-OUT-DISPLAY.
      DISPLAY
      CLOB-XML-OUT-DISPLAY-DATA(1:CLOB-XML-OUT-DISPLAY-LENGTH).
```

Related concepts

XML parsing
XML parsing is the process of converting XML data from its textual XML format to its hierarchical format.

XML schema validation
XML schema validation is the process of determining whether the structure, content, and data types of an XML document are valid according to an XML schema.

Updates of XML columns

To update entire documents in an XML column, you can use the SQL UPDATE statement. You can include a WHERE clause when you want to update specific rows. To update portions of XML documents, use the XMLMODIFY function with a basic XQuery updating expression.

Related concepts

XML parsing
XML parsing is the process of converting XML data from its textual XML format to its hierarchical format.

XMLEXISTS predicate for querying XML data
The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.

Updates of entire XML documents

To update an entire XML document in an XML column, use the SQL UPDATE statement. Include a WHERE clause when you want to update specific rows.

The input to the XML column must be a well-formed XML document, as defined in the XML 1.0 specification. A document node will be created implicitly if one does not already exist. The application data type can be XML (XML AS BLOB, XML AS CLOB, XML AS DBCLOB), character, or binary.

XML data in an application can be in textual XML format or extensible dynamic binary XML Db2 client/server binary XML format (binary XML format). Binary XML format is valid only for JDBC, SQLJ, and ODBC applications. When you update data in an XML column, it must be converted to its XML hierarchical format. The Db2 database server performs this operation implicitly when XML data from a host variable directly updates an XML column. Alternatively, you can invoke the XMLPARSE function explicitly when you perform the update operation, to convert the data to the XML hierarchical format.

When you update an XML column, you might also want to validate the input XML document against a registered XML schema. You can do that in one of the following ways:

- Implicitly, if the XML column has an XML type modifier defined on it.
- Explicitly, with the DSN_XMLVALIDATE function.

You can use XML column values to specify which rows are to be updated. To find values within XML documents, you need to use XQuery expressions. One way of specifying XQuery expressions is the XMLEXISTS predicate, which allows you to specify an XQuery expression and determine if the expression results in an empty sequence. When XMLEXISTS is specified in the WHERE clause, rows will be updated if the XQuery expression returns a non-empty sequence.

The following examples demonstrate how XML data can be updated in XML columns. The examples use table MYCUSTOMER, which is a copy of the sample CUSTOMER table. The examples assume that MYCUSTOMER already contains a row with a customer ID value of 1004. The XML data that updates existing column data is in file c7.xml, and looks like this:

```
<customerinfo xmlns="http://posample.org" Cid="1004">
<name>Christine Haas</name>
```

```
<addr country="Canada">
<street>12 Topgrove</street>
<city>Toronto</city>
<prov-state>Ontario</prov-state>
<pcode-zip>N9Y-8G9</pcode-zip>
</addr>
<phone type="work">905-555-5238</phone>
<phone type="home">416-555-2934</phone>
</customerinfo>
```

Example: In a JDBC application, read XML data from file c7.xml as binary data, and use it to update the data in an XML column:

```
PreparedStatement updateStmt = null;
String sqls = null;
int cid = 1004;
sqls = "UPDATE Customer SET Info=? WHERE Cid=?";
updateStmt = conn.prepareStatement(sqls);
updateStmt.setInt(2, cid);
File file = new File("c7.xml");
updateStmt.setBinaryStream(1, new FileInputStream(file), (int)file.length());
updateStmt.executeUpdate();
```

Example: In an embedded C application, update data in an XML column from an XML AS BLOB host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   sqlint64 cid;
   SQL TYPE IS XML AS BLOB (10K) xml_hostvar;
EXEC SQL END DECLARE SECTION;
…
cid=1004;
/* Read data from file c7.xml into xml_hostvar */
…
EXEC SQL UPDATE MYCUSTOMER SET xmlcol=:xml_hostvar WHERE Cid=:cid;
```

In these examples, the value of the Cid attribute within the <customerinfo> element happens to be stored in the CID relational column as well. Because of this, the WHERE clause in the UPDATE statements used the relational column CID to specify the rows to update. In the case where the values that determine which rows are chosen for update are found only within the XML documents themselves, the XMLEXISTS predicate can be used. For example, the UPDATE statement in the previous embedded C application example can be changed to use XMLEXISTS as follows:

```
EXEC SQL UPDATE MYCUSTOMER SET xmlcol=:xml_hostvar
       WHERE XMLEXISTS ('declare default element namespace "http://posample.org";
                        /customerinfo[@Cid = $c]'
                        passing INFO, cast(:cid as integer) as "c");
```

Partial updates of XML documents

To update part of an XML document in an XML column, use the SQL UPDATE statement with the XMLMODIFY built-in scalar function.

The XMLMODIFY function specifies a *basic updating expression* that you can use to insert nodes, delete nodes, replace nodes, or replace the values of a node in XML documents that are stored in XML columns.

Before you can use XMLMODIFY to update part of an XML document, the column that contains the XML document must support XML versions.

The types of basic updating expressions are:

insert expression
Inserts copies of one or more nodes into a designated position in a node sequence.

replace expression
Replaces an existing node with a new sequence of zero or more nodes, or replaces a node's value while preserving the node's identity.

delete expression
Deletes zero or more nodes from a node sequence.

Example

Suppose that you want to replace the second shipTo node in a purchaseOrder document that has purchase order ID (POID) 5000, and looks like this:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

You can use an SQL UPDATE statement like this to replace the shipTo node:

```
UPDATE PURCHASEORDER
 SET INFO = XMLMODIFY(
  'declare namespace ipo="http://www.example.com/IPO";
  replace node /ipo:purchaseOrder/shipTo[name="Joe Lee"]
  with $x', XMLPARSE(
   '<shipTo exportCode="1" xsi:type="ipo:USAddress">
    <name>Joe Lee</name>
    <street>555 Quarry Road</street>
    <city>Palo Alto</city>
    <state/>CA
    <postcode>94304</postcode>
    </shipTo>') AS "x")
 WHERE POID=5000
```

After the statement is executed, the contents of the document in the PORDER column are:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
```

```
  <shipTo exportCode="1" xsi:type="ipo:USAddress">
   <name>Joe Lee</name>
   <street>555 Quarry Road</street>
   <city>Palo Alto</city>
   <state>CA</state>
   <postcode>94304</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>505 First Street</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

Deletion of rows with XML documents from tables

To delete rows that contain XML documents, you can use the DELETE SQL statement. You can include a WHERE clause when you want to delete specific rows.

You can specify which rows are to be deleted based on values within XML columns. To find values within XML documents, you need to use XQuery expressions. One way of specifying XQuery expressions is with the XMLEXISTS predicate. When you specify XMLEXISTS in a WHERE clause, rows are deleted if the XQuery expression returns a non-empty sequence.

If an XML column is nullable, to delete a value from the XML column without deleting the row, use the UPDATE SQL statement to set the column value to NULL.

The following examples demonstrate how XML data can be deleted from XML columns. The examples use table MYCUSTOMER, which is a copy of the sample CUSTOMER table, and assume that MYCUSTOMER has been populated with all of the Customer data.

Example: Delete the rows from table MYCUSTOMER for which the CID column value is 1002.

```
DELETE FROM MYCUSTOMER WHERE CID=1002
```

Example: Delete the rows from table MYCUSTOMER for which the value of the `city` element is `Markham`.

```
DELETE FROM MYCUSTOMER
 WHERE XMLEXISTS ('declare default element namespace "http://posample.org";
//addr[city="Markham"]' passing INFO)
```

Example: Delete the XML document in the row of MYCUSTOMER for which the value of the `city` element is `Markham`, but leave the row.

```
UPDATE MYCUSTOMER SET INFO = NULL
  WHERE XMLEXISTS ('$declare default element namespace "http://posample.org";
  //addr[city="Markham"]' passing INFO)
```

Related concepts

XMLEXISTS predicate for querying XML data

The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.

XML versions

Multiple versions of an XML document can coexist in an XML table. The existence of multiple versions of an XML document can lead to improved concurrency through lock avoidance. In addition, multiple versions can save real storage by avoiding a copy of the old values in the document into memory in some cases.

Db2 supports multiple versions of an XML document in an XML column if the base table space for the table that contains the XML column is a universal table space, and all other XML columns in the table support multiple versions.

If an XML table does not support multiple XML versions, you can convert the table to a table that supports multiple XML versions by following these steps:

1. Unload the data from the table that contains the XML columns.
2. Drop the table.
3. Create the table in a universal table space. The new table supports multiple XML versions.
4. Load the data into the table.

With XML versions, when you insert an XML document into an XML column, Db2 assigns a version number to the XML document. If the entire XML document is updated, Db2 creates a new version of the document in the XML table. If a portion of the XML document is updated, Db2 creates a new version of the updated portion. When Db2 uses XML versions, more data set space is required than when versions are not used. However, Db2 periodically deletes versions that are no longer needed. In addition, you can run the REORG utility against the XML table space that contains the XML document to remove unneeded versions. Db2 removes versions of a document when update operations that require the versions are committed, and when there are no readers that reference the unneeded versions.

XML versions are different from table space versions or index versions. The purpose of XML versions is to optimize concurrency and memory usage. The purpose of table space and index versions is to maximize data availability.

Example of improved concurrency with XML versions: The following example demonstrates how multiple XML versions can improve concurrency when the same XML documents are modified multiple times within the same transaction.

Suppose that table T1, which is in a universal table space, is defined like this:

```
CREATE T1 (INT1 INT,
XML1 XML,
XML2 XML);
```

The table contains the following data.

INT1	XML1	XML2
350	<A1>111</A1>	<A2>aaa</A2>
100	<A1>111</A1>	<A2>aaa</A2>
250	<A1>111</A1>	<A2>aaa</A2>

An application performs SQL read operations that are represented by the following pseudocode:

```
EXEC SQL
 DECLARE CURSOR C1 FOR
 SELECT INT1, XML1
  FROM T1
  ORDER BY INT1
  FOR READ ONLY;
```

At the same time, another application performs SQL write operations that are represented by the following pseudocode:

```
EXEC SQL UPDATE T1
 SET XML1 = XMLPARSE(DOCUMENT '<B1>222</B1>');
EXEC SQL OPEN CURSOR C1;
EXEC SQL UPDATE T1
 SET XML1 = XMLPARSE(DOCUMENT '<C1>333</C1>');
EXEC SQL FETCH FROM C1 INTO :HVINT1, :HVXML1;
```

With multiple versions, the reading application does not need to hold a lock, so the updating application can do its update operations without waiting for the reading application to finish. The reading application reads the old versions of the XML values, which are consistent data.

Example of improved storage usage with XML versions: The following example demonstrates how multiple XML versions can result in the use of less real storage when an XML document is the object of a self-referencing update operation.

Suppose that table T1, which is in a universal table space, is defined like this:

```
CREATE T1 (INT1 INT,
XML1 XML,
XML2 XML);
```

The table contains the following data.

INT1	XML1	XML2
350	<A1>111</A1>	<A2>aaa</A2>
100	<A1>111</A1>	<A2>aaa</A2>
250	<A1>111</A1>	<A2>aaa</A2>

An application performs SQL operations that are represented by the following pseudocode:

```
EXEC SQL
 UPDATE T2
 SET XML1 = XML2,                 1
  XML2 = XML1                     2
 WHERE INT1 = 100;
EXEC SQL
 COMMIT                           3 ;
```

The results of those operations are:

1. When column XML1 is updated, Db2 stores the updated document as a new version in the XML table for column XML1. There are now two versions of the XML document for the second row of column XML1:

   ```
   First version:  <A1>111</A1>
   Second version: <A2>aaa</A2>
   ```

2. When column XML2 is updated, Db2 stores the updated document as a new version in the XML table for column XML2. There are now two versions of each XML document for the second row of column XML2:

   ```
   First version:  <A2>aaa</A2>
   Second version: <A1>111</A1>
   ```

3. The update operations are committed, so the old versions are no longer needed. Db2 deletes those versions from the XML tables for columns XML1 and XML2.

Without multiple XML versions, Db2 needs to copy the original versions of the updated documents into memory, so that their values are not lost. For large XML documents, storage shortages might result.

XML support in triggers

You can use the CREATE TRIGGER SQL statement to create BEFORE UPDATE or AFTER UPDATE triggers on XML columns. You can also use this statement to create INSERT or DELETE triggers on tables that include XML columns.

Triggers on tables with XML columns have the following restrictions:

- A transition variable cannot have the XML type.
- A column of a transition table that is referenced in the trigger body cannot have the XML type.

Example: Create a BEFORE UPDATE trigger on table MYCUSTOMER, which is a copy of the sample CUSTOMER table.

```
CREATE TRIGGER UPDBEFORE
  NO CASCADE BEFORE UPDATE ON MYCUSTOMER
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET N.CID=O.CID+10000;
  END
```

Although MYCUSTOMER contains two XML columns, the transition variable N.CID refers to a non-XML column, so this trigger is valid.

Example: Create an INSERT trigger on the MYCUSTOMER table.

```
CREATE TRIGGER INSAFTR
  AFTER INSERT ON MYCUSTOMER
  REFERENCING NEW TABLE AS N
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SELECT N.CID FROM N;
  END
```

Although transition table N has two XML columns, the trigger body does not refer to the XML columns of the transition table, so this trigger is valid.

If you need to use data from an XML column in a transition variable, you can circumvent the restriction on transition variables of the XML data type in triggers by using the XMLTABLE function to access the data in the XML column as non-XML data types.

Example: Suppose that the CUST table is defined like this:

```
CREATE TABLE CUST (
 ID   BIGINT NOT NULL PRIMARY KEY,
 NAME VARCHAR(30),
 CITY VARCHAR(20),
 ZIP  VARCHAR(12),
 INFO XML)
```

Create an INSERT trigger on the CUST table that copies name, city, and zip code information to itself for rows that are inserted into the CUST table. The data that you need to copy is in XML column INFO. You cannot refer to INFO directly in the trigger body. However, you can use the XMLTABLE function to create a result table with non-XML columns that contain the fields that you need. Then you can use a subselect to retrieve the row of the result table that corresponds to the row whose insertion activates the trigger.

```
CREATE TRIGGER INS_CUST
   AFTER INSERT ON CUST
   REFERENCING NEW AS NEWROW
   FOR EACH ROW MODE DB2SQL
   BEGIN ATOMIC
     UPDATE CUST
       SET (NAME, CITY, ZIP) =
           (SELECT X.NAME, X.CITY, X.ZIP
             FROM CUST, XMLTABLE('CUSTOMERINFO' PASSING CUST.INFO
                   COLUMNS
                     NAME VARCHAR(30)  PATH 'NAME',
                     CITY VARCHAR(20)  PATH 'ADDR/CITY',
                     ZIP  VARCHAR(12)  PATH 'ADDR/PCODE-ZIP') AS X
             WHERE CUST.ID = NEWROW.ID
```

```
            )
        WHERE CUST.ID = NEWROW.ID;
    END
```

Related reference

CREATE TRIGGER (Db2 SQL)

XML parsing

XML parsing is the process of converting XML data from its textual XML format to its hierarchical format.

You can let the Db2 database manager perform parsing implicitly, or you can call the XMLPARSE function to perform XML parsing explicitly.

Implicit XML parsing occurs in the following cases:

- When you pass data to the database server using a host variable of type XML, or use a parameter marker of type XML

  The database server does the parsing when it binds the value for the host variable or parameter marker for use in statement processing.
- When you assign a host variable, parameter marker, or SQL expression with a string data type (character, graphic or binary) to an XML column in an INSERT, UPDATE, DELETE, or MERGE statement. The parsing occurs when the Db2 database system implicitly adds an XMLPARSE function to the statement.

You perform *explicit XML parsing* when you invoke the XMLPARSE function on the input XML data. You can use the result of XMLPARSE in any context that accepts an XML data type. For example, you can assign the result to an XML column.

The XMLPARSE function takes a character or binary data type as input. For embedded dynamic SQL applications, if the argument of the XMLPARSE function is a parameter marker, it must be a typed marker. For example:

```
INSERT INTO MYCUSTOMER (CID, INFO)
 VALUES (?, xmlparse(document cast(? as clob(1k)) preserve whitespace))
```

Related concepts

Updates of XML columns

To update entire documents in an XML column, you can use the SQL UPDATE statement. You can include a WHERE clause when you want to update specific rows. To update portions of XML documents, use the XMLMODIFY function with a basic XQuery updating expression.

XML schema validation

XML schema validation is the process of determining whether the structure, content, and data types of an XML document are valid according to an XML schema.

Related reference

XMLPARSE (Db2 SQL)

XML parsing and whitespace handling

During explicit XML parsing, you can control the preservation or stripping of boundary whitespace characters when you store the data in the database.

According to the XML standard, whitespace is space characters (U+0020), carriage returns (U+000D), line feeds (U+000A), or tabs (U+0009) that are in the document to improve readability. When any of these characters appear as part of a text string, they are not considered to be whitespace.

Boundary whitespace is whitespace characters that appear between elements. For example, in the following document, the spaces between <a> and and between and are boundary whitespace.

```
<a> <b> and between </b> </a>
```

With explicit invocation of XMLPARSE, you use the STRIP WHITESPACE or PRESERVE WHITESPACE option to control preservation of boundary whitespace. The default is stripping of boundary whitespace.

The XML standard specifies an `xml:space` attribute that controls the stripping or preservation of whitespace within XML data. Possible values are `preserve` or `default`. The Db2 database server ignores any other values. The `preserve` value causes boundary whitespace within an element to be preserved, regardless of application settings, such as the XMLPARSE whitespace setting. The `default` value causes application settings to be used for boundary whitespace handling. `xml:space` attributes override any whitespace settings for implicit or explicit XML parsing, except for end-of-line processing. For end-of-line processing, when a carriage return character and a line feed character appear together, they are replaced with a line feed character. A carriage return character that appears by itself is replaced with a line feed character. These replacements occur, regardless of the `xml:space` attribute.

For example, in the following document, the spaces immediately before and after <b> are always preserved, regardless of any XML parsing options, because the spaces are within a node with the attribute `xml:space="preserve"`:

```
<a xml:space="preserve"> <b> <c>c</c>b </b></a>
```

However, in the following document, the spaces immediately before and after <b> can be controlled by the XML parsing options, because the spaces are within a node with the attribute `xml:space="default"`:

```
<a xml:space="default"> <b> <c>c</c>b </b></a>
```

XML parsing and DTDs

If the input data contains an internal document type declaration (DTD), the XML parsing process also checks the syntax of those DTDs.

In addition, the parsing process:

- Applies default values that are defined by the internal DTDs
- Expands entity references

Example: implicit XML parsing File c8.xml contains the following document:

```
<customerinfo xml:space="preserve" xmlns="http://posample.org" Cid='1008'>
  <name>Kathy Smith</name>
  <addr country='Canada'>
  <street>14 Rosewood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>M6W 1E6</pcode-zip>
  </addr>
  <phone type='work'>416-555-3333</phone>
</customerinfo>
```

In a JDBC application, read the XML document from the file, and insert the data into XML column Info of table MYCUSTOMER, which is a copy of the sample Customer table. Let the Db2 database server perform an implicit XML parse operation.

```
PreparedStatement insertStmt = null;
String sqls = null;
int cid = 1008;
sqls = "INSERT INTO MYCUSTOMER (Cid, Info) VALUES (?, ?)";
insertStmt = conn.prepareStatement(sqls);
insertStmt.setInt(1, cid);
File file = new File("c8.xml");
insertStmt.setBinaryStream(2, new FileInputStream(file), (int)file.length());
insertStmt.executeUpdate();
```

No whitespace handling is specified, so the default behavior of stripping whitespace is assumed. However, the document contains the `xml:space="preserve"` attribute, so whitespace is preserved. This means that, after end-of-line processing, the line feeds and spaces between the elements in the document remain.

If you retrieve the stored data, content looks like this:

```
<customerinfo xml:space="preserve" xmlns="http://posample.org" Cid='1008'>
  <name>Kathy Smith</name>
  <addr country='Canada'>
  <street>14 Rosewood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>M6W 1E6</pcode-zip>
  </addr>
  <phone type='work'>416-555-3333</phone>
</customerinfo>
```

Example: explicit XML parsing Assume that the following document is in BLOB host variable blob_hostvar.

```
<customerinfo xml:space="default" xmlns="http://posample.org" Cid='1009'>
  <name>Kathy Smith</name>
  <addr country='Canada'>
  <street>15 Rosewood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>M6W 1E6</pcode-zip>
  </addr>
  <phone type='work'>416-555-4444</phone>
</customerinfo>
```

In a static embedded C application, insert the document from the host variable into XML column Info of table MYCUSTOMER. The host variable is not an XML type, so you can execute XMLPARSE explicitly. Specify STRIP WHITESPACE to remove any boundary whitespace.

```
EXEC SQL BEGIN DECLARE SECTION;
   SQL TYPE BLOB (10K) blob_hostvar;
EXEC SQL END DECLARE SECTION;
…
EXEC SQL INSERT INTO MYCUSTOMER (Cid, Info)
  VALUES (1009,
  XMLPARSE(DOCUMENT :blob_hostvar STRIP WHITESPACE));
```

The document contains the `xml:space="default"` attribute, so the XMLPARSE specification of STRIP WHITESPACE controls whitespace handling. This means that the carriage returns, line feeds, and spaces between the elements in the document are removed.

If you retrieve the stored data, you see a single line with the following content:

```
<customerinfo xml:space="default" xmlns="http://posample.org" Cid='1009'>
<name>Kathy Smith</name><addr country='Canada'><street>15 Rosewood</street>
<city>Toronto</city><prov-state>Ontario</prov-state><pcode-zip>M6W 1E6</pcode-zip>
</addr><phone type='work'>416-555-4444</phone></customerinfo>
```

Example: parsing of a document with an internal DTD In a C language application, host variable clob_hostvar contains the following document, which contains an internal DTD:

```
<!DOCTYPE prod [<!ELEMENT description (name,details,price,weight)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT details (#PCDATA)>
   <!ELEMENT price (#PCDATA)>
   <!ELEMENT weight (#PCDATA)>
   <!ENTITY desc  "Anvil">
]>
<product xmlns="http://posample.org" pid=''110-100-01'' >
 <description>
 <name>&desc;</name>
 <details>Very heavy</details>
 <price>         9.99            </price>
 <weight>1 kg</weight>
 </description>
</product>
```

Insert the data into table MyProduct, which is a copy of the sample Product table:

```
EXEC SQL BEGIN DECLARE SECTION;
   SQL TYPE CLOB (10K) clob_hostvar;
```

```
EXEC SQL END DECLARE SECTION;
…
EXEC SQL insert into
  MyProduct ( pid, name, Price, PromoPrice, PromoStart, PromoEnd, description )
  values ( '110-100-01','Anvil', 9.99, 7.99, '11/02/2004','12/02/2004',
  XMLPARSE ( DOCUMENT :clob_hostvar STRIP WHITESPACE ));
```

XMLPARSE specifies stripping of whitespace, so boundary whitespace within the document is removed. In addition, when the database server executes XMLPARSE, it replaces the entity reference `&desc;` with its value.

If you retrieve the stored data, you see a single line with the following content:

```
<product xmlns="http://posample.org" pid="110-100-01"><description><name>Anvil
</name><details>Very heavy</details><price>         9.99            </price>
<weight>1 kg</weight></description></product>
```

XML schema validation

XML schema validation is the process of determining whether the structure, content, and data types of an XML document are valid according to an XML schema.

In addition, XML schema validation strips ignorable whitespace from the input document.

There are two ways that you can validate an XML document:

- Automatically, by including an XML type modifier in the XML column definition in a CREATE TABLE or ALTER TABLE statement. When a column has an XML type modifier, Db2 implicitly validates documents that are inserted into the column or documents in the column that are updated.
- Manually, by executing the DSN_XMLVALIDATE built-in function when you insert a document into an XML column or update a document in an XML column.

Validation is optional when you insert data into an XML column with no XML type modifier. Validation is mandatory when you insert data into an XML column with an XML type modifier.

Related concepts

Data model generation in XQuery
Before an XQuery expression can be processed, the input documents must be represented in the pureXML data model.

Related reference

DSN_XMLVALIDATE (Db2 SQL)

XML schema validation and ignorable whitespace

XML schema validation removes ignorable whitespace from a document.

According to the XML standard, *whitespace* is space characters (U+0020), carriage returns (U+000D), line feeds (U+000A), or tabs (U+0009) that are in the document to improve readability. When any of these characters appear as part of a text string, they are not considered to be whitespace.

Ignorable whitespace is whitespace that can be eliminated from the XML document. The XML schema document determines which whitespace is ignorable whitespace. If an XML document defines an element-only complex type (an element that contains only other elements), the whitespace between the elements is ignorable. If the XML schema defines a simple element that contains a non-string type, the whitespace within that element is ignorable.

Example: The `description` element in the sample product.xsd XML schema document is defined like this:

```
<xs:element name="description" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" minOccurs="0" />
      <xs:element name="details" type="xs:string" minOccurs="0" />
      <xs:element name="price" type="xs:decimal" minOccurs="0" />
      <xs:element name="weight" type="xs:string" minOccurs="0" />
```

```
 …
 </xs:complexType>
</xs:element>
```

The `description` element has an element-only complex type because it contains only other elements. Therefore, whitespace between elements in a `description` element is ignorable whitespace. The `price` element can also contain ignorable whitespace because it is a simple element that contains a non-string type.

Suppose that schema PRODUCT is registered in the XML schema repository (XSR).

Suppose that you create table MYPRODUCT like this:

```
CREATE TABLE MYPRODUCT (
 PID VARCHAR(10) NOT NULL PRIMARY KEY,
 NAME VARCHAR(128),
 PRICE DECIMAL(30,2),
 PROMOPRICE DECIMAL(30,2),
 PROMOSTART DATE,
 PROMOEND DATE,
 DESCRIPTION XML)
```

You insert the following document into XML column DESCRIPTION in the MYPRODUCT table, and you validate the XML data against the XML schema document product.xsd, which is located in the XML schema repository on the same database server as the MYPRODUCT table.

```
<product xmlns="http://posample.org" pid="110-100-01" >
 <description>
 <name>Anvil</name>
 <details>Very heavy</details>
 <price>         9.99            </price>
 <weight>1 kg</weight>
 </description>
</product>
```

When you retrieve the stored data, you can see that the XML schema validation process removes ignorable whitespace. The retrieved data is a single line with the following content:

```
<product xmlns="http://posample.org" pid="110-100-01"><description><name>Anvil
</name><details>Very heavy</details><price>9.99</price><weight>1 kg</weight>
</description></product>
```

The PRODUCT schema defines the whitespace around the `name`, `details`, `price`, and `weight` elements, and the whitespace within the price element as ignorable whitespace, so XML schema validation removes it.

XML schema validation with an XML type modifier

You can automate XML schema validation by adding an XML type modifier to an XML column definition.

An *XML type modifier* associates a set of one or more XML schemas with the XML data type. This type modifier enforces that all XML documents that are stored in an XML column are validated according to one of the XML schemas that are specified in the type modifier. Before schema validation through an XML type modifier can occur, all schema documents that make up an XML schema must be registered in the built-in XML schema repository (XSR).

You define an XML type modifier in a CREATE TABLE or ALTER TABLE statement as part of an XML column definition. The XML type modifier can identify more than one XML schema. You might want to associate more than one XML schema with an XML type modifier for the following reasons:

- The requirements for an XML schema evolve over time.

  An XML column might contain documents that describe only one type of information, but some fields in newer documents might need to be different from fields in the older documents. As new document versions are required, you can add new XML schemas to the XML type modifier.
- A single XML column contains XML documents of different kinds.

An XML column might contain documents that have several different formats. In this case, each type of document needs its own XML schema.

Alternatively, you might want to associate a single XML schema with multiple type modifiers. An XML schema can define many different documents. You might need to separate the XML documents into different columns, but specify the same XML schema in a type modifier for each column.

For example, a sales department might have one XML schema that defines purchase orders and billing statements. You can store purchase orders in one XML column, and billing statements in another XML column. Both XML columns have an XML type modifier that points to the same XML schema, but each column uses a different document in the XML schema.

Not all XML schemas that the XML type modifier identifies need to be registered before you execute the CREATE or ALTER statement. If the XML type modifier specifies a target namespace, only the XML schemas in that target namespace that exist when the CREATE or ALTER statement is executed are associated with the XML type modifier.

The following examples of defining an XML type modifier refer to these XML schemas.

XML schema name	Target namespace	Schema location	Timestamp when schema was registered in the XSR
PO1	http://www.example.com/PO1	http://www.example.com/PO1.xsd	2008-10-01 10:30:59.0100
PO2	http://www.example.com/PO2	http://www.example.com/PO2.xsd	2009-09-25 13:15:00.0200
PO3	No namespace	http://www.example.com/PO3.xsd	2009-06-25 13:15:00.0200
PO4	http://www.example.com/PO2	http://www.example.com/PO4.xsd	2009-10-25 13:15:00.0200

Example: Use the following SQL statement to create a table named PURCHASEORDERV1, with an XML type modifier on the CONTENT column. The XML type modifier uses the URI and LOCATION keywords to uniquely identify XML schema PO2.

```
CREATE TABLE PURCHASEORDERV1(
 ID INT NOT NULL,
 CONTENT XML(XMLSCHEMA URI 'http://www.example.com/PO2'
 LOCATION 'http://www.example.com/PO2.xsd'))
```

Example: Use the following SQL statement to create a table named PURCHASEORDERV2, with an XML type modifier on the CONTENT column. The XML type modifier uses the URI keyword to identify the XML schemas. If you execute the CREATE statement before 2009-10-25 13:15:00.0200, the XML type modifier identifies only XML schema PO2, and PO2 is used to validate any INSERT operations that are performed. Db2 does not add PO4 to the XML type modifier after PO4 is registered. If you execute the CREATE statement after 2009-10-25 13:15:00.0200, an SQL error occurs because the XML type modifier uses the URI keyword to identify two XML schemas: PO2 and PO4. The URI keyword must identify only one XML schema.

```
CREATE TABLE PURCHASEORDERV2(
 ID INT NOT NULL,
 CONTENT XML(XMLSCHEMA URI 'http://www.example.com/PO2'))
```

Example: Use the following SQL statement to create a table named PURCHASEORDERV3, with an XML type modifier on the CONTENT column that uses XML schema PO3. XML schema PO3 has no namespace, so you need to use the NO NAMESPACE keyword in the XML type modifier.

```
CREATE TABLE PURCHASEORDERV3(
 ID INT NOT NULL,
 CONTENT XML(XMLSCHEMA NO NAMESPACE
```

```
 LOCATION 'http://www.example.com/PO3.xsd'))
)
```

Example: Suppose that XML schema PO1 has two global elements: purchaseOrder and comment. Use the following SQL statement to create a table named PURCHASEORDERV4, with an XML type modifier on the CONTENT column that causes validation to be performed against the purchaseOrder element in XML schema PO1.

```
CREATE TABLE PURCHASEORDERV4(
 ID INT NOT NULL,
 CONTENT XML(XMLSCHEMA ID SYSXSR.PO1
 ELEMENT "purchaseOrder"))
```

Example: Use the following SQL statement to create a table named PURCHASEORDERV5, with an XML type modifier on the CONTENT column that includes multiple XML schemas: PO1, PO2, PO3, and PO4.

```
CREATE TABLE PURCHASEORDERV5(
 ID INT NOT NULL,
 CONTENT XML(XMLSCHEMA ID SYSXSR.PO1, ID SYSXSR.PO2,
 ID SYSXSR.PO3, ID SYSXSR.PO4))
```

Example: Suppose that new documents that will be added to table PURCHASEORDERV1 will have a new format. They need to conform to XML schema PO1. Alter the XML column to add PO1 to the XML type modifier:

```
ALTER TABLE PURCHASEORDERV1(
 ALTER CONTENT
 SET DATA TYPE XML(XMLSCHEMA
  ID SYSXSR.PO1,
  ID SYSXSR.PO2))
```

The table space that contains the XML documents for the CONTENT column is not put in CHECK-pending status, because all existing documents conform to XML schema SYSXSR.PO2.

Example: Suppose that you no longer want documents in the CONTENT column to conform to XML schema SYSXSR.PO2. Alter the XML column to remove SYSXSR.PO2 to the XML type modifier:

```
ALTER TABLE PURCHASEORDERV1
  ALTER CONTENT
  SET DATA TYPE XML(XMLSCHEMA
  ID SYSXSR.PO1))
```

The table space that contains the XML documents for the CONTENT column is put in CHECK-pending status, because all existing documents now need to conform only to XML schema SYSXSR.PO1.

Example: Suppose that you no longer need to do automatic validation of documents in the CONTENT column. Alter the column to remove the XML type modifier:

```
ALTER TABLE PURCHASEORDERV1
  ALTER CONTENT
  SET DATA TYPE XML
```

The table space that contains the XML documents for the CONTENT column is not put in CHECK-pending status, because there is no longer an XML schema to which the existing documents need to conform.

How Db2 chooses an XML schema from an XML type modifier

You can include more than one XML schema in an XML type modifier. When you insert into or update an XML column, Db2 chooses one XML schema to do validation.

Db2 uses the following process to determine which XML schema to use.

- If the operation is an update operation, and an XML schema that is specified by the XML type modifier has already been used to validate the original document, Db2 uses the same XML schema to validate the updated document.

- If there is only one XML schema whose target namespace matches the namespace name of the root element node in the document that is being validated (the XML *instance document*), Db2 chooses that XML schema to validate the XML document.
- If there is more than one XML schema with a target namespace that matches the namespace name of the root element, Db2 chooses an XML schema by using the *schema location hint*. The root element node of an XML instance document can contain an xsi:schemaLocation attribute. That attribute consists of one or more pairs of URI references, separated by white space. The first member of each pair is a namespace name, and the second member of the pair is a URI that describes where to find an appropriate schema document for that namespace. The second member of each pair is the schema location hint for the namespace name that is specified in the first member.

  For example, this is a schema location attribute:

  ```
  xsi:schemaLocation="http://www.example.com/PO2 http://www.example.com/PO4.xsd"
  ```

  The first member of the pair, http://www.example.com/PO2, is the namespace name. The second member of the pair, http://www.example.com/PO4.xsd, is the URI that provides the schema location hint.

  Db2 uses the schema location hint to choose an XML schema in the following way:

  1. If the root element node contains an xsi:schemaLocation attribute, Db2 searches the attribute value for a schema location hint with a corresponding namespace name that matches the namespace name in the root element node.
  2. If Db2 finds a schema location hint, Db2 uses the hint to identify an XML schema whose schema location URI is identical to the schema location hint. Db2 validates the input document against that schema.
  3. If the root element does not contain an xsi:schemaLocation attribute, or the xsi:schemaLocation attribute does not contain a schema location hint with a corresponding namespace name that matches the namespace name in the root element node, Db2 uses the XML schema with the same target namespace and the latest registration timestamp.

  Recommendation: Include a schema location hint in instance documents, to simplify identification of the correct XML schema.
- If the root element of the XML instance document does not have a namespace name, only XML schemas with no target namespace are candidates for use in validation. Db2 chooses an XML schema in the following way:

  1. If a single XML schema in the XML type modifier has no target namespace, Db2 uses that XML schema for validation.
  2. If more than one XML schema in the XML type modifier has no target namespace, and the XML instance document contains an xsi:noNamespaceSchemaLocation attribute, Db2 uses the value of xsi:noNamespaceSchemaLocation, which is the schema location hint, to choose an XML schema.
  3. If the root element does not contain the xsi:noNamespaceSchemaLocation attribute, or the schema location hint does not match the schema location URI of any XML schema in the XML type modifier, Db2 uses the XML schema with the latest registration timestamp from those XML schemas that have no target namespace.

Example: Suppose that the XML schema repository (XSR) contains the following XML schemas.

XML schema name	Target namespace	Schema location	Timestamp when schema was registered in the XSR
PO1	http://www.example.com/PO1	http://www.example.com/PO1.xsd	2008-10-01 10:30:59.0100
PO2	http://www.example.com/PO2	http://www.example.com/PO2.xsd	2009-09-25 13:15:00.0200

XML schema name	Target namespace	Schema location	Timestamp when schema was registered in the XSR
PO3	No namespace	http://www.example.com/PO3.xsd	2009-06-25 13:15:00.0200
PO4	http:// www.example.com/PO2	http://www.example.com/PO4.xsd	2009-10-25 13:15:00.0200

Also suppose that table PURCHASEORDERV5 is defined like this:

```
CREATE TABLE PURCHASEORDERV5(
 ID INT NOT NULL,
 CONTENT XML(XMLSCHEMA ID SYSXSR.PO1, ID SYSXSR.PO2,
 ID SYSXSR.PO3, ID SYSXSR.PO4))
```

You execute the following INSERT statements:

```
INSERT INTO PURCHASEORDERV5 VALUES(1,                                    1
 '<po:purchaseOrder xmlns:po="http://www.example.com/PO1">
 …
 </po:purchaseOrder>'
);
INSERT INTO PURCHASEORDERV5 VALUES(2,                                    2
 '<po:purchaseOrder xmlns:po="http://www.example.com/PO2"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.example.com/PO2
 http://www.example.com/PO2.xsd">
 …
 </po:purchaseOrder>');
INSERT INTO PURCHASEORDERV5 VALUES(2,                                    3
 '<po:purchaseOrder xmlns:po="http://www.example.com/PO2"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.example.com/PO2
 http://www.example.com/PO4.xsd">
 …
 </po:purchaseOrder>');
INSERT INTO purchase_orders VALUES(3,                                    4
 '<purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://www.example.com/PO3.xsd">
 …
 </purchaseOrder>');
```

The following table lists the XML schemas that Db2 uses, and the reasons for those choices.

Insert statement number	XML schemas that are used for validation, in order of choice	Reason
1	PO1	The namespace name in the root element of the instance document is http://www.example.com/PO1. This name matches only the target namespace for XML schema PO1.
2	PO2, PO4	The namespace name in the root element in the instance document is http:// www.example.com/PO2. which matches the target namespace of XML schemas PO2 and PO4. The root element of the instance document also contains an xsi:schemaLocation attribute whose value provides the schema location hint http://www.example.com/PO2.xsd. The schema location hint matches the schema location for XML schema PO2. Therefore Db2 chooses PO2 to validate the instance document. If validation with PO2 fails, Db2 uses PO4.

Insert statement number	XML schemas that are used for validation, in order of choice	Reason
3	PO4, PO2	The namespace name in the root element in the instance document is http:// www.example.com/PO2. which matches the target namespace of XML schemas PO2 and PO4. The root element of the instance document also contains an xsi:schemaLocation attribute whose value provides the schema location hint http://www.example.com/PO4.xsd. The schema location hint matches the schema location for XML schema PO4. Therefore Db2 chooses PO4 to validate the instance document. If validation with PO4 fails, Db2 uses PO2.
4	PO3	The root element of the instance document has no namespace name. XML schema PO3 has no target namespace. Therefore, Db2 uses PO3 for validation.

Revalidation after XML document updates

After you update an XML document in a column that has an XML type modifier, Db2 revalidates all or part of the document.

If the XML type modifier includes several XML schemas, Db2 uses the same XML schema for revalidation that it used for the original validation.

If you update an entire document, Db2 revalidates the entire document. However, if you use the XMLMODIFY function to update only a portion of the document, Db2 might need to validate only the updated portion.

The following table lists the rules that Db2 uses for determining how much of a document to revalidate.

XMLMODIFY option	Revalidation behavior
insert nodes	Revalidation behavior depends on the source expression and the type of insert operation: • If the source expression is a sequence of attribute nodes, Db2 revalidates each attribute in that sequence under its new parent and the parent of the target node. • If the operation is insert nodes...into, Db2 revalidates the parent of the target node. • If the operation is insert nodes...before or insert nodes...after, Db2 revalidates the parent of the target node.
delete nodes	Db2 revalidates from the common ancestor of the deleted nodes.
replace node	Db2 revalidates the parent of the target node.
replace value of node	Db2 revalidates the target node and the parent of the target node.
Any update operation	Db2 revalidates from the target node's topmost ancestor with an xsi:type attribute.

Because Db2 revalidates only the changed portion of an XML document, some constraints that are defined in the XML schema on the instance document cannot be enforced during revalidation if they require other portions of the instance document. Those constraints are:

- Indicators that enforce uniqueness of elements or attributes

  If a uniqueness constraint is specified on an ancestor of a node that is to be revalidated, the constraint is not enforced.
- ID and IDREF attributes

Db2 does not validate ID and IDREF attributes during revalidation.

- Key and keyref elements

  If a key element and a corresponding keyref element appear in the node that is to be revalidated, Db2 validates them. If the key element or the keyref element appear elsewhere in the document, Db2 does not validate them.

Example: Suppose that table PURCHASEORDERV4 is defined like this:

```
CREATE TABLE PURCHASEORDERV4(
 ID INT NOT NULL,
 CONTENT XML(XMLSCHEMA ID SYSXSR.PO1
 ELEMENT "purchaseOrder"))
```

Insert a row into the PURCHASEORDERV4 table:

```
INSERT INTO PURCHASEORDERV4 VALUES(1,
 '<po:purchaseOrder xmlns:po="http://www.example.com/PO1">
   <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>95160</postcode>
   </shipTo>
 …
   <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
   </items>
 </po:purchaseOrder>
);
```

Use XMLMODIFY to make the following updates:

XMLMODIFY operation: Add a <shipDate> element to the first <item> element in the document in column CONTENT:

```
UPDATE PURCHASEORDERV4
 SET CONTENT =
  XMLMODIFY ('declare namespace po="http://www.example.com/PO1";
  insert node $x into /po:purchaseOrder/items/item[1]',
  XMLELEMENT(name "shipDate", '2009-01-20') as "x")
 WHERE ID = 1;
```

Resulting revalidation: Db2 revalidates the <items> element in the document.

XMLMODIFY operation: Add a country attribute to the <shipTo> element in the document in column CONTENT:

```
UPDATE PURCHASEORDERV4
 SET CONTENT =
  XMLMODIFY (
  'declare namespace po="http://www.example.com/PO1";
  insert node $x/@country into /po:purchaseOrder/shipTo',
  XMLELEMENT(name "shipTo",
  XMLATTRIBUTES('US' as "country")) as "x")
 WHERE ID = 1;
```

Resulting revalidation: Db2 revalidates only the shipTo element in the document.

XMLMODIFY operation: Replace the value of the <shipDate> element in the first <item> element:

```
UPDATE PURCHASEORDERV4
 SET content = XMLMODIFY (
  'declare namespace po="http://www.example.com/PO1";
  replace value of node
  /po:purchaseOrder/items/item[1]/shipDate
  with "2009-02-15"')
 WHERE ID = 1;
```

Resulting revalidation: Db2 revalidates only the first <item> element.

XMLMODIFY operation: Delete the second <item> element in the document.

```
UPDATE PURCHASEORDERV4
 SET content = XMLMODIFY('delete nodes //item[2]')
 WHERE ID = 1;
```

Resulting revalidation: Db2 revalidates only the <items> element, which is the parent node of the deleted <item> element.

XML schema validation with DSN_XMLVALIDATE

One way to do XML schema validation is by executing the DSN_XMLVALIDATE built-in function.

Before you can invoke DSN_XMLVALIDATE, all schema documents that make up an XML schema must be registered in the built-in XML schema repository (XSR). An XML schema provides the rules for a valid XML document.

DSN_XMLVALIDATE returns a value with the XML data type.

There are a number of forms of DSN_XMLVALIDATE:

```
DSN_XMLVALIDATE(string-expression)
DSN_XMLVALIDATE(xml-expression)
DSN_XMLVALIDATE(string-expression, varchar-expression)
DSN_XMLVALIDATE(xml-expression, varchar-expression)
DSN_XMLVALIDATE(string-expression1, string-expression2, string-expression3)
DSN_XMLVALIDATE(xml-expression1, string-expression2, string-expression3)
```

For all forms, the first parameter contains the document that you want to validate.

For forms with one parameter, the target namespace and optional schema location of the XML schema must be in the root element of the instance document that you want to validate.

For forms with two parameters, the second parameter is the name of the schema object to use for validation of the document. That object must be registered in the XML schema repository.

For forms with three parameters, the second and third parameter contain the names of a namespace URI and a schema location hint that identify the XML schema object to use for validation of the document. That object must be registered in the XML schema repository.

Related reference

DSN_XMLVALIDATE (Db2 SQL)

Moving from SYSFUN.DSN_XMLVALIDATE to SYSIBM.DSN_XMLVALIDATE

There are two versions of DSN_XMLVALIDATE: a user-defined function and a built-in function. The user-defined function is deprecated. You should use the built-in function instead.

Procedure

To move from the DSN_XMLVALIDATE user-defined function to the DSN_XMLVALIDATE built-in function:

1. For applications that invoke DSN_XMLVALIDATE using the qualified name SYSFUN.DSN_XMLVALIDATE:
   a) Change the name to SYSIBM.DSN_XMLVALIDATE.
   b) Prepare the applications again.

2. For applications that invoke DSN_XMLVALIDATE without using the qualified name, you do not need to modify the applications. Db2 automatically uses the SYSIBM.DSN_XMLVALIDATE built-in function.
3. Optional: Remove the XMLPARSE function that surrounds DSN_XMLVALIDATE.

   The SYSFUN.DSN_XMLVALIDATE user-defined function must be invoked from within the XMLPARSE function. The SYSIBM.DSN_XMLVALIDATE built-in function does not need to be invoked from within the XMLPARSE function.

Example

Suppose that an application calls the SYSFUN.DSN_XMLVALIDATE user-defined function:

```
EXEC SQL INSERT INTO T1(C1) VALUES (XMLPARSE (DOCUMENT
  SYSFUN.DSN_XMLVALIDATE(:xmldoc, 'SYSXSR.MYXMLSCHEMA')));
```

Update the INSERT statement like this to call the SYSIBM.DSN_XMLVALIDATE built-in function:

```
EXEC SQL INSERT INTO T1(C1) VALUES (
  SYSIBM.DSN_XMLVALIDATE(:xmldoc, 'SYSXSR.MYXMLSCHEMA'));
```

How Db2 chooses an XML schema for DSN_XMLVALIDATE

When you execute DSN_XMLVALIDATE as part of inserting into or updating an XML column, Db2 chooses one XML schema to do validation.

Db2 uses the following process to determine which XML schema to use.

- If the DSN_XMLVALIDATE invocation includes an XML schema name, Db2 uses the XML schema that is uniquely identified by the XML schema name.
- If the DSN_XMLVALIDATE invocation includes a target namespace or a schema location hint, or both, Db2 searches the Db2 XML schema repository (XSR) for an XML schema name that corresponds to the combination of the target namespace and schema location hint.
  - If exactly one XML schema name corresponds to the combination of the target namespace and schema location hint, Db2 uses that XML schema name.
  - If multiple XML schema names correspond to the combination of the target namespace and schema location hint, Db2 uses the XML schema with the most recent registration timestamp.
  - If no XML schema name corresponds to the combination of the target namespace and schema location hint, Db2 issues an error.
- If the DSN_XMLVALIDATE invocation does not specify an XML schema name or target namespace and schema location hint, Db2 examines the *instance document* (the document that is being validated) to determine the XML schema.
  1. Db2 determines a target namespace and a schema location hint from the instance document, in the following way:
     - If the root element node in the instance document contains a namespace name, Db2 uses that namespace name as the target namespace name.
     - If the root element node in the instance document does not contain a namespace name, Db2 does not use a target namespace name.
     - If the root element node in the instance document contains an `xsi:schemaLocation` attribute, Db2 uses its value as a schema location hint.
     - If the root element node in the instance document contains an `xsi:noNamespaceSchemaLocation` attribute, Db2 uses its value as the schema location hint for schemas with no target namespace.
  2. Db2 searches the Db2 XML schema repository (XSR) for an XML schema name that corresponds to the target namespace, schema location hint, or both, from the instance document.
     - If exactly one XML schema name corresponds to the combination of the target namespace and schema location hint, Db2 uses that XML schema name.

- If multiple XML schema names correspond to the combination of the target namespace and schema location hint, Db2 uses the XML schema with the most recent registration timestamp.
- If no XML schema name corresponds to the combination of the target namespace and schema location hint, Db2 issues an error.

Example: Suppose that the XML schema repository (XSR) contains the following XML schemas.

XML schema name	Target namespace	Schema location	Timestamp when schema was registered in the XSR
PO1	http://www.example.com/PO1	http://www.example.com/PO1.xsd	2008-10-01 10:30:59.0100
PO2	http://www.example.com/PO2	http://www.example.com/PO2.xsd	2009-09-25 13:15:00.0200
PO3	No namespace	http://www.example.com/PO3.xsd	2009-06-25 13:15:00.0200
PO4	http://www.example.com/PO2	http://www.example.com/PO4.xsd	2009-10-25 13:15:00.0200

Also suppose that table PURCHASEORDERV5 is defined like this:

```
CREATE TABLE PURCHASEORDERV5(
 ID INT NOT NULL,
 CONTENT XML)
```

You execute the following INSERT statements:

```
INSERT INTO PURCHASEORDERV5 VALUES(1,                          1
 DSN_XMLVALIDATE('<po:purchaseOrder xmlns:po="http://www.example.com/PO1">
 …
 </po:purchaseOrder>')
);
INSERT INTO PURCHASEORDERV5 VALUES(2,                          2
 DSN_XMLVALIDATE('<po:purchaseOrder xmlns:po="http://www.example.com/PO2"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.example.com/PO2
 http://www.example.com/PO2.xsd">
 …
 </po:purchaseOrder>','SYSXSR.PO2')
);
INSERT INTO PURCHASEORDERV5 VALUES(2,                          3
 DSN_XMLVALIDATE('<po:purchaseOrder xmlns:po="http://www.example.com/PO2"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.example.com/PO2
 http://www.example.com/PO4.xsd">
 …
 </po:purchaseOrder>',
 'http://www.example.com/PO2')
);
INSERT INTO purchase_orders VALUES(3,                          4
 DSN_XMLVALIDATE(
 '<purchaseOrder
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://www.example.com/PO3.xsd">
 …
 </purchaseOrder>')
);
```

The following table lists the XML schema that Db2 uses, and the reasons for that choice.

Insert statement number	XML schemas that are used for validation, in order of choice	Reason
1	PO1	The DSN_XMLVALIDATE invocation does not specify an XML schema or target namespace and schema location hint, so Db2 uses the information in the instance document. The namespace name in the root element of the instance document is http://www.example.com/PO1. This name matches only the target namespace for XML schema PO1.
2	PO2	The DSN_XMLVALIDATE invocation specifies XML schema SYSXSR.PO2.
3	PO4	The DSN_XMLVALIDATE invocation specifies namespace http://www.example.com/PO2. Two XML schemas, PO2 and PO4, have that target namespace. Db2 uses PO4, because it has the later timestamp.
4	PO3	The DSN_XMLVALIDATE invocation does not specify an XML schema or target namespace and schema location hint, so Db2 uses the information in the instance document. The root element node in the instance document contains an `xsi:noNamespaceSchemaLocation` attribute with value http://www.example.com/PO3.xsd, so Db2 uses XML schema PO3, which has no target namespace, and the schema location http://www.example.com/PO3.xsd.

How to determine whether an XML document has been validated

You can use the SQL XMLXSROBJECTID scalar function to determine whether an XML document that is stored in a table has undergone XML validation, and which XML schema was used to validate that document.

XMLXSROBJECTID returns the XSR object identifier of the XML schema that was used to validate the input XML document. The XSR object identifier corresponds to the XSROBJECTID column in the SYSIBM.XSROBJECTS catalog table. After you call XMLXSROBJECTID, you can use the returned value to query SYSIBM.XSROBJECTS for the XML schema information. If the XML document has not been validated, XMLXSROBJECTID returns 0.

Examples

Example: The following SQL statement calls XMLXSROBJECTID to determine which XML documents in the INFO column of the CUSTOMER table have not been validated. The statement then calls DSN_XMLVALIDATE to validate those documents against XML schema SYSXSR.P01.

```
UPDATE CUSTOMER
 SET INFO = DSN_XMLVALIDATE(INFO, 'SYSXSR.PO1')
 WHERE XMLXSROBJECTID(INFO)=0
```

Example: The following SQL statement retrieves the target namespaces and XML schema names for the XML schemas that were used to validate XML documents in the INFO column of the CUSTOMER table.

```
SELECT DISTINCT S.XSROBJECTNAME, S.TARGETNAMESPACE
 FROM CUSTOMER C, XSROBJECTS S
 WHERE XMLXSROBJECTID(INFO) = S.XSROBJECTID
```

Casts between XML data types and SQL data types

Casting of SQL data types to XML data types occurs implicitly. The XMLCAST specification simplifies the task of converting an XML value to an SQL data type.

For casting from an XML schema data type to an SQL data type, the XMLCAST specification eliminates the need to serialize an XML value before casting it to an SQL data type. Casting of an XML schema data type to an SQL data type is useful if you want to return the results of the XMLQUERY function as SQL for further processing in an SQL statement, such as comparing XML values to SQL values or using XML values to order result tables.

You can cast the result of XMLQUERY to an SQL data type only when the XQuery expression that is specified in the XMLQUERY function returns a sequence that contains one item.

Implicit casting of an SQL data type to an XML schema type occurs in the XMLEXISTS predicate or the XMLQUERY or XMLTABLE function, when a column value is passed to an XQuery expression. Although you can use the XMLCAST specification to cast an SQL data type to an XML schema data type, it is usually unnecessary.

Related concepts

XMLEXISTS predicate for querying XML data
The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.

XMLQUERY function for retrieval of portions of an XML document
XMLQUERY is an SQL scalar function that lets you execute an XQuery expression from within an SQL context.

XMLCAST in SQLJ applications (Db2 Application Programming for Java)

Casting between data types (Db2 SQL)

Related reference

XMLCAST specification (Db2 SQL)

Examples of casts from XML schema data types to SQL data types

You can use the XMLCAST specification to cast an XML schema data type to an SQL data type.

Example: casting to compare XML values to SQL values

Suppose that the PORDER column of the PURCHASEORDER table contains the following document:

```
<PurchaseOrder xmlns="http://posample.org"  PoNum="5000"
  OrderDate="2007-02-18" Status="Unshipped">
   <item>
       <partid>100-100-01</partid>
       <name>Snow Shovel, Basic 22"</name>
       <quantity>3</quantity>
       <price>9.99</price>
   </item>
   <item>
       <partid>100-103-01</partid>
       <name>Snow Shovel, Super Deluxe 26" Wide</name>
       <quantity>5</quantity>
       <price>49.99</price>
   </item>
</PurchaseOrder>
```

Use the XMLCAST specification to cast a value with the xs:string type to a value with the VARCHAR type so that you can compare the values.

```
SELECT P.POID FROM PURCHASEORDER P, PRODUCT R WHERE
  R.PID =
    XMLCAST(XMLQUERY(
   'declare default element namespace "http://posample.org";
   $d/PurchaseOrder/item[name="Snow Shovel, Basic 22"]/partid'
   PASSING P.PORDER AS "d") AS VARCHAR(10))
```

The result table of the SELECT statement is:

POID
5000

Example: casting to order a result table

Suppose that the sample PRODUCT table has the following rows:

PID	DESCRIPTION
100-100-01	<product xmlns="http://posample.org" pid="100-100-01" > <description> <name>Snow Shovel, Basic 22"</name> <details>Basic Snow Shovel, 22" wide, straight handle with D-Grip</details> <price>9.99</price> <weight>1 kg</weight> </description> </product>
100-101-01	<product xmlns="http://posample.org" pid="100-101-01" > <description> <name>Snow Shovel, Deluxe 24"</name> <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with D-Grip</details> <price>19.99</price> <weight>2 kg</weight> </description> </product
100-103-01	<product xmlns="http://posample.org" pid="100-103-01" > <description> <name>Snow Shovel, Super Deluxe 26"</name> <details>Super Deluxe Snow Shovel, 26" wide, ergonomic battery heated curved handle with upgraded D-Grip</details> <price>49.99</price> <weight>3 kg</weight> </description> </product>
100-201-01	<product xmlns="http://posample.org" pid="100-201-01" > <description> <name>Ice Scraper, Windshield 4" Wide</name> <details>Basic Ice Scraper 4" wide, foam handle</details> <price>3.99</price> </description> </product>

Use the XMLCAST specification to cast values in XML documents in the sample PRODUCT table to an SQL data type so that you can use those values in an ORDER BY clause.

```
SELECT PID
 FROM PRODUCT
 ORDER BY
  XMLCAST(XMLQUERY ('declare default element namespace "http://posample.org";
  $d/product/description/name'
  PASSING DESCRIPTION AS "d") AS VARCHAR(128))
```

The result table of the SELECT statement is:

PID
100-201-01
100-100-01

PID
100-101-01
100-103-01

Example: casting an attribute node for retrieval of an attribute value

When you retrieve an attribute value, you cannot serialize the attribute node directly. For example, this query results in an error:

```
SELECT XMLQUERY('declare default element namespace "http://posample.org";
/PurchaseOrder/@PoNum' PASSING PORDER)
 FROM PURCHASEORDER
 WHERE POID=5000
```

Instead, you can select the attribute node, and then use XMLCAST to serialize the selected node before retrieving the attribute value. For example:

```
SELECT XMLCAST(XMLQUERY('declare default element namespace "http://posample.org";
/PurchaseOrder/@PoNum' PASSING PORDER) as INT)
 FROM PURCHASEORDER
 WHERE POID=5000
```

Example: casting xs:date values to the DATE type

If you cast an xs:date value without a time zone component to a DATE type, the result is the same as the input. For example:

```
SELECT XMLCAST(XMLQUERY(' "2007-10-12" ') AS DATE)
 FROM SYSIBM.SYSDUMMY1
```

The result is 2007-10-12.

If you cast an xs:date value with a time zone component to a DATE type, the result is adjusted to UTC time. For example:

```
SELECT XMLCAST(XMLQUERY(' "2007-10-12+13:00" ') AS DATE)
 FROM SYSIBM.SYSDUMMY1
```

The input value is assumed to be at the beginning of the day on 2007-10-12, in a time zone that is 13 hours ahead of UTC time. Therefore, after the date is adjusted to UTC time, the result is 2007-10-11.

Example: casting xs:time values to the TIME type

If you cast an xs:time value without a time zone component to a TIME type, the result is the input value with the fractional part truncated. For example:

```
SELECT XMLCAST(XMLQUERY(' "13:20:15.054" ') AS TIME)
 FROM SYSIBM.SYSDUMMY1
```

The result is 13:20:15.

If you cast an xs:time value with a time zone component to a TIME type, the result is adjusted to UTC time, and the fractional part is truncated. For example:

```
SELECT XMLCAST(XMLQUERY(' "13:20:15.054+05:00" ') AS TIME)
 FROM SYSIBM.SYSDUMMY1
```

The input value is in a time zone that is five hours ahead of UTC time. Therefore, after the time is adjusted to UTC time, the result is 08:20:15.

Example: casting xs:dateTime values to the TIMESTAMP or TIMESTAMP(*p*) type

If you cast an xs:dateTime value without a time zone component to a TIMESTAMP(*p*) type, and the number of fractional digits in the input value is less than *p*, the result is the input value with the fractional part expanded to *p* digits. If *p* is not specified, the default precision for the result is 6. For example:

```
SELECT XMLCAST(XMLQUERY('xs:dateTime("2009-03-25T05:01:01.123456789")')
  AS TIMESTAMP(12))
 FROM SYSIBM.SYSDUMMY1
```

The result is 2009-03-25 05:01:01.123456789000.

If you cast an xs:dateTime value without a time zone component to a TIMESTAMP(*p*) type, and the number of fractional digits in the input value is greater than *p*, the result is the input value with the fractional part truncated to *p* digits. If *p* is not specified, the default precision for the result is 6. For example:

```
SELECT XMLCAST(XMLQUERY('xs:dateTime("2009-03-25T05:01:01.123456789")')
  AS TIMESTAMP)
 FROM SYSIBM.SYSDUMMY1
```

The result is 2009-03-25 05:01:01.123456.

If you cast an xs:dateTime value with a time zone component to a TIMESTAMP(*p*) type, and the number of fractional digits in the input value is less than *p*, the result is adjusted to UTC time, and the fractional part is expanded to *p* digits. If *p* is not specified, the default precision for the result is 6. For example:

```
SELECT XMLCAST(XMLQUERY(' "2009-03-25T05:01:01.123456789+08:00" ')
  AS TIMESTAMP(12))
 FROM SYSIBM.SYSDUMMY1
```

The input value is in a time zone that is eight hours ahead of UTC time. Therefore, after the date and time are adjusted to UTC time, the result is 2009-03-24 21:01:01.123456789000.

If you cast an xs:dateTime value with a time zone component to a TIMESTAMP(*p*) type, and the number of fractional digits in the input value is greater than *p*, the result is adjusted to UTC time, and the fractional part is truncated to *p* digits. If *p* is not specified, the default precision for the result is 6. For example:

```
SELECT XMLCAST(XMLQUERY(' "2009-03-25T05:01:01.123456789+08:00" ')
  AS TIMESTAMP)
 FROM SYSIBM.SYSDUMMY1
```

The input value is in a time zone that is eight hours ahead of UTC time. Therefore, after the date and time are adjusted to UTC time, the result is 2009-03-24 21:01:01.123456.

Retrieving XML data

You can retrieve entire XML documents from XML columns by using an SQL SELECT statement. Alternatively, you can use SQL with XML extensions (SQL/XML) to retrieve portions of documents.

The SQL/XML functions that supports retrieval of portions of XML documents are XMLQUERY and XMLTABLE. To filter table rows by XML document content, use the SQL/XML XMLEXISTS predicate.

Related concepts

XML serialization

XML serialization is the process of converting XML data from its internal representation in a Db2 table to the textual XML format that it has in an application.

Retrieval of an entire XML document from an XML column

Retrieval of an entire XML document from an XML column is similar to retrieval of a value from a LOB column.

Retrieval of an entire XML document is different from retrieval of a LOB value in the following ways:

- LOB values can have locators, but XML values cannot.

This means that you need to do one of these things:

- Allocate enough application storage to retrieve entire XML values, use file reference variables, or use the SQL FETCH WITH CONTINUE statement.
- Invoke XMLSERIALIZE to convert the XML column data into a CLOB or BLOB data type, and fetch the result into a LOB locator.

- XML values can have internal encoding as well as external encoding.

  With LOB values, you need to consider only whether there are differences between the database server encoding and the application encoding when you retrieve data. XML values can have internal encoding as well as application encoding (external encoding).

Related reference
FETCH (Db2 SQL)

XMLQUERY function for retrieval of portions of an XML document

XMLQUERY is an SQL scalar function that lets you execute an XQuery expression from within an SQL context.

You can pass variables to the XQuery expression specified in XMLQUERY. XMLQUERY returns an XML value, which is an XML sequence. This sequence can be empty or can contain one or more items.

When you execute XQuery expressions from within an XMLQUERY function, you can:

- Retrieve parts of stored XML documents, instead of entire XML documents.
- Enable XML data to participate in SQL queries.
- Operate on both relational and XML data in the same SQL statement.
- Apply further SQL processing to the returned XML values (for example, ordering results with the ORDER BY clause of a subselect), after you use XMLCAST to cast the results to a non-XML type.

XQuery is case-sensitive, so you need to ensure that the case of variables that you specify in an XMLQUERY function and in its XQuery expression match.

Related reference
XMLQUERY (Db2 SQL)

Non-empty sequences returned by XMLQUERY

If evaluation of the XQuery expression that you specify in XMLQUERY results in a non-empty sequence, the XMLQUERY function returns that sequence.

Example: Suppose that two of the documents in the INFO column of the sample CUSTOMER table look like these:

```
<customerinfo xmlns="http://posample.org" Cid="1002">
  <name>Jim Noodle</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
</customerinfo>

<customerinfo xmlns="http://posample.org" Cid="1003">
  <name>Robert Shoemaker</name>
    <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
  <phone type="home">416-555-2937</phone>
  <phone type="cell">905-555-8743</phone>
```

```
  <phone type="cottage">613-555-3278</phone>
</customerinfo>
```

You execute the following statement:

```
SELECT CID, XMLQUERY ('declare default element namespace "http://posample.org";
  /customerinfo/phone' passing INFO)
 AS "PHONE FROM INFO"
 FROM CUSTOMER
 WHERE CID IN (1002,1003)
```

The result table contains the following two rows:

Table 9. Example of a result table from XMLQUERY that returns non-empty sequences

CID	PHONE FROM INFO
1002	<?xml version="1.0" encoding="IBM037"?><phone xmlns="http://posample.org" type="work">905-555-7258</phone>
1003	<?xml version="1.0" encoding="IBM037"?><phone xmlns="http://posample.org" type="work">905-555-7258</phone><phone xmlns="http://posample.org" type="home">416-555-2937</phone><phone xmlns="http://posample.org" type="cell">905-555-8743</phone><phone xmlns="http://posample.org" type="cottage">613-555-3278</phone>

The first row contains a sequence of one <phone> element, and the second row has a sequence of four <phone> elements. This result demonstrates that when XMLQUERY returns a sequence that contains more than one element, the serialization process concatenates the elements into a single string. The result in the second row is not a well-formed document. Ensure that any application that receives this result can properly handle this behavior.

Related reference

XMLQUERY (Db2 SQL)

Empty sequences returned by XMLQUERY

XMLQUERY returns an empty sequence if the XQuery expression returns an empty sequence.

Example: In the following query, XMLQUERY returns an empty sequence for each row of the CUSTOMER table that does not have a <city> element with a value of Aurora in the INFO column.

```
SELECT Cid, XMLQUERY ('declare default element namespace "http://posample.org";
  //addr[city="Aurora"]' passing INFO)
AS "ADDRESS FROM INFO"
FROM CUSTOMER
```

Only one XML document that contains a <city> element with the value of Aurora. The following table results from the previous SELECT statement:

Table 10. Example of a result table from XMLQUERY that returns empty sequences

CID	ADDRESS FROM INFO
1000	<?xml version="1.0" encoding="IBM037"?>
1001	<?xml version="1.0" encoding="IBM037"?>
1002	<?xml version="1.0" encoding="IBM037"?>
1003	<?xml version="1.0" encoding="IBM037"?><addr xmlns="http://posample.org" country="Canada"><street>1596 Baseline</street><city>Aurora</city><prov-state>Ontario</prov-state><pcode-zip>N8X-7F8</pcode-zip></addr>
1004	<?xml version="1.0" encoding="IBM037"?>

Table 10. Example of a result table from XMLQUERY that returns empty sequences (continued)	
CID	**ADDRESS FROM INFO**
1005	<?xml version="1.0" encoding="IBM037"?>

Empty sequences are returned for rows that do not have a <city> element with the value of Aurora. Empty sequences are returned as strings of length 0, rather than NULL values. The <addr> element is returned in the third row, however, because it satisfies the XQuery expression.

You can filter out the rows with empty sequences by applying a predicate, such as the XMLEXISTS predicate.

Example: Rewrite the previous query to return only rows with non-empty sequences:

```
SELECT Cid, XMLQUERY ('declare default element namespace "http://posample.org";
                       /customerinfo/addr' passing c.INFO)
 AS "ADDRESS FROM INFO"
FROM Customer as c
WHERE XMLEXISTS ('declare default element namespace "http://posample.org";
                  //addr[city="Aurora"]' passing c.INFO)
```

The table that results from this query is as follows:

Table 11. Result table	
CID	**ADDRESS FROM INFO**
1003	<?xml version="1.0" encoding="IBM037"?><addr xmlns="http://posample.org" country="Canada"><street>1596 Baseline</street><city>Aurora</city><prov-state>Ontario</prov-state><pcode-zip>N8X-7F8</pcode-zip></addr>

Related concepts

XMLEXISTS predicate for querying XML data

The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.

Related reference

XMLQUERY (Db2 SQL)

XMLEXISTS predicate for querying XML data

The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.

The XMLEXISTS predicate specifies an XQuery expression. If the XQuery expression returns an empty sequence, the value of the XMLEXISTS predicate is false. Otherwise, XMLEXISTS returns true. Rows that correspond to an XMLEXISTS value of true are returned.

XQuery is case-sensitive, so you need to ensure that the variables that you specify in an XMLEXISTS function and in its XQuery expression have the same case.

Important: Because a logical expression or a comparison expression always results in a boolean value of true or false, an XMLEXISTS predicate with a top-level logical expression or comparison expression is always true. For example, the following XMLEXISTS predicate is always true:

```
XMLEXISTS('//addr/city="Toronto"')
```

This is usually not the result that you need when you use an XMLEXISTS predicate to filter results.

Example of an XMLEXISTS predicate: Return only those rows from the sample CUSTOMER table for which the <city> value in the INFO column is Toronto and the Cid attribute value in the INFO column is

1004. The XMLEXISTS predicates in the following SELECT statement return true for the appropriate INFO values.

```
SELECT Cid, Info
FROM CUSTOMER
WHERE XMLEXISTS ('declare default element namespace "http://posample.org";
                 //addr[city="Toronto"]' passing INFO)
AND XMLEXISTS ('declare default element namespace "http://posample.org";
                 /customerinfo[@Cid="1004"]' passing INFO)
```

The result table contains the following rows:

Table 12. Example of a result table from a SELECT with an XMLEXISTS predicate

CID	INFO
1004	<?xml version="1.0" encoding="IBM037"?><customerinfo xmlns="http://posample.org" Cid=''1004''><name>Matt Foreman</name> <addr country=''Canada''><street>1596 Baseline</street> <city>Toronto</city><prov-state>Ontario</prov-state> <pcode-zip>M3Z-5H9</pcode-zip></addr> <phone type=''work''>905-555-4789</phone> <phone type=''home''>416-555-3376</phone> <assistant><name>Gopher Runner</name> <phone type=''home''>416-555-3426</phone> </assistant> </customerinfo>
1005	<?xml version="1.0" encoding="IBM037"?><customerinfo xmlns="http://posample.org" Cid=''1004''><name>Matt Foreman</name> <addr country=''Canada''><street>1596 Baseline</street> <city>Toronto</city><prov-state>Ontario</prov-state> <pcode-zip>M3Z-5H9</pcode-zip></addr> <phone type=''work''>905-555-4789</phone> <phone type=''home''>416-555-3376</phone> <assistant><name>Gopher Runner</name> <phone type=''home''>416-555-3426</phone> </assistant> </customerinfo>

Related reference

XMLEXISTS predicate (Db2 SQL)

Constant and parameter marker passing to XMLEXISTS and XMLQUERY

You can specify XQuery variables as part of the XQuery expression in the XMLEXISTS predicate and the XMLQUERY function.

You pass the values into these variables through the `passing` clause. These values are SQL expressions.

Because the values that are passed to the XQuery expression are non-XML values, they must be cast, either implicitly or explicitly, to types that are supported by XQuery.

Example: Implicit casting

In the following query, the SQL character string constant 'Aurora', which is not an XML type, is implicitly cast to an XML type in the XMLEXISTS predicate. Following the implicit cast, the constant has the XML schema subtype of xs:string, and is bound to the variable $cityName. The $cityName variable can then be used in the predicate of the XQuery expression.

```
SELECT XMLQUERY ('declare default element namespace "http://posample.org";
                  $d/customerinfo/addr' passing c.INFO as "d")
FROM Customer as c
WHERE XMLEXISTS('declare default element namespace "http://posample.org";
                 $d//addr[city=$cityName]'
                 passing c.INFO as "d",
                 'Aurora' AS "cityName")
```

Example: Explicit casting

In the following query, the XMLCAST specification casts a string constant to the XML data type.

```
SELECT XMLQUERY ('declare default element namespace "http://posample.org";
                  $d/customerinfo/addr' passing c.INFO as "d")
FROM Customer as c
WHERE XMLEXISTS('declare default element namespace "http://posample.org";
```

```
                $d//addr[city=$cityName]'
                passing c.INFO as "d",
                XMLCAST('San Jose' as XML) AS "cityName")
```

Related concepts

XMLQUERY function for retrieval of portions of an XML document

XMLQUERY is an SQL scalar function that lets you execute an XQuery expression from within an SQL context.

Related reference

XMLQUERY (Db2 SQL)

XMLEXISTS predicate (Db2 SQL)

XMLTABLE function for returning XQuery results as a table

The XMLTABLE SQL/XML function returns a table from the evaluation of XQuery expressions.

XQuery expressions normally return values as a sequence. However, XMLTABLE lets you execute an XQuery expression and return values as a table. The table that is returned can contain columns of any SQL data type, including the XML data type.

You can pass variables to the *row* XQuery expression that is specified in XMLTABLE. The result of the row XQuery expression defines the portions of an XML document that you use to define a row of the returned table. You specify *column* XQuery expressions in the COLUMNS clause of the XMLTABLE function to generate the column values of the resulting table. In the COLUMNS clause, you define characteristics of a column by specifying the column name, data type, and how the column value is generated. Alternatively, you can omit the COLUMNS clause and let Db2 generate a single, unnamed XML column.

You can include an XMLNAMESPACES function as the first argument of XMLTABLE, to specify the XML namespace for all XQuery expressions in the XMLTABLE function. Namespace declarations in individual XQuery expressions override the XMLNAMESPACES argument.

You can specify the contents of a result table column through a column XQuery expression that you specify in the PATH clause of XMLTABLE. If you do not specify a PATH clause, Db2 uses the result table column name as the PATH argument. For example, if a result table column name is @partNum, and the input XML documents must have an attribute named partNum, the result table column values are the values of the partNum attribute.

If the column XQuery expression that defines a result table column returns an empty sequence, XMLTABLE returns a NULL value in the result table column. XMLTABLE lets you supply a default value instead of a NULL value. You do this by specifying a DEFAULT clause in the column definition.

If you want to generate a sequence number for each row that XMLTABLE generates, you can include a column definition with the FOR ORDINALITY clause. The FOR ORDINALITY clause causes XMLTABLE to generate a column with values that start at 1. If a single document generates more than one row, the sequence number is incremented by 1. For example, if an XML document generates three result table rows, the sequence numbers for those rows are 1, 2, and 3.

Important: If the column XQuery expression that is specified in the PATH option of XMLTABLE returns a sequence of more than one item, the data type of the result table column must be XML.

Related reference

XMLTABLE (Db2 SQL)

XMLTABLE advantages

In certain situations, XQuery expression results are easier to process if they are in a table than if they are in a sequence.

Returning a table instead of a sequence enables the following operations to be performed from within an SQL query context:

- Iteration over results of an XQuery expression from within an SQL fullselect

For example, in the following query, the SQL fullselect iterates over the table that results from executing the XQuery expression //customerinfo in XMLTABLE.

```
SELECT X.*
 FROM CUSTOMER,
 XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
   '//customerinfo'
   PASSING CUSTOMER.INFO
    COLUMNS
       "CUSTNAME" VARCHAR(30) PATH 'name',
       "CITY"     VARCHAR(30) PATH 'addr/city') X
```

- Insertion of values from stored XML documents into tables

  This technique is a simple form of decomposition, where decomposition is the process of storing fragments of an XML document in columns of relational tables.

- Individual processing of items in a sequence

  If you return the items in a sequence as a single row, with each item in a separate column, it is easier to process the individual items.

- Sorting on values from an XML document

  For example, in the following query, results are sorted by the customer names that are stored in XML documents in the INFO column of the CUSTOMER table.

```
SELECT X.*
 FROM CUSTOMER,
 XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
   '//customerinfo'
   PASSING CUSTOMER.INFO
    COLUMNS
       "CUSTNAME" VARCHAR(30) PATH 'name',
       "CITY"     VARCHAR(30) PATH 'addr/city') X
    ORDER BY X.CUSTNAME
```

- Storing of some XML values as relational and some values as XML

Related concepts

XMLTABLE example: Inserting values returned from XMLTABLE

The XMLTABLE SQL/XML function can be used to retrieve values from within stored XML documents. The retrieved values can then be inserted into a table.

XMLTABLE example: Inserting values returned from XMLTABLE

The XMLTABLE SQL/XML function can be used to retrieve values from within stored XML documents. The retrieved values can then be inserted into a table.

For example, the following XML documents are stored in the sample CUSTOMER table:

```
<customerinfo xmlns="http://posample.org" Cid="1001">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
</customerinfo>

<customerinfo xmlns="http://posample.org" Cid="1003">
  <name>Robert Shoemaker</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X-7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
  <phone type="home">416-555-2937</phone>
  <phone type="cell">905-555-8743</phone>
```

```
  <phone type="cottage">613-555-3278</phone>
</customerinfo>
```

You want to insert values from these documents into a table with the following definition:

```
CREATE TABLE CUSTADDR (CUSTNAME VARCHAR(30),
                       CUSTSTREET VARCHAR(30),
                       CUSTCITY VARCHAR(30)
                       CUSTSTATE VARCHAR(30),
                       CUSTZIP VARCHAR(30))
```

The following INSERT statement, which uses XMLTABLE, populates CUSTADDR with values from the XML documents:

```
INSERT INTO CUSTADDR
  SELECT X.*
  FROM CUSTOMER,
  XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
  '//customerinfo'
  PASSING CUSTOMER.INFO
              COLUMNS
                  "CUSTNAME" VARCHAR(30) PATH 'name',
                  "CUSTSTREET" VARCHAR(30) PATH 'addr/street',
                  "CUSTCITY" VARCHAR(30) PATH 'addr/city',
                  "CUSTSTATE" VARCHAR(30) PATH 'addr/prov-state',
                  "CUSTZIP" VARCHAR(30) PATH 'addr/pcode-zip'
               ) as X
```

After you execute the INSERT statement, the CUSTADDR table looks like this:

Table 13. Contents of the CUSTADDR table after insert of a result table generated by XMLTABLE

CUSTNAME	CUSTSTREET	CUSTCITY	CUSTSTATE	CUSTZIP
Kathy Smith	25 EastCreek	Markham	Ontario	N9C 3T6
Robert Shoemaker	1596 Baseline	Aurora	Ontario	N8X-7F8

XMLTABLE example: Returning one row for each occurrence of an item

If an XML document contains multiple occurrences of an element, you can use XMLTABLE to generate a row for each occurrence of the element.

For example, the following XML documents are stored in the sample CUSTOMER table:

```
<customerinfo xmlns="http://posample.org" Cid="1001">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
</customerinfo>

<customerinfo xmlns="http://posample.org" Cid="1003">
  <name>Robert Shoemaker</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
  <phone type="home">416-555-2937</phone>
  <phone type="cell">905-555-8743</phone>
  <phone type="cottage">613-555-3278</phone>
</customerinfo>
```

You can use a query like this to create a table in which every phone value is returned in a separate row:

```
SELECT X.*
FROM CUSTOMER C, XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
```

```
  '$cust/customerinfo/phone' PASSING C.INFO as "cust"
  COLUMNS "CUSTNAME" VARCHAR(30) PATH '../name',
  "PHONETYPE" VARCHAR(30) PATH '@type',
  "PHONENUM" VARCHAR(15) PATH '.') as X
  WHERE CID=1001 OR CID=1003
```

The result table of the SELECT statement is:

Table 14. Result table from a query that uses XMLTABLE to retrieve multiple occurrences of an item

CUSTNAME	PHONETYPE	PHONENUM
Kathy Smith	work	905-555-7258
Robert Shoemaker	work	905-555-7258
Robert Shoemaker	home	416-555-2937
Robert Shoemaker	cell	905-555-8743
Robert Shoemaker	cottage	613-555-3278

The following SELECT statement returns each phone element as an XML document, instead of a string value:

```
SELECT X.*
FROM CUSTOMER C, XMLTABLE (xmlnamespaces (DEFAULT 'http://posample.org'),
  '$cust/customerinfo/phone' PASSING C.INFO as "cust"
  COLUMNS "CUSTNAME" CHAR(30) PATH '../name',
  "PHONETYPE" CHAR(30) PATH '@type',
  "PHONENUM" XML PATH '.') as X
  WHERE CID=1001 OR CID=1003
```

This query yields the following results for the two XML documents:

Table 15. Result table

CUSTNAME	PHONETYPE	PHONENUM
Kathy Smith	work	<phone xmlns="http://posample.org" type="work">905-555-7258</phone>
Robert Shoemaker	work	<phone xmlns="http://posample.org" type="work">905-555-7258</phone>
Robert Shoemaker	home	<phone xmlns="http://posample.org" type="work">416-555-2937</phone>
Robert Shoemaker	cell	<phone xmlns="http://posample.org" type="work">905-555-8743</phone>
Robert Shoemaker	cottage	<phone xmlns="http://posample.org" type="work">613-555-3278</phone>

XMLTABLE example: Specifying a default value for a column in the result table

You can specify a default value for any column in the XMLTABLE result table by using a DEFAULT clause. The default value is used if the XQuery expression that defines the column returns an empty sequence.

For example, the following XML documents are stored in the sample CUSTOMER table. Neither document has an age element.

```
<customerinfo xmlns="http://posample.org" Cid="1001">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
```

```
  <phone type="work">905-555-7258</phone>
</customerinfo>

<customerinfo xmlns="http://posample.org" Cid="1003">
  <name>Robert Shoemaker</name>
    <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
  <phone type="home">416-555-2937</phone>
  <phone type="cell">905-555-8743</phone>
  <phone type="cottage">613-555-3278</phone>
</customerinfo>
```

You can use a query like this to create a result table in which the column value is "***No age***" if a document has no age for a customer:

```
SELECT X.*
  FROM CUSTOMER C, XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
  '$cust/customerinfo' PASSING C.INFO as "cust"
  COLUMNS "CUSTNAME" VARCHAR(30) PATH './name',
  "AGE" VARCHAR(30) PATH './age' DEFAULT '***No age***') AS X
  WHERE CID=1001 OR CID=1003
```

The result table of the SELECT statement is:

Table 16. Result table from a query in which XMLTABLE has a default value for an item

CUSTNAME	AGE
Kathy Smith	***No age***
Robert Shoemaker	***No age***

XMLTABLE example: Specifying an ordinality column in the result table

You can specify that the result table of an XMLTABLE invocation includes an ordinality column.

An ordinality column has the following properties:

- Has the BIGINT data type
- Starts at one for each document that generates a result table row
- Is incremented by one for each result table row that is generated by a single document

For example, the following XML document is stored in the sample CUSTOMER table.

```
<customerinfo xmlns="http://posample.org" Cid="1003">
  <name>Robert Shoemaker</name>
    <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X 7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
  <phone type="home">416-555-2937</phone>
  <phone type="cell">905-555-8743</phone>
  <phone type="cottage">613-555-3278</phone>
</customerinfo>
```

You can use a query like this to create an ordinality column in the XMLTABLE result table:

```
SELECT X.*
  FROM CUSTOMER C, XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
  '$cust/customerinfo/phone' PASSING C.INFO as "cust"
  COLUMNS "SEQNO" FOR ORDINALITY,
  "PHONE TYPE" VARCHAR(15) PATH './@type',
  "PHONE NUMBER" VARCHAR(15) PATH '.'
  ) AS X
  WHERE CID=1003
```

The result table of the SELECT statement is:

Table 17. Result table from a query in which XMLTABLE has an ordinality column

SEQNO	PHONE TYPE	PHONE NUMBER
1	work	905-555-7258
2	home	416-555-2937
3	cell	905-555-8743
4	cottage	613-555-3278

XML support in native SQL routines

Native SQL routines support parameters and variables with the XML data type.

XML parameters can be used in SQL statements in the same way as variables of any other data type. In addition, variables with the XML data type can be passed as parameters to XQuery expressions in XMLEXISTS, XMLQUERY and XMLTABLE expressions.

An XML value returned from a remote site within an SQL procedural language stored procedure must be well-formed.

Example: native SQL procedure The following code demonstrates the declaration, use, and assignment of XML parameters and variables in a native SQL procedure. The example uses table T1, which has one column named C1, which has the XML data type.

```
CREATE PROCEDURE PROC1(IN PARM1 XML, IN PARM2 VARCHAR(32000))
 LANGUAGE SQL
 BEGIN
  DECLARE var1 XML;                                                  1
  IF(XMLEXISTS('$x/ITEM[value < 200]' passing by ref PARM1 as "x"))THEN
   INSERT INTO T1 VALUES(PARM1);                                     2
    END IF;
  SET var1 =                                                         3
   XMLDOCUMENT(XMLELEMENT(NAME "ORDER",
    XMLCONCAT(PARM1, var1)));
  INSERT INTO T1 VALUES(var1);                                       4
 END #
```

The SQL procedure performs the following operations on XML parameters and variables:

1. Declares an XML variable named var1.
2. Checks whether the value of XML parameter PARM1 contains an item with a value less than 200. If it does, the SQL procedure inserts the XML value into column C1 in table T1.
3. Concatenates the contents of PARM1 and var1, creates an element named ORDER that contains the concatenated content, returns the ORDER element as a document node, and assigns that document node to XML variable var1.
4. Inserts the value that is in XML variable var1 into column C1 in table T1.

Example: non-inline SQL function The following code demonstrates the declaration, use, and assignment of XML parameters and variables in a non-inline SQL scalar function. This function takes two parameters as input: an XML document that contains book order information, with prices in U.S. dollars or Canadian, and the monetary exchange rate. The function returns an XML document that contains the prices in Canadian dollars.

```
CREATE FUNCTION CANOrder(BOOKORDER XML, USTOCANRATE double)
 RETURNS XML
 DETERMINISTIC
 NO EXTERNAL ACTION
 CONTAINS SQL
 BEGIN ATOMIC
  DECLARE USPrice decimal(15,2);
  DECLARE CANPrice decimal(15,2);
  DECLARE OrderInCAN XML;                                    1
  SET USPrice = XMLCAST(XMLQUERY('/bookorder/USprice' PASSING BOOKORDER)
```

```
   AS decimal(15,2));                                          2
  SET CANPrice = XMLCAST(XMLQUERY('/bookorder/CANprice' PASSING BOOKORDER)
   AS decimal(15,2));                                          3
  IF CANPrice is NULL or CANPrice <=0 THEN
   IF USPrice >0 THEN
    SET CANPrice = USPrice * USTOCANRATE;
   ELSE
    SET CANPrice = 0;
  END IF;
  SET OrderInCAN =                                             4
   XMLDOCUMENT(
    XMLELEMENT(NAME "bookorder",
    XMLQUERY('/bookorder/bookname' PASSING BOOKORDER),
    XMLELEMENT(NAME "CANprice", CANPrice))
   );
  RETURN OrderInCAN;
 END#
```

The SQL function performs the following operations:

1. Declares an XML variable named OrderInCAN, which will hold the order with prices in Canadian dollars that is returned to the caller.
2. Retrieves the U.S. price from the input document, which is in the BOOKORDER parameter.
3. Looks for a Canadian price in the input document. If the document contains no Canadian prices, the XMLCAST function on the XMLQUERY function returns NULL.
4. Builds the output document, whose top-level element is bookorder, by concatenating the bookname element from the original order with a CANprice element, which contains the calculated price in Canadian dollars.

Suppose that an input document looks like this:

```
<bookorder>
 <bookname>TINTIN</bookname>
 <USprice>100.00</USprice>
</bookorder>
```

If the exchange rate is 0.9808 Canadian dollars for one U.S. dollar, the output document looks like this:

```
<bookorder><bookname>TINTIN</bookname><CANprice>9.81</CANprice></bookorder>
```

Example: SQL table function: The following code demonstrates the declaration, use, and assignment of XML parameters and variables in an SQL table function. This function takes three parameters as input: an XML document that contains order information, a maximum price for the order, and the title of the book that is ordered. The function returns a table that contains an XML column with receipts that are generated from all of the input parameters, and a BIGINT column that contains the order IDs that are retrieved from the input parameter that contains the order information document.

```
CREATE FUNCTION ORDERTABLE
 (ORDERDOC XML, PRICE decimal(15,2), BOOKTITLE varchar(50))
 RETURNS TABLE (RECEIPT XML, ORDERID BIGINT)
 LANGUAGE SQL
 SPECIFIC ORDERTABLE
 NOT DETERMINISTIC
 READS SQL DATA
 RETURN
  SELECT ORDER.RECEIPT, ORDER.ID
   FROM XMLTABLE(XMLNAMESPACES(DEFAULT 'http://posample.org'),
   '/orderdoc/bookorder[USprice < $A and bookname = $B]'
   PASSING ORDERDOC, PRICE as A, BOOKTITLE as B
   COLUMNS
    ID BIGINT PATH '@OrderID',
    RECEIPT XML PATH '.')
   AS ORDER;
```

The SQL table function uses the XMLTABLE function to generate the result table for the table that is returned by the function. The XMLTABLE function generates a row for each ORDERDOC input document in which the title matches the book title in the BOOKTITLE input parameter, and the price is less than the value in the PRICE input parameter. The columns of the returned table are the Receipt node of the ORDERDOC input document, and the OrderID element from the ORDERDOC input document.

Suppose that the input parameters have these values:

PRICE: 200, BOOKTITLE: TINTIN, ORDERDOC:

```
<orderdoc xmlns="http://posample.org" OrderID="5001">
 <name>Jim Noodle</name>
 <addr country="Canada">
  <street>25 EastCreek</street>
  <city>Markham</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>N9C-3T6</pcode-zip>
 </addr>
 <phone type="work">905-555-7258</phone>
 <bookorder>
  <bookname>TINTIN</bookname>
  <USprice>100.00</USprice>
 </bookorder>
</orderdoc>
```

The returned table looks like this:

ID	RECEIPT
5001	<orderdoc xmlns="http://posample.org" OrderID="5001"> <name>Jim Noodle</name> <addr country="Canada"> <street>25 EastCreek</street> <city>Markham</city> <prov-state>Ontario</prov-state> <pcode-zip>N9C-3T6</pcode-zip> </addr> <phone type="work">905-555-7258</phone> <bookorder> <bookname>TINTIN</bookname> <USprice>100.00</USprice> </bookorder> </orderdoc>

Requests for data in XML columns by earlier Db2 clients

If you retrieve data from an XML column to a client that is at an earlier release than DB2 9, your database client cannot handle XML data.

During DRDA processing, when the database server recognizes a client that cannot support XML data, by default, the Db2 database server describes XML data values as BLOB values and sends the data to the client as BLOB data. The BLOB data is the textual XML representation of the XML data, with a complete XML declaration.

If you want to receive the data as a CLOB or DBCLOB data type, instead of a BLOB data type, you can invoke the XMLSERIALIZE function on the column data to instruct the Db2 database server to convert the data to the specified data type before it sends the data to the client.

When you do not invoke XMLSERIALIZE to retrieve data from a database server to a client at an earlier release level, the column from which you retrieve the data does not behave exactly like a BLOB column. For example, although you can use the LIKE predicate on a BLOB column, you cannot use the LIKE predicate on an XML column that returns BLOB data.

Related concepts

XML serialization

XML serialization is the process of converting XML data from its internal representation in a Db2 table to the textual XML format that it has in an application.

Functions for constructing XML values

Several SQL/XML publishing functions can be used together to publish relational data in XML format.

Those functions are:

XMLAGG aggregate function
Returns an XML sequence containing an item for each non-null value in a set of XML values.

XMLATTRIBUTES scalar function
Constructs XML attributes from the arguments. This function can only be used as an argument of the XMLELEMENT function.

XMLCOMMENT scalar function
Returns an XML value with a single comment node with the input argument as the content.

XMLCONCAT scalar function
Returns a sequence containing the concatenation of a variable number of XML input arguments.

XMLDOCUMENT scalar function
Returns an XML value with a single document node with zero or more children nodes.

This function creates a document node, which by definition, every XML document must have. A document node is not visible in the textual representation of XML, however, every document that is to be stored in a Db2 table must contain a document node.

It is not necessary to call XMLDOCUMENT when you insert an XML value into an XML column, or update an XML column with an XML value. Db2 implicitly adds the document node for you. For example, the following INSERT statement explicitly calls XMLDOCUMENT to insert value COL2:

```
INSERT INTO T1 (INT1,XML1)
SELECT X.COL1, XMLDOCUMENT(X.COL2)
FROM XMLTABLE('/A/B' PASSING CAST (? AS XML)
       COLUMNS COL1 INTEGER PATH '@id',
               COL2 XML PATH '.') X;
```

However, you can omit XMLDOCUMENT, and Db2 implicitly adds the document node for COL2:

```
INSERT INTO T1 (INT1,XML1)
SELECT X.COL1, X.COL2
FROM XMLTABLE('/A/B' PASSING CAST (? AS XML)
       COLUMNS COL1 INTEGER PATH '@id',
               COL2 XML PATH '.') X;
```

XMLELEMENT scalar function
Returns an XML value that is an XML element node.

XMLFOREST scalar function
Returns an XML value that is a sequence of XML element nodes.

XMLNAMESPACES declaration
Constructs namespace declarations from the arguments. This declaration can only be used as an argument of the XMLELEMENT or XMLFOREST functions.

XMLPI scalar function
Returns an XML value with a single processing instruction node.

XMLTEXT scalar function
Returns an XML value with a single text node having the input argument as the content.

You can combine these functions to construct XML values that contain different types of nodes. You need to specify the functions in the order in which you want the corresponding elements to appear.

Example

Suppose that you want to construct the following document, which has constant values:

```
<elem1 xmlns="http://posample.org" id="111">
  <!-- example document -->
  <child1>abc</child1>
  <child2>def</child2>
</elem1>
```

The document consists of:

- Three element nodes (`elem1`, `child1`, and `child2`)
- A namespace declaration
- An `id` attribute on `elem1`
- A comment node

To construct this document, you need to invoke publishing functions in the following order:

1. Create an element node named `elem1`, using XMLELEMENT.
2. Add a default namespace declaration to the XMLELEMENT function call for `elem1`, using XMLNAMESPACES.
3. Create an attribute named `id` using XMLATTRIBUTES, placing it after the XMLNAMESPACES declaration.
4. Create a comment node using XMLCOMMENT, within the XMLELEMENT function call for `elem1`.
5. Create a sequence of element nodes that are named `child1` and `child2`, using the XMLFOREST function, within the XMLELEMENT function call for `elem1`.

The following SELECT statement constructs the document:

```
SELECT XMLELEMENT (NAME "elem1",
  XMLNAMESPACES (DEFAULT 'http://posample.org'),
  XMLATTRIBUTES ('111' AS "id"),
  XMLCOMMENT (' example document '),
  XMLFOREST('abc' as "child1",
  'def' as "child2"))
  FROM SYSIBM.SYSDUMMY1
```

Example

Suppose that you want to construct an XML document from `name` elements in the Description column of the sample Product table.

The documents in the Description column look similar to this one:

```
<product xmlns="http://posample.org" pid="100-100-01">
 <description>
  <name>Snow Shovel, Basic 22"</name>
  <details>Basic Snow Shovel, 22" wide, straight handle with D-Grip</details>
  <price>9.99</price>
  <weight>1 kg</weight>
 </description>
</product>
```

You want the constructed document to look like this:

```
<allProducts xmlns="http://posample.org">
  <item>Snow Shovel, Basic 22"</item>
  <item>Snow Shovel, Deluxe 24"</item>
  <item>Snow Shovel, Super Deluxe 26"</item>
  <item>Ice Scraper, Windshield 4" Wide</item>
</allProducts>
```

The document consists of:

- Five element nodes (`allProducts`, and four `item` elements)
- A namespace declaration

To construct this document, you need to invoke publishing functions in the following order:

1. Create an element node named `allProducts`, using XMLELEMENT.
2. Add a default namespace declaration to the XMLELEMENT function call for `allProducts`, using XMLNAMESPACES.
3. Create a sequence of element `item` nodes, using the XMLELEMENT function within the XMLAGG function call.

The following SELECT statement constructs the document:

```
SELECT XMLELEMENT (NAME "allProducts",
                   XMLNAMESPACES (DEFAULT 'http://posample.org'),
                   XMLAGG(XMLELEMENT (NAME "item", p.name)))
FROM Product p
```

Example

Suppose that you want to construct an XML document from name elements in documents in the Description column of the sample Product table, and Quantity column values of the sample Inventory table. The join column for the Product and Quantity tables is the Pid column.

You want the generated XML document to look like this:

```
<saleProducts xmlns="http://posample.org">
  <prod id="100-100-01">
    <name>Snow Shovel 22"</name>
    <numInStock>5</numInStock>
  </prod>
  <prod id="100-101-01">
    <name>Snow Shovel 24"</name>
    <numInStock>25</numInStock>
  </prod>
  <prod id="100-103-01">
    <name>Deluxe Snow Shovel 26"</name>
    <numInStock>55</numInStock>
  </prod>
  <prod id="100-201-01">
    <name>Ice Scraper 4"</name>
    <numInStock>99</numInStock>
  </prod>
</saleProducts>
```

The document consists of:

- Thirteen element nodes (saleProducts, and four prod elements, each of which contains a name element and a numInStock element)
- a namespace declaration

To construct this document, you need to invoke publishing functions in the following order:

1. Create an element node named saleProducts, using XMLELEMENT.
2. Add a default namespace declaration to the XMLELEMENT function call for saleProducts, using XMLNAMESPACES.
3. Create a sequence of element prod nodes, using the following function invocations within the XMLAGG function invocation to construct the prod nodes:
   a. XMLELEMENT, to create prod node
   b. XMLATTRIBUTES, to add an id attribute to each prod element
   c. XMLFOREST, to construct the name and numInStock elements

The following SELECT statement constructs the document:

```
SELECT XMLELEMENT (NAME "saleProducts",
                   XMLNAMESPACES (DEFAULT 'http://posample.org'),
                   XMLAGG (XMLELEMENT (NAME "prod",
                           XMLATTRIBUTES (p.Pid AS "id"),
                           XMLFOREST (p.name as "name",
                                      i.quantity as "numInStock"))))
FROM PRODUCT p, INVENTORY i
WHERE p.Pid = i.Pid
```

Example

When you construct an XML value using XMLELEMENT or XMLFOREST, you might encounter NULL values in the source tables. You can use the EMPTY ON NULL and NULL ON NULL options of XMLELEMENT and XMLFOREST to specify whether to generate an empty element or no element when the functions encounter a NULL value. The default NULL handling for XMLELEMENT is EMPTY ON NULL. The default NULL handling for XMLFOREST is NULL ON NULL.

Suppose that the LOCATION column of the INVENTORY table contains a NULL value in one row. You want to construct elements from the LOCATION column that contain an empty sequence if the LOCATION value is NULL. The following SELECT statement does that:

```
SELECT XMLELEMENT (NAME "newElem",
                   XMLATTRIBUTES (PID AS "prodID"),
                   XMLFOREST (QUANTITY as "quantity",
                              LOCATION as "loc" OPTION EMPTY ON NULL))
FROM INVENTORY
```

Related reference

XMLAGG (Db2 SQL)
XMLATTRIBUTES (Db2 SQL)
XMLCOMMENT (Db2 SQL)
XMLCONCAT (Db2 SQL)
XMLDOCUMENT (Db2 SQL)
XMLELEMENT (Db2 SQL)
XMLFOREST (Db2 SQL)
XMLNAMESPACES (Db2 SQL)
XMLPI (Db2 SQL)
XMLTEXT (Db2 SQL)

Special character handling in SQL/XML publishing functions

The SQL/XML publishing functions make substitutions for some special characters when SQL values and identifiers are converted to XML values.

Special character handling for SQL values

Certain characters are considered special characters in XML documents. Those characters must appear in their escaped format, using their entity representation. Those special characters are:

Table 18. Special characters and their entity representations

Special character	Entity representation
<	<
>	>
&	&
"	" (in attribute values only)

When you publish SQL values as XML values using the SQL/XML publishing functions, the functions replace those special characters with their predefined entities.

Special character handling for SQL identifiers

When you construct XML values from SQL values, you might need to map an SQL identifier to an XML qualified name, or QName. The set of characters that are allowed in delimited SQL identifiers differs, however, from those that are permitted in a QName. This difference means that some characters that are valid for SQL identifiers are not valid in QNames.

For example, consider the delimited SQL identifier "phone@work". Because the @ character is not a valid character in a QName, the character is escaped, and the QName becomes phone_x0040_work.

This default escape behavior applies only to column names. For SQL identifiers that are provided as the element name in XMLELEMENT, or as alias names in the AS clause of XMLFOREST and XMLATTRIBUTES, there are no escape defaults. You must provide valid QNames in these cases.

Related concepts

Functions for constructing XML values

Several SQL/XML publishing functions can be used together to publish relational data in XML format.

XML serialization

XML serialization is the process of converting XML data from its internal representation in a Db2 table to the textual XML format that it has in an application.

You can invoke the XMLSERIALIZE function, to request that the Db2 database server perform XML serialization before it sends the XML data to the client application. This process is called explicit serialization. Alternatively, you can omit the XMLSERIALIZE call, and retrieve data from an XML column directly into application variables. The Db2 database server serializes the XML data during retrieval. This process is called implicit serialization.

With implicit serialization, the data has the XML type when it is sent to the client, if the client supports the XML data type. For Db2 ODBC and embedded SQL applications, the Db2 database server adds an XML declaration, with the appropriate encoding specification, to the data. For Java applications, the Db2 database server does not add an XML declaration, unless you use the deprecated `DB2Xml.getDB2Xmlxxx` methods to retrieve the data.

Implicit serialization is the preferred method in most cases. Sending XML data to the client allows the Db2 client to handle the XML data properly. Explicit serialization requires additional handling by the client.

You can retrieve XML data in the binary XML format (Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format), rather than as textual XML data, to avoid serialization. Retrieval of data in the binary XML format is supported only in JDBC, SQLJ, or ODBC applications, or by the UNLOAD utility.

After an explicit XMLSERIALIZE invocation, the data has a non-XML data type in the database server, and is sent to the client as that data type.

XMLSERIALIZE lets you specify:

- The SQL data type to which the data is converted when it is serialized

  The data type is a CLOB, BLOB, DBCLOB data type.
- Whether the output data should include the following explicit XML declaration (EXCLUDING XMLDECLARATION or INCLUDING XMLDECLARATION):

```
<?xml version="1.0" encoding="UTF-8"?>
```

The output from XMLSERIALIZE is Unicode UTF-8-encoded data.

Be sure that you understand the implications of requesting an explicit encoding specification when you execute XMLSERIALIZE. If you retrieve the textual XML data into a non-binary data type, the data is converted to the application encoding, but the encoding specification is not modified. Therefore, the encoding of the data might not agree with the encoding specification. This situation results in XML data that cannot be parsed by application processes that rely on the encoding name.

In general, implicit serialization is preferable. However, under the following circumstances, it is better to do an explicit XMLSERIALIZE:

- When XML documents are very large

  Because there are no XML locators, if the XML documents are very large, you can use XMLSERIALIZE to convert the data to a LOB type so that you can use LOB locators.
- When the client does not support XML data

  If the client is an earlier version that does not support the XML data type, and you use implicit XML serialization, the Db2 database server converts the data to the BLOB data type. If you want the retrieved data to be some other data type, you can dynamically execute an SQL statement that invokes XMLSERIALIZE to specify CLOB or DBCLOB output.
- When you want to pass XML data to a stored procedure or user-defined function

Db2 for z/OS stored procedures and user-defined functions do not support parameters with the XML data type. Therefore, if you want to pass data from an XML column to a routine, you need to invoke XMLSERIALIZE on the data to convert it to a BLOB, CLOB, or DBCLOB type.

The best data type to which to convert XML data is the BLOB data type, because retrieval of binary data results in fewer encoding issues.

Example: XML column Info in sample table Customer contains a document that contains the hierarchical equivalent of the following data:

```
<customerinfo xml:space="default" xmlns="http://posample.org" Cid='1000'>
  <name>Kathy Smith</name>
  <addr country='Canada'>
  <street>5 Rosewood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>M6W 1E6</pcode-zip>
  </addr>
  <phone type='work'>416-555-1358</phone>
</customerinfo>
```

Invoke XMLSERIALIZE to serialize the data and convert it to a BLOB type before retrieving it into a host variable.

```
SELECT XMLSERIALIZE(Info as BLOB(1M)) from Customer
  WHERE CID='1000'
```

Example: In a C program, retrieve the customerinfo document for customer ID 1000 into an XML as BLOB host variable. Doing this results in implicit XML serialization. The retrieved data is in the UTF-8 encoding scheme, and it contains an XML declaration.

```
EXEC SQL BEGIN DECLARE SECTION;
 SQL TYPE IS XML AS BLOB (1M) xmlCustInfo;
EXEC SQL END DECLARE SECTION;
…
EXEC SQL SELECT INFO INTO :xmlCustInfo
  FROM Customer
  WHERE Cid=1000;
```

Example: This JDBC program avoids XML serialization by retrieving the data in the binary data format. The program sets the `DataSource` property xmlFormat to indicate that the data should be retrieved in the binary XML format. Then the program retrieves the customerinfo document for customer ID 1000 into an SQLXML object. Next, the program retrieves the data from the SQLXML object into a `DOMSource` object, so that the retrieved data is in a non-textual representation. This technique requires JDBC 4.0 or later.

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // IBM Data Server Driver for JDBC
                                            // and SQLJ implementation of JDBC
…
com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
  new com.ibm.db2.jcc.DB2SimpleDataSource();
                                            // Create the DataSource object
db2ds.setDriverType(4);                     // Set the driver type
db2ds.setDatabaseName("san_jose");          // Set the location
db2ds.setUser("db2adm");                    // Set the user
db2ds.setPassword("db2adm");                // Set the password
db2ds.setServerName("mvs1.sj.ibm.com");
                                            // Set the server name
db2ds.setPortNumber(5021);                  // Set the port number
db2ds.setXMLFormat(
  com.ibm.db2.jcc.DB2BaseDataSource.XML_FORMAT_BINARY);
                                            // Set XML format to binary
…
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT INFO FROM Customer WHERE Cid='1000'");
SQLXML sqlxml = rs.getSQLXML(1);
DOMSource domSource = sqlxml.getSource(DOMSource.class);
                                          // Get a DOMSource object from
                                          // the SQLXML object, to avoid
                                          // XML serialization
Document document = (Document) domSource.getNode();
```

Related concepts

Retrieving XML data

You can retrieve entire XML documents from XML columns by using an SQL SELECT statement. Alternatively, you can use SQL with XML extensions (SQL/XML) to retrieve portions of documents.

Differences in an XML document after storage and retrieval

When you store an XML document in a table and then retrieve that document, the retrieved document might not be the same as the original document.

This behavior is defined by the XML and SQL/XML standard.

Some of the changes to the document occur when the document is stored. Those changes are:

- If you execute DSN_XMLVALIDATE, the database server strips ignorable whitespace from the input document.
- If you do not request XML schema validation, the database server:
  - Strips boundary whitespace, if you do not request preservation
  - Replaces all carriage return and line feed pairs (U+000D and U+000A), or carriage returns (U+000D), within the document with line feeds (U+000A)
  - Performs attribute-value normalization, as specified in the XML 1.0 specification

    This process causes line feed (U+000A) characters in attributes to be replaced with space characters (U+0020).

Additional changes occur when you retrieve the data from an XML column. Those changes are:

- If the data has an XML declaration before it is sent to the database server, the XML declaration is not preserved.

  With implicit serialization, for Db2 ODBC and embedded SQL applications, the Db2 database server adds an XML declaration, with the appropriate encoding specification, to the data. For Java and .NET applications, the Db2 database server does not add an XML declaration, but if you retrieve the data into a `DB2Xml` object and use certain methods to retrieve the data from that object, the IBM Data Server Driver for JDBC and SQLJ adds an XML declaration.

  If you execute the XMLSERIALIZE function, the Db2 database server adds an XML declaration with an encoding specification for UTF-8 encoding, if you specify the INCLUDING XMLDECLARATION option.
- Within the content of a document or in attribute values, certain characters are replaced with entity references for their predefined XML entities. Those characters and their predefined entities are:

Character	Unicode value	Entity reference
AMPERSAND	U+0026	&
LESS-THAN SIGN	U+003C	<
GREATER-THAN SIGN	U+003E	>

- Within attribute values or text values, certain characters are replaced with their character references for their numeric representations. Those characters and their character references are:

Character	Unicode value	Character reference
CHARACTER TABULATION	U+0009	
LINE FEED	U+000A
CARRIAGE RETURN	U+000D
NEXT LINE	U+0085	…
LINE SEPARATOR	U+2028	

- Within attribute values, the QUOTATION MARK (U+0022) character is replaced with its predefined XML entity reference ".
- If the input document has a DTD declaration, the declaration is not preserved, and no markup based on the DTD is generated.
- If the input document contains CDATA sections, those sections are not preserved in the output.

Related concepts

XML serialization
XML serialization is the process of converting XML data from its internal representation in a Db2 table to the textual XML format that it has in an application.

XML schema validation
XML schema validation is the process of determining whether the structure, content, and data types of an XML document are valid according to an XML schema.

Transforming an XML document with XSLTRANSFORM

The standard way to transform XML data into other formats is by Extensible Stylesheet Language Transformations (XSLT). You can use the XSLTRANSFORM function to convert XML documents into HTML, plain text, or different XML schemas.

XSLT uses style sheets to convert XML into other data formats. You can convert part or all of an XML document and select or rearrange the data with the XPath query language and the built-in functions of XSLT. XSLT is commonly used to convert XML to HTML, but can also be used to transform XML documents that comply with one XML schema into documents that comply with another schema. XSLT can also be used to convert XML data into formats such as comma-delimited text or formatting languages.

XSLT style sheets are written in Extensible Stylesheet Language (XSL), an XML schema. For information on how to write XSL see the W3C recommendation for XSL Transformations.

The XSLTRANSFORM function accepts an XML document from an expression, an XSL style sheet, and parameter values to the XSL style sheet as input. The XML document is transformed with the instructions in the XSL style sheet.

The XSLTRANSFORM user-defined function is created during installation or migration. The XSLTRANSFORM function requires Java. Setup is similar to the WLM environment for XML schema repository Java stored procedures.

Related tasks

Additional steps for enabling the function for XSLTRANSFORM routines support (Db2 Installation and Migration)

Defining the WLM environment and JCL startup procedure for the Java language XML schema repository stored procedure
The XML schema validation stored procedure, XSR_COMPLETE, which is written in Java, can share a WLM environment with other Java routines. You need a JCL procedure that is tailored for starting that WLM environment.

Related reference

XSLTRANSFORM (Db2 SQL)

Related information

XSL Transformations (XSLT)

Chapter 3. XML data indexing

An XML index can be used to improve the efficiency of queries on XML documents that are stored in an XML column.

In contrast to traditional relational indexes, where index keys are composed of one or more table columns that you specify, an XML index uses a particular XML pattern expression to index paths and values in XML documents stored within a single column. The data type of that column must be XML.

Instead of providing access to the beginning of a document, index entries in an XML index provide access to nodes within the document by creating index keys based on XML pattern expressions. Because multiple parts of a XML document can satisfy an XML pattern, Db2 might generate multiple index keys when it inserts values for a single document into the index.

You create an XML index using the CREATE INDEX statement, and drop an XML index using the DROP INDEX statement. The GENERATE KEY USING XMLPATTERN clause you include with the CREATE INDEX statement specifies what you want to index.

Some of the keywords used with the CREATE INDEX statement for indexes on non-XML columns do not apply to indexes over XML data.

Example: You want to create an XML index in the INFO column of the sample CUSTOMER table. The following document shows the format of documents in the INFO column.

```
<customerinfo xmlns="http://posample.org" Cid="1000">
<name>Kathy Smith</name>
<addr country="Canada">
<street>5 Rosewood</street>
<city>Toronto</city>
<prov-state>Ontario</prov-state>
<pcode-zip>M6W-1E6</pcode-zip>
</addr>
<phone type="work">416-555-1358</phone>
</customerinfo>
```

Users of the CUSTOMER table often retrieve customer information using the customer ID. You might use an index like this one to make that retrieval more efficient:

```
CREATE UNIQUE INDEX CUST_CID_XMLIDX ON CUSTOMER(INFO)       1
 GENERATE KEY USING XMLPATTERN                              2
'declare default element namespace "http://posample.org";
/customerinfo/@Cid'
 AS SQL VARCHAR(4)                                          3
```

Figure 4. Example of XML index creation

Notes to Figure 4 on page 81:

1 The XML index is defined on the INFO column of the CUSTOMER table. INFO must be of the XML data type.

2 The GENERATE KEY USING XMLPATTERN clause provides information about what you want to index. This clause is called an XML index specification. The XML index specification contains an XML pattern clause. The XML pattern clause in this example indicates that you want to index the values of the `Cid` attribute of each `customerinfo` element, and that the namespace for all unqualified elements is http://posample.org. The namespace declaration is necessary because the XML documents have a namespace of http://posample.org.

3 AS SQL VARCHAR(4) indicates that indexed values are stored as VARCHAR(4) values.

© Copyright IBM Corp. 2007, 2020

Pattern expressions

For an XML document in an XML column, Db2 indexes only the parts that satisfy an XML pattern expression.

To index on an XML pattern, you provide an index specification clause in the CREATE INDEX statement. The index specification clause begins with GENERATE KEY USING XMLPATTERN, followed by an XML pattern and a data type for the XML index.

Only one index specification clause is allowed in a CREATE INDEX statement. However, you can create multiple XML indexes on an XML column.

To identify those parts of the document that you want to index, you use an XML pattern to specify a set of nodes in the XML document. This pattern expression is similar to an XPath expression, but it differs in that only a subset of the XPath language is supported.

The following examples show various pattern expressions that index data in the Info column of the sample Customer table. Statements 1 and 2 are logically equivalent. Statement 2 uses the unabbreviated syntax. Statements 3 and 4 are logically equivalent. Statement 4 uses the unabbreviated syntax.

```
CREATE INDEX CSTPHIX1 on Customer(Info)                                    1
 GENERATE KEY USING XMLPATTERN '/customerinfo/phone/@type' AS SQL VARCHAR(12)
```

```
CREATE INDEX CSTPHIX2 on Customer(Info)                                    2
 GENERATE KEY USING XMLPATTERN '/child::customerinfo/child::phone/attribute::type'
 AS SQL VARCHAR(12)
```

```
CREATE INDEX CSTPHIX3 on Customer(Info)                                    3
 GENERATE KEY USING XMLPATTERN '//@type' AS SQL DECFLOAT
```

```
CREATE INDEX CSTPHIX4 on Customer(Info)                                    4
 GENERATE KEY USING XMLPATTERN '/descendant-or-self::node()/attribute::type'
 AS SQL DECFLOAT
```

You can tailor your pattern expressions to be more specific or more general. For example, suppose that some of the documents in the Info column of the Customer table have an `Cid` attribute on the `name` element, as well as on the `customerinfo` element. That is, the XML documents in the Info column can have either of these two paths: '/customerinfo/@Cid' and '/customerinfo/name/@Cid'. You can write an XML pattern that includes either of these paths, or both paths.

For example, if you want to index on the customer ID for a specific customer, you can create an index with the XML pattern '/customerinfo/name/@Cid'. Queries with predicates of the form '/customerinfo/name[@Cid="1000"]' can use this index.

Alternatively, you can create an XML pattern that indexes the customer ID attribute whether it appears in the `customerinfo` element or the `name` element. The pattern expression '//@Cid' does that.

XML pattern expressions can contain the fn:exists or fn:upper-case functions. If a pattern expression contains fn:exists, the data type that is associated with the pattern expression must be VARCHAR(1).

Related concepts

Data types associated with pattern expressions

Every XML pattern expression that you specify in a CREATE INDEX statement must be associated with a data type. The data type must be VARCHAR, DECFLOAT, DATE, or TIMESTAMP.

Related reference

CREATE INDEX (Db2 SQL)

Namespace declarations in XML index definitions

In the XMLPATTERN clause of the CREATE INDEX statement, you can specify an optional namespace declaration that maps a URI to a namespace prefix. Then you can use the namespace prefix when you refer to element and attribute names in the XML pattern expression.

Example: Suppose that you want to create an index for documents that look like this:

```
<customerinfo xmlns="http://posample.org" Cid="1000">
<name>Kathy Smith</name>
<addr country="Canada">
<street>5 Rosewood</street>
<city>Toronto</city>
<prov-state>Ontario</prov-state>
<pcode-zip>M6W-1E6</pcode-zip>
</addr>
<phone type="work">416-555-1358</phone>
</customerinfo>
```

In a CREATE INDEX statement, use a namespace declaration to map the namespace URI http://posample.org to the character `m`, and qualify all elements with that namespace prefix:

```
CREATE INDEX CUST_PHONE_XMLIDX on CUSTOMER(INFO)
 GENERATE KEY USING XMLPATTERN
 'declare namespace m="http://posample.org";
 /m:customerinfo/m:phone/@type' AS SQL VARCHAR(12)
```

You can include multiple namespace declarations in the same XMLPATTERN expression, but the namespace prefix must be unique within the list of namespace declarations. In addition, you can declare a default namespace for elements that do not have a prefix. If you do not explicitly specify a namespace or namespace prefix for an element, Db2 uses the default namespace. You can declare only one default namespace. If you do not specify a default namespace, the namespace is *no namespace*.

Default namespace declarations do not apply to attributes.

Example: Write a CREATE INDEX statement to use a default namespace of http://posample.org for all elements:

```
CREATE INDEX CUST_PHONE_XMLIDX on CUSTOMER(INFO)
 GENERATE KEY USING XMLPATTERN
 'declare default element namespace "http://posample.org";
 /customerinfo/phone/@type' AS SQL VARCHAR(12)
```

The namespace for the `@type` attribute is *no namespace*. If you want to qualify `@type`, you need to do that explicitly, as shown below.

Example: Suppose that column INFO in table CUSTOMER contains documents of this form:

```
<customerinfo xmlns:n="http://posample.org"
xmlns="http://posample.org" Cid="1010">
<name>Christine Haas</name>
<addr country="United States">
<street>1000 Oakwood</street>
<city>Toledo</city>
<prov-state>Ohio</prov-state>
<pcode-zip>43537</pcode-zip>
</addr>
<phone n:type="work">567-555-1469</phone>
</customerinfo>
```

You need an index that looks like this to match those documents:

```
CREATE INDEX CUST_PHONE_XMLIDX on CUSTOMER(INFO)
 GENERATE KEY USING XMLPATTERN
 'declare default element namespace "http://posample.org";
  declare namespace m="http://posample.org";
 /customerinfo/phone/@m:type' AS SQL VARCHAR(12)
```

The namespace prefix that you use to create an the index does not need to match the namespace prefix that you use in XML documents. However, the fully-expanded QNames must match.

Related concepts

XML namespaces and qualified names in XQuery
XQuery uses XML namespaces to prevent naming collisions. An *XML namespace* is a collection of names that is identified by a namespace URI. Namespaces provide a way of qualifying names that are used for elements, attributes, data types, and functions in XQuery.

Related reference

CREATE INDEX (Db2 SQL)

Data types associated with pattern expressions

Every XML pattern expression that you specify in a CREATE INDEX statement must be associated with a data type. The data type must be VARCHAR, DECFLOAT, DATE, or TIMESTAMP.

If a pattern expression contains fn:exists, the data type that is associated with the pattern expression must be VARCHAR(1).

You can interpret the result of pattern expression as multiple data types. For example, the value 123 has a character representation, but it can also be interpreted as the number 123. You can create different indexes on the same pattern expression with different data types, so that the data can be indexed, regardless of its data type.

Example: Create indexes for the character or numeric representation of the Cid attribute in XML documents in the Info column of the sample Customer table:

```
CREATE INDEX CUST_XMLIDX_CHAR on Customer(Info)
 GENERATE KEY USING XMLPATTERN '/customerinfo/@Cid' AS SQL VARCHAR(4)
```

```
CREATE INDEX CUST_XMLIDX_NUM on Customer(Info)
 GENERATE KEY USING XMLPATTERN '/customerinfo/@Cid' AS SQL DECFLOAT
```

Example: Create an index for the date representation of the shipDate element in XML documents in the PORDER column of the PURCHASEORDER table.

```
CREATE INDEX PO_XMLIDX1 ON PURCHASEORDER (PORDER)
 GENERATE KEY USING XMLPATTERN '//items/shipDate'
 AS SQL DATE
```

Related concepts

Pattern expressions
For an XML document in an XML column, Db2 indexes only the parts that satisfy an XML pattern expression.

Related reference

CREATE INDEX (Db2 SQL)

XML schemas and XML indexes

If you validate your XML documents against an XML schema, ensure that the data type specifications in the XML schema match the data types that you use for your indexes.

Example: Suppose that an XML schema for documents in the Description column of the sample Product table looks like this:

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://posample.org"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">
  <xs:element name="product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="description" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name">
```

```
               <xs:simpleType>
                <xs:restriction base="xs:string">
                 <xs:maxLength value="20"/>
                </xs:restriction>
               </xs:simpleType>
              <xs:element name="details" type="xs:string" minOccurs="0" />
              <xs:element name="price" type="xs:decimal" minOccurs="0" />
              <xs:element name="weight" type="xs:string" minOccurs="0" />

              <xs:element name="batteries" nillable="true" minOccurs="0"
               maxOccurs="unbounded">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="included" type="xs:string" />
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>

            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="pid" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

After looking at the queries you need to execute, you determine that you need indexes on `name` and `price`. The XML schema provides guidance on what data type to pick for the index: the price element is decimal, so you need to choose the DECFLOAT data type for the index on `price`. `name` has a string data type, with a maximum length of 20 Unicode characters. You should choose the VARCHAR(80) type for the index on `name`. The maximum length of 80 bytes ensures that the index can accommodate the largest possible `name` value, which is a 20-character name in which each character has the maximum length of four bytes. Your indexes might look like this:

```
CREATE INDEX priceindex on Product(Description)
 GENERATE KEY USING XMLPATTERN '/product/description/price' AS DECFLOAT
CREATE INDEX colorindex on company(productdocs)
 GENERATE KEY USING XMLPATTERN '/product/description/name' AS SQL VARCHAR(80)
```

Related concepts

Pattern expressions

For an XML document in an XML column, Db2 indexes only the parts that satisfy an XML pattern expression.

Related reference

CREATE INDEX (Db2 SQL)

The UNIQUE keyword in an XML index definition

The UNIQUE keyword in XML index definitions has a slightly different meaning than it does for relational index definitions.

For relational indexes, the UNIQUE keyword in the CREATE INDEX statement enforces uniqueness across all rows in a table. For indexes over XML data, the UNIQUE keyword enforces uniqueness across all documents in an XML column.

For an XML index, Db2 enforces uniqueness for:

- The data type of the index
- The XML path to a node
- The value of the node after the XML value has been cast to the SQL data type that is specified for the index

Because rounding can occur during conversion of an index key value to the specified data type for the index, multiple values that appear to be unique in the XML document might result in duplicate key errors.

Related reference
CREATE INDEX (Db2 SQL)

Access methods with XML indexes

Several data access methods use XML indexes.

PSPI

When you create a table with XML columns, Db2 implicitly creates a *document ID index* (DOCID index) on the base table and a *node ID index* (NODEID index) on each associated XML table. The document ID index associates base table rows with rows to which XML indexes point. The data for a document in an XML table is stored in multiple records. A node ID index links the records for an XML document.

When you explicitly create an index on an XML column, the *XML index* contains composite key values that map XML values to DOCID and NODEID pairs. The XML index indexes the nodes in an XML document that match an XPath expression in the index definition. Db2 compares an XPath expression in a predicate to the XPath expression in an XML index to determine index key entries that contain matched key values. Db2 uses the DOCIDs from the DOCID and NODEID pairs of the identified index key entries to locate the corresponding base table rows efficiently.

The following data access methods are used for predicates that have eligible XML indexes.

Access method name	ACCESSTYPE value in PLAN_TABLE	Purpose
DOCID list access	DX	Retrieval of base table rows that correspond to XML table rows. Db2 searches an XML index, retrieves all the qualified DOCIDs, and creates a DOCID list. Db2 uses the DOCID index to convert the DOCID list to a RID list that it uses to fetch base table rows.
DOCID ANDing	DI	Retrieval of rows for two predicates that include XPath expressions, when the predicates are connected by AND. Db2 creates a DOCID list for each predicate and forms the intersection of them.
DOCID ORing	DU	Retrieval of rows for two predicates that include XPath expressions, when the predicates are connected by OR. Db2 creates a DOCID list for each predicate and forms the union of them.

A matching predicate is not always an exact match with the XPath expression in an XML index. The following information describes some of the most common types of matching and restrictions on matching.

Truly exact match
An exact match, meaning that both XPath expressions are identical. This method is used only for the XML index with the SQL data type VARCHAR. For example: XPath expression in `XMLEXISTS: /a/b/c`, and XPath expression in the XML index: /a/b/c.

Exact match but the ending part of the XPath expression in XMLEXISTS is in a predicate
Used only when the XPath predicate is a general comparison with operator =, <, <=, >, or >=. The data type of the operands in the predicate must match to the index data type. For example, XPath expression in `XMLEXISTS: /a[b/@c > 123]`, and XPath expression in the XML index: /a/b/@c.

Partial exact match with residual steps
Used to evaluate the XPath expression which has more steps than the first two methods. These extra steps in the XPath expression of XMLEXISTS are called 'residual steps'. For example: XPath

expression in XMLEXISTS: /a/b[c > “xyz”]//d[e=8], and XPath expression in the XML index: /a/b/c.

Partial match for index filtering
The methods above have segments in the XPath expressions of XMLEXISTS that match “well” with the XPath expression of an index. This method handles the cases where the XPath expression in XMLEXISTS does not match so well with the XPath expression of an index. For example: XPath expression in XMLEXISTS: /a[b/c = 5]/d, and XPath expression in the XML index: //c.

Partial exact match with ANDing and ORing on DOCID lists
The XPath expression might be decomposed into multiple XPath segments which are ANDed or ORed together to produce a super set of the final result. The methods above apply to each of the decomposed XPath segments to determine whether the XML index can be used to evaluate the XPath segment. For example: XPath expression in XMLEXISTS: /a/b[c = “xyz” and d > “abc”], and XPath expressions in the XML indexes: /a/b/c, and /a/b/d.

Partial match for filtering combined with ANDing and ORing on DOCID lists
Partial match for filtering can be combined with partial exact match with ANDing and ORing on DOCID lists. For example: XPath expression in XMLEXISTS: /a/b[@c = 5 or d > “a”]/e, and XPath expressions in the XML indexes: //@c, and /a/b/d.

PSPI

Related concepts
Storage structure for XML data (Introduction to Db2 for z/OS)
XMLEXISTS predicate for querying XML data
The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.
Multiple index access (ACCESSTYPE='M', 'MX', 'MI', 'MU', 'DX', 'DI', or 'DU') (Db2 Performance)

Related reference
XMLEXISTS predicate (Db2 SQL)
PLAN_TABLE (Db2 Performance)

Example of DOCID ANDing access (ACCESSTYPE='DI')

Two XMLEXISTS predicates that are connected with AND might be eligible for DOCID ANDing access.

PSPI

Both predicates must have an eligible XML index.

The following query retrieves all XML documents from the INFO column of the CUSTOMER table for a customer whose zip code is 95141 and whose street name, when converted to uppercase, is BAILEY AVE.

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x/customerinfo/address[@zip=”95141”
 and fn:upper-case(street)="BAILEY AVE"]'
 PASSING CUSTOMER.INFO as "x")
```

The following index matches the first predicate.

```
CREATE INDEX CUST_ZIP_STR ON CUSTOMER(INFO)
    GENERATE KEY USING XMLPATTERN '/customerinfo/address/@zip'
  AS VARCHAR(10)
```

The following index matches the second predicate.

```
CREATE INDEX CUST_STREET_UPPER ON CUSTOMER(INFO)
 GENERATE KEY USING XMLPATTERN '/customerinfo/address/street/fn:upper-case(.)'
 AS VARCHAR(50)
```

Db2 uses DOCID list access (DX) to get the DOCID lists for the individual predicates, and then uses DOCID ANDing access (DI) to get the intersection of DOCID lists after an index scan of CUST_ZIP_STR and CUST_STREET_UPPER.

An excerpt from the The PLAN_TABLE output for the query looks like this:

Table 19. Example of columns of PLAN_TABLE for DOCID ANDing access

PLAN NO	ACCESS TYPE	MATCH COLS	ACCESS NAME	MIXOP SEQ
1	M	0		0
1	DX	1	CUST_ZIP_STR	1
1	DX	1	CUST_STREET_UPPER	2
1	DI	0		3

PSPI

Related concepts

Multiple index access (ACCESSTYPE='M', 'MX', 'MI', 'MU', 'DX', 'DI', or 'DU') (Db2 Performance)

Related reference

PLAN_TABLE (Db2 Performance)

XMLEXISTS predicate (Db2 SQL)

Example of DOCID ORing access (ACCESSTYPE='DU')

Two XMLEXISTS predicates that are connected with OR might be eligible for DOCID ORing access.

PSPI

Both predicates must have an eligible XML index.

The following query retrieves all XML documents from the INFO column of the CUSTOMER table for a customer whose zip code is 95141 or whose street name, when converted to uppercase, is BAILEY AVE.

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x/customerinfo/address[@zip=”95141”
 or fn:upper-case(street) = "BAILEY AVE"]'
 PASSING CUSTOMER.INFO as "x")
```

The following index matches the first predicate.

```
CREATE INDEX CUST_ZIP_STR ON CUSTOMER(INFO)
    GENERATE KEY USING XMLPATTERN '/customerinfo/address/@zip'
  AS VARCHAR(10)
```

The following index matches the second predicate.

```
CREATE INDEX CUST_STREET_UPPER ON CUSTOMER(INFO)
 GENERATE KEY USING XMLPATTERN '/customerinfo/address/street/fn:upper-case(.)'
 AS VARCHAR(50)
```

Db2 uses DOCID list access (DX) to get the DOCID lists for the individual predicates, and then uses DOCID ORing access (DI) to get the union of DOCID lists after an index scan of CUST_ZIP_STR and CUST_STREET_UPPER.

An excerpt from the The PLAN_TABLE output for the query looks like this:

Table 20. Example of columns of PLAN_TABLE for DOCID ORing access

PLAN NO	ACCESS TYPE	MATCH COLS	ACCESS NAME	MIXOP SEQ
1	M	0		0
1	DX	1	CUST_ZIP_STR	1

Table 20. Example of columns of PLAN_TABLE for DOCID ORing access (continued)

PLAN NO	ACCESS TYPE	MATCH COLS	ACCESS NAME	MIXOP SEQ
1	DX	1	CUST_STREET_UPPER	2
1	DU	0		3

PSPI

Related concepts

Multiple index access (ACCESSTYPE='M', 'MX', 'MI', 'MU', 'DX', 'DI', or 'DU') (Db2 Performance)

Related reference

PLAN_TABLE (Db2 Performance)

XMLEXISTS predicate (Db2 SQL)

Examples of index definitions and queries that use them

Examples demonstrate some common types of predicates that reference XML documents, and compatible indexes for those predicates.

Related concepts

XMLEXISTS predicate for querying XML data

The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.

Related reference

CREATE INDEX (Db2 SQL)

Examples of XML index usage by equal predicates

Examples demonstrate the use of XML indexes by equal predicates.

Example: The following query includes an equal predicate on a string type. It retrieves all documents from the INFO column of the CUSTOMER table for customers whose zip code is 95141.

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x/customerinfo/address[@zip="95141"]'
 PASSING CUSTOMER.INFO AS "x")
```

To be compatible with this query, the XML index needs to include the `zip` attribute node among the indexed nodes, and to store values in the index as a VARCHAR type.

The query can use this XML index:

```
CREATE INDEX CUST_ZIP_STR on CUSTOMER(INFO)
    GENERATE KEY USING XMLPATTERN '/customerinfo/address/@zip'
  AS VARCHAR(10)
```

Example: Suppose that you change the query in the previous example to look like this:

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x//address[@zip="95141"]'
 PASSING CUSTOMER.INFO AS "x")
```

Index CUST_ZIP_STR in the previous example cannot be used for this query because the XPath expression in the XMLEXISTS predicate now specifies a superset of the nodes that the index specifies. In the query, the `zip` attribute node is under an address element that is the descendant of *any* node. Index

CUST_ZIP_STR specifies only the `zip` attribute under the address element that is a child of the customerinfo element. Define an index like this for use with the query in this example:

```
CREATE INDEX CUST_ZIP_STR2 ON CUSTOMER(INFO)
    GENERATE KEY USING XMLPATTERN '//address/@zip'
  AS VARCHAR(10)
```

Example: The following query includes an equal predicate on a numeric type. It retrieves documents from the DESCRIPTION column of the PRODUCT table for items with a price equal to 9.99.

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x//address[@zip="95141"]'
 PASSING CUSTOMER.INFO AS "x")
```

To be compatible, the XML index needs to include price nodes among the indexed nodes, and to store values as the DECFLOAT type.

The query can use this XML index:

```
CREATE INDEX PRODINDEX ON PRODUCT(DESCRIPTION)
 GENERATE KEY USING XMLPATTERN '//price' AS SQL DECFLOAT
```

Example: The following query includes an equal predicate on a text node. It retrieves all documents from the Info column of the sample Customer table for which the assistant name is `Gopher Runner`.

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x/customerinfo/assistant[name="Gopher Runner"]'
 PASSING BY REF INFO AS "x")
```

To be compatible with this query, the XML index needs to include the text node within the `name` element that is under the `assistant` element, and needs to store values in the index as a VARCHAR type.

The query can use this XML index:

```
CREATE INDEX CUSTINDEX on CUSTOMER(INFO)
 GENERATE KEY USING XMLPATTERN '/customerinfo/assistant/name/text()'
 AS SQL VARCHAR(20)
```

Examples of XML index usage by predicates that test for node existence

If an XMLEXISTS predicate contains the fn:exists or fn:not function, it matches an XML index that contains the fn:exists or fn:not function.

Example: The following query retrieves all customerinfo documents from the INFO column of the CUSTOMER table for which the address node has a `zip` attribute.

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x/customerinfo/address[fn:exists(@zip)]'
passing CUSTOMER.INFO as "x")
```

The following query retrieves all customerinfo documents from the INFO column of the CUSTOMER table for which the address node does not have a `zip` attribute.

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x/customerinfo/address[fn:not(@zip)]'
passing CUSTOMER.INFO as "x")
```

Both of these queries can use the following XML index:

```
CREATE INDEX CUST_ZIP_EXISTENCE on CUSTOMER(INFO)
 GENERATE KEY USING XMLPATTERN '/customerinfo/address/fn:exists(@zip)'
 AS VARCHAR(1)
```

For queries that test for existence, VARCHAR(1) is the compatible data type for the XML index, because the index key values can be only 'T' or 'F'.

Example of XML index usage by predicates with case-insensitive comparisons

In XML, string comparisons are case-sensitive. However, some applications might require case-insensitive comparisons. You can use the fn:upper-case function to do a case-insensitive comparison.

Example: The following query retrieves all XML documents from the INFO column of the CUSTOMER table for customers whose child address has a street whose upper-case is “BAILEY AVE”. The query will return the XML document, when the street is “Bailey Ave”, or “bailey ave”, or “Bailey AVE”, and so on.

```
SELECT INFO FROM CUSTOMER
 WHERE XMLEXISTS('$x/customerinfo/address[fn:upper-case(street) =
 "BAILEY AVE"]'
 PASSING CUSTOMER.INFO AS "x")
```

The following index can be used for the predicate:

```
CREATE INDEX CUST_STREET_UPPER on CUSTOMER(INFO)
 GENERATE KEY USING XMLPATTERN '/customerinfo/address/street/fn:upper-case(.)'
 AS VARCHAR(50)
```

Example of index usage for an XMLEXISTS predicate with the fn:starts-with function

An XMLEXISTS predicate that contains the fn:starts-with function and an XML index need to meet several conditions for an index match.

Those conditions are:

- In the XMLEXISTS predicate, the second argument of the fn:starts-with function must be a string literal.
- The XML index must have the VARCHAR type.
- The pattern expression in the index must match the XPath expression in the predicate, except for the fn:starts-with function.

Example: The following query includes a predicate that checks whether the `productName` value starts with the string "Lapis".

```
SELECT PORDER FROM PURCHASEORDER
 WHERE XMLEXISTS(
 '/purchaseOrder/items/item[fn:starts-with(productName,"Lapis")]'
 PASSING PURCHASEORDER.PORDER)
```

The following index matches the predicate.

```
CREATE INDEX POSTRTSW ON PURCHASEORDER(PORDER)
 GENERATE KEYS USING XMLPATTERN
 '/purchaseOrder/items/item/productName'
 AS SQL VARCHAR(20)
```

Example of index usage for an XMLEXISTS predicate with the fn:substring function

An XMLEXISTS predicate that contains the fn:substring function and an XML index need to meet the several conditions for an index match.

Those conditions are:

- In the XMLEXISTS predicate:
  - The second argument of the fn:substring functions must be 1.
  - The operand to which the expression that contains the fn:substring function is compared is a string literal.
  - The third argument of the fn:substring function must be an integer constant that is equal to the length of the string literal.
- The pattern expression in the index must match the XPath expression in the predicate, except for the fn:substring function.

Example: The following query includes a predicate that checks whether the productName value is the string "Lapis", with the characters in uppercase or lowercase.

```
SELECT PORDER FROM PURCHASEORDER
 WHERE XMLEXISTS(
 '/purchaseOrder/items/item[fn:substring(productName, 1,5)= "LAPIS"]'
 PASSING PURCHASEORDER.PORDER)
```

The following index matches the predicate.

```
CREATE INDEX POSUBSTR ON PURCHASEORDER(PORDER)
 GENERATE KEYS USING XMLPATTERN
 '/purchaseOrder/items/item/productName'
 AS SQL VARCHAR(20)
```

Example of XML index usage by join predicates

If an XMLEXISTS predicate contains a join of two tables, the join condition compares two XPath expressions. An XML index that the predicate uses must be on the first table in the join order, and the XPath expression must match both XPath expressions in the join condition.

Example: The following query retrieves XML documents from the CUSTOMER and ORDER tables for which a customer ID in the CUSTOMER table matches the customer ID in the ORDER table. The customer IDs are compared as strings.

```
SELECT INFO FROM CUSTOMER, ORDER
 WHERE XMLEXISTS('$x/customerinfo[@Cid = $y/order/customer/@id/fn:string(.)]'
 passing CUSTOMER.INFO as "x", ORDER.ORDERINFO as "y")
```

The first table in the join is the CUSTOMER table, so the query can use the following XML index on the CUSTOMER table:

```
CREATE INDEX CUST_CID_STR ON CUSTOMER(INFO)
 GENERATE KEYS USING XMLPATTERN
 ‘/customerinfo/@Cid'
 AS SQL VARCHAR(10)
```

Because the XPath expressions in the join predicate are compared as strings, the index must store entries in the index as the VARCHAR type.

Example: The following query retrieves XML documents from the ORDER and CUSTOMER tables for which a customer ID in the ORDER table matches the customer ID in the CUSTOMER table. The customer IDs are compared as numeric values.

```
SELECT INFO FROM CUSTOMER, ORDER
 WHERE XMLEXISTS('$y/order/customer[@id = $x/customerinfo/@id/xs:decimal(.)]'
 passing CUSTOMER.INFO as "x", ORDER.ORDERINFO as "y")
```

The first table in the join is the ORDER table, so the query can use the following XML index on the ORDER table:

```
CREATE INDEX ORDER_CID_NUM ON ORDER(ORDERINFO)
 GENERATE KEYS USING XMLPATTERN
 ‘/order/customer/@id'
 AS SQL DECFLOAT
```

Because the XPath expressions in the join predicate are compared as numeric values, the index must store entries in the index as the DECFLOAT type.

Example of XML index usage by queries with XMLTABLE

If the FROM clause of a query contains an XMLTABLE function with a row-XQuery-expression, and Db2 transforms the query to contain an XMLEXISTS predicate, after transformation, the query can use an XML index.

PSPI

The original query might have an XMLTABLE function with a row-XQuery-expression, or the query might be the result of transformation of an SQL predicate to an XPath predicate in a row-XQuery-expression.

Suppose that the CUSTOMER table contains this document in the INFO column:

```
<customerinfo xmlns="http://posample.org" Cid="1010">
  <name>Elaine Decker</name>
  <addr zip="95999">
    <street>100 Oak</street>
    <city>Mountain Valley</city>
    <state>CA</state>
    <country>USA</country>
  </addr>
  <phone type="work">408-555-2310</phone>
</customerinfo>
```

Example: The following query uses the XMLTABLE function to retrieve the zip, street, city, and state elements from customerinfo documents as columns in a result table.

```
SELECT X.*
 FROM CUSTOMER,
  XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
   '$x/customerinfo/address[@zip=95999]'
   PASSING INFO as "x"
     COLUMNS
         ZIP INT PATH '@zip',
        STREET VARCHAR(50) PATH 'street',
        CITY    VARCHAR(30) PATH 'city',
   STATE VARCHAR(2) PATH 'state') AS X
```

The original query cannot use an XML index. However, the original query has a row-XQuery-expression that Db2 can transform into an XMLEXISTS predicate. After transformation, the query looks like this:

```
SELECT X.*
 FROM CUSTOMER,
  XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
   '$x/customerinfo/address[@zip=95999]'
   PASSING INFO as "x"
     COLUMNS
         ZIP INT PATH '@zip',
        STREET VARCHAR(50) PATH 'street',
        CITY    VARCHAR(30) PATH 'city',
   STATE VARCHAR(2) PATH 'state') AS X
 WHERE XMLEXISTS('$x/customerinfo/address[@zip=95999]'
 PASSING INFO AS "x")
```

The transformed query can use this index:

```
CREATE INDEX ORDER_ZIP_NUM ON CUSTOMER(INFO)
 GENERATE KEYS USING XMLPATTERN
 'declare default element namespace "http://posample.org/";
 /customerinfo/address/@zip'
 AS SQL DECFLOAT
```

The XML index needs to be defined on the INFO column of the CUSTOMER table because the XMLEXISTS predicate in the transformed query uses the INFO column of the CUSTOMER table.

Example: The following query retrieves the same information as the query in the previous example, but an SQL predicate determines which rows are returned.

```
SELECT X.*
 FROM CUSTOMER,
  XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
   '$x/customerinfo/address'
   PASSING INFO AS “x”
   COLUMNS
         ZIP INT PATH '@zip',
        STREET VARCHAR(50) PATH 'street',
        CITY    VARCHAR(30) PATH 'city',
   STATE VARCHAR(2) PATH 'state') AS X
    WHERE X.ZIP = 95999
```

Db2 can transform the query so that the SQL predicate becomes an XPath predicate in the row-XQuery-expression of the XMLTABLE function. The transformed query looks like this:

```
SELECT X.*
 FROM CUSTOMER,
  XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
   '$x/customerinfo/address/[@zip=95999]'
   PASSING INFO AS “x”
   COLUMNS
         ZIP INT PATH '@zip',
        STREET VARCHAR(50) PATH 'street',
        CITY    VARCHAR(30) PATH 'city',
   STATE VARCHAR(2) PATH 'state') AS X
```

Db2 can then transform the query again so that the row-XQuery-expression becomes an XMLEXISTS predicate. After transformation, the query looks like this:

```
SELECT X.*
 FROM CUSTOMER,
  XMLTABLE (XMLNAMESPACES(DEFAULT 'http://posample.org'),
   '$x/customerinfo/address[@zip=95999]'
   PASSING INFO as "x"
     COLUMNS
         ZIP INT PATH '@zip',
        STREET VARCHAR(50) PATH 'street',
        CITY    VARCHAR(30) PATH 'city',
   STATE VARCHAR(2) PATH 'state') AS X
 WHERE XMLEXISTS('$x/customerinfo/address[@zip=95999]'
 PASSING INFO AS "x")
```

The transformed query can use this index:

```
CREATE INDEX ORDER_ZIP_NUM ON CUSTOMER(INFO)
 GENERATE KEYS USING XMLPATTERN
 'declare default element namespace "http://posample.org/";
 /customerinfo/address/@zip'
 AS SQL DECFLOAT
```

PSPI

Related concepts

Transformation of SQL predicates to XML predicates (Db2 Performance)

Chapter 4. XML support in Db2 utilities

You can use IBM Db2 for z/OS utilities on XML objects. The utilities handle XML objects similar to the way that they handle LOB objects. For some utilities, you need to specify certain XML keywords.

CHECK DATA

In addition to checking LOB relationships, the CHECK DATA utility also checks XML relationships.

CHECK DATA can check the consistency between a base table space and the corresponding XML table spaces.

In addition, if you specify the INCLUDE XML TABLESPACES option, CHECK DATA can check the structural integrity of XML documents. CHECK DATA can verify the following items for XML objects:

- All rows in an XML column exist in the XML table space.
- All documents in the XML table space are structurally valid.
- Each index entry in the node ID index has a corresponding XML document.
- Each XML document in the XML table space has corresponding entries in the node ID index.
- Each entry in the document ID column in the base table space has a corresponding entry in the node ID index over the XML table space.
- Each entry in the node ID index contains a corresponding value in the document ID column.
- Each value in the document ID column has a corresponding entry in the document ID index.
- Each entry in the document ID index has a corresponding value in the document ID column.
- If an XML column has an XML type modifier, all XML documents in the column are valid with respect to at least one XML schema that is associated with the XML type modifier.

If the base table space is not consistent with any related XML table spaces, or a problem is found during any of the previously listed checks, CHECK DATA reports the error.

For XML checking, the default behavior of CHECK DATA is to check only the consistency between each XML column and its node ID index. However, you can modify the scope of checking by specifying combinations of the CHECK DATA SCOPE keyword and the INCLUDE XML TABLESPACES keyword. The following table lists keyword combinations and the types of checks that are performed.

Table 21. CHECK DATA SCOPE and INCLUDE XML TABLESPACES keywords that control the scope of XML checking

Scope of XML checking	SCOPE keyword	INCLUDE XML TABLESPACES keyword
Consistency of XML base table column and node ID index only	SCOPE AUXONLY, SCOPE ALL, or SCOPE PENDING	Not specified
All XML checking	SCOPE ALL or SCOPE PENDING	INCLUDE XML TABLESPACES
All XML checking except XML schema validation	SCOPE ALL or SCOPE PENDING	INCLUDE XML TABLESPACES XMLSCHEMA
XML schema validation only	SCOPE XMLSCHEMAONLY	INCLUDE XML TABLESPACES

For example, table space DSNXDX1A.DSNXSX1D contains a table named XMLTBL with XML column XMLCOL, which has an XML type modifier. If you specify the following statement, CHECK DATA checks LOB relationships, the base table space, XML relationships, and the structural integrity of XML documents for column XMLCOL, and does XML schema validation on the documents for column XMLCOL:

© Copyright IBM Corp. 2007, 2020

```
CHECK DATA TABLESPACE DSNXDX1A.DSNXSX1D
 INCLUDE XML TABLESPACES(TABLE XMLTBL XMLCOLUMN XMLCOL)
```

If you specify the following statement, CHECK DATA checks LOB relationships, the base table space, XML relationships, and the structural integrity of XML documents for column XMLCOL, but does not do XML schema validation on the documents for column XMLCOL:

```
CHECK DATA TABLESPACE DSNXDX1A.DSNXSX1D
 INCLUDE XML TABLESPACES(TABLE XMLTBL XMLCOLUMN XMLCOL) XMLSCHEMA
```

You can also specify the action that Db2 performs when it finds an error in one of these columns by specifying one of the following appropriate keywords:

Table 22. CHECK DATA error keywords

Column in error	Action that CHECK DATA takes	Keyword
XML column	Report the error only	XMLERROR REPORT
	Report the error, set the column in error to an invalid status, and delete the invalid documents in the XML table space	XMLERROR INVALIDATE
LOB column	Report the error only	LOBERROR REPORT
	Report the error and set the column in error to an invalid status	LOBERROR INVALIDATE
XML or LOB column	Report the error only	AUXERROR REPORT
	Report the error and set the column in error to an invalid status	AUXERROR INVALIDATE

For example, the following statement specifies that CHECK DATA is to check XML and LOB relationships. Db2 is to report any LOB errors and XML errors, and set any XML columns in error to an invalid status.

```
CHECK DATA TABLESPACE DSNXDX1A.DSNXSX1D
                  SCOPE AUXONLY
      LOBERROR REPORT
      XMLERROR INVALIDATE
```

CHECK INDEX

You can use the CHECK INDEX utility to check XML indexes, document ID indexes, and node ID indexes. You do not need to specify any additional keywords.

COPY

You can use the COPY utility to copy XML objects. You do not need to specify any additional keywords. When you specify that Db2 is to copy a table space with XML columns, Db2 does not automatically copy any related XML table spaces or indexes. You must explicitly specify the XML objects that you want to copy.

For example, the following statement specifies that Db2 is to copy base table space DB1.BASETS1 and the XML table space DB1.XBAS0001.

```
//COPYX    EXEC DSNUPROC,SYSTEM=DSN
//SYSIN    DD *
          TEMPLATE A DSN(&DB..&SP..COPY1) UNIT CART STACK YES
COPY      TABLESPACE DB1.BASETS1 COPYDDN(A)
          TABLESPACE DB1.XBAS0001 COPYDDN(A)
```

COPYTOCOPY

You can use the COPYTOCOPY utility to copy existing copies of the XML objects. You do not need to specify any additional keywords.

EXEC SQL

You cannot declare a cursor that includes XML data. Thus, you cannot use the Db2 UDB family cross-loader function to transfer data in XML columns. However, you can declare a cursor on a table with XML columns if the cursor does not include any XML columns.

For example, suppose that you create the following table with an XML column:

```
CREATE TABLE ORDERS
  (ORDERNO INTEGER,
   PURCHASE_ORDER XML);
```

You cannot declare the following cursor, because it includes XML data in the PURCHASE_ORDER column:

```
EXEC SQL
  DECLARE C1 CURSOR FOR SELECT * FROM ORDERS
ENDEXEC
```

However, you can declare a cursor that includes non-XML columns, as in the following example:

```
EXEC SQL
  DECLARE C2 CURSOR FOR SELECT ORDERNO FROM ORDERS
ENDEXEC
```

LISTDEF

When you create object lists with the LISTDEF utility, you can specify whether you want related XML objects to be included or excluded. Use the following keywords to indicate the objects to include or exclude:

ALL
Base, LOB, and XML objects (This keyword is the default.)

BASE
Base objects only

LOB
LOB objects only.

XML
XML objects only.

For example, the LISTDEF statements in the following table generate the indicated lists.

Table 23. Example LISTDEF statements

LISTDEF statement	Objects that are included in the list
`LISTDEF LISTALL INCLUDE TABLESPACES DATABASE ACCOUNTS` `INCLUDE INDEXSPACES DATABASE ACCOUNTS`	• All tables spaces in the ACCOUNTS database, including XML and LOB table spaces • All index spaces in the ACCOUNTS database
`LISTDEF LISTXML INCLUDE TABLESPACES DATABASE ACCOUNTS XML` `INCLUDE INDEXSPACES DATABASE ACCOUNTS XML`	• All XML table spaces in the ACCOUNTS database • All XML index spaces in the ACCOUNTS database

`LISTDEF LIST INCLUDE TABLESPACES DATABASE ACCOUNTS ALL`
`INCLUDE INDEXSPACES DATABASE ACCOUNTS ALL`
`EXCLUDE INDEXSPACES DATABASE ACCOUNTS XML`
	• All tables spaces in the ACCOUNTS database, including XML and LOB table spaces
	• All index spaces the ACCOUNTS database except for XML index spaces

LOAD

You can use the LOAD utility to load XML data.

The input data can be in the textual XML format or the binary XML format (Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format). Binary XML input data must be in the non-delimited format.

If you load data into an XML column that has an XML type modifier, the LOAD utility validates the input data according to the XML schema that is specified in the XML type modifier. If LOAD detects an XML schema violation for a row, it deletes the row and issues an error message.

The steps for loading XML data are similar to the steps for loading other types of data, except that you need to also perform the following actions:

- In the input data set:
  - If the data set is in delimited format, ensure that the XML input fields follow the standard LOAD utility delimited format.
  - If the data set is not in delimited format, specify the XML input fields similar to the way that you specify VARCHAR input. Specify the length of the field in a 2-byte binary field that precedes the data.
- In the LOAD statement:
  - Specify the keyword XML for all input fields of type XML.
  - If you want the whitespace to be preserved in the XML data, also specify the keywords PRESERVE WHITESPACE. By default, LOAD strips the whitespace.

    When data in the binary XML format is loaded into a table, and PRESERVE WHITESPACE is not specified, Db2 strips whitespace only when the input data contains whitespace tags.

For example, the following LOAD statement specifies that Db2 is to load data from the MYSYSREC data set into the PRODUCTS table:

```
LOAD DATA INDDN(MYSYSREC)
 INTO TABLE PRODUCTS
  (CATEGORY       POSITION  (1) CHAR(8),
   PURCHASE_ORDER POSITION  (10) XML PRESERVE WHITESPACE)
```

Assume that the MYSYSREC data set logically contains the following data: (This example is a logical representation and is not intended to show how the data actually looks in the input data set.)

```
Shovel 339<product pid="100-100-01" xmlns="http://podemo.org">
            <description>
                 <name>Snow Shovel, Basic 22"</name>
                 <details>Basic Snow Shovel, 22" wide, straight handle with D-Grip<details>
                 <price>9.99</price>
                 <weight>1 kg<weight>
            </description>
          <product>

Shovel  358<product pid="100-101-01" xmlns="http://podemo.org">
            <description>
                 <name>Snow Shovel, Deluxe 24"</name>
                 <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with
                 D-Grip<details>
                 <price>19.99</price>
                 <weight>2 kg<weight>
            </description>
          <product>
```

After loading this data, the PRODUCTS table contains the following information:

Table 24. PRODUCTS table

DB2_GENERATED_document ID_FOR_XML[1]	CATEGORY	PURCHASE_ORDER
1	Shovel	<product pid="100-100-01" xmlns="http://podemo.org"> <description> <name>Snow Shovel, Basic 22"</name> <details>Basic Snow Shovel, 22" wide, straight handle with D-Grip</details> <price>9.99</price> <weight>1 kg<weight> </description> <product>
2	Shovel	<product pid="100-101-01" xmlns="http://podemo.org"> <description> <name>Snow Shovel, Deluxe 24"</name> <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with D-Grip</details> <price>19.99</price> <weight>2 kg<weight> </description> <product>

Note:

1. Db2 automatically generates the document ID column for each row that is loaded into a table with at least one XML column. The document ID column is partially hidden. It is not included in the result set of a SELECT * statement. However, you can query this column by name and view information about this column and its index in the catalog. Several utilities report information on this column in their output.

Loading XML data with the LOAD utility has the following restrictions:

- You cannot specify that XML input fields be loaded into non-XML columns, such as CHAR or VARCHAR columns.
- Db2 does not perform any specified compression until the next time that you run the REORG utility on this data.
- Db2 ignores any specified FREEPAGE and PCTFREE values until the next time that you run the REORG utility on this data.
- If you specify PREFORMAT, Db2 preformats the base table space, but not the XML table space.
- You cannot directly load the document ID column of the base table space.
- You cannot specify a default value for an XML column.
- You cannot load XML values that are greater than 32 KB. To load such values, use file reference variables in the LOAD utility, or use applications with SQL XML AS file reference variables.

QUIESCE

When you specify QUIESCE TABLESPACESET, the table space set includes related XML objects. You do not have to specify any additional keywords in the QUIESCE statement.

REBUILD INDEX

You can use the REBUILD INDEX utility to rebuild XML indexes, document ID indexes, and node ID indexes. You do not need to specify any additional keywords in the REBUILD INDEX statement. REBUILD INDEX with SHRLEVEL CHANGE is not valid for XML indexes.

RECOVER

You can use the RECOVER utility to recover XML objects. You do not need to specify any additional keywords in the RECOVER statement.

When you recover an XML index or table space to a point in time, you should recover all related objects to the same point in time. Related objects include XML objects, LOB objects, and referentially related objects. If you do not recover all related objects to the same point in time, one or more objects might be placed in a restrictive state.

REORG INDEX

You can use the REORG utility to reorganize XML indexes. When you specify that you want XML indexes to be reorganized, you must also specify the WORKDDN keyword and provide the specified temporary work file. The default is SYSUT1.

REORG TABLESPACE

You can use the REORG TABLESPACE utility to reorganize XML objects. You do not need to specify any additional keywords in the REORG statement.

When you specify the name of the base table space in the REORG statement, Db2 reorganizes only that table space and not any related XML objects. If you want Db2 to reorganize the XML objects, you must specify those object names.

When you run REORG on an XML table space that supports XML versions, REORG discards rows for versions of an XML document that are no longer needed.

For XML table spaces and base table spaces with XML columns, you cannot specify the following options in the REORG statement:

- DISCARD
- REBALANCE
- UNLOAD EXTERNAL

REPAIR

You can use the REPAIR utility on XML objects.

You can use the REPAIR utility to:

- Set the status of an XML column to invalid.
- Delete a corrupted XML document and its node ID index entries.

The most common use for the REPAIR utility for XML objects is to take corrective action after you run CHECK DATA with SHRLEVEL CHANGE on a table space with XML columns. CHECK DATA with SHRLEVEL CHANGE operates on shadow data sets, so it does not modify XML columns or XML table spaces. Instead, CHECK DATA generates REPAIR statements that you can run to delete invalid XML documents and to mark the corresponding XML columns as invalid.

REPORT

When you specify REPORT TABLESPACESET, the output report includes XML objects in the list of members in the table space set. The following sample output shows a table space set for a table that contains a LOB column and an XML column:

```
TABLESPACE SET REPORT:

TABLESPACE         : DBDKCX.TS0001
  TABLE            : SYSADM.DKCTEST
  INDEXSPACE       : DBDKCX.IRdocument IDD
  INDEX            : SYSADM.I_document IDDKCTEST


XML TABLESPACE SET REPORT:

TABLESPACE                 : DBDKCX.TS0001

  BASE TABLE               : SYSADM.DKCTEST
```

```
  COLUMN                   : COL2
    XML TABLESPACE         : DBDKCX.XDKC0000
      XML TABLE            : SYSADM.XDKCTEST
      XML NODEID INDEXSPACE: DBDKCX.IRNODEID
      XML NODEID INDEX     : SYSADM.I_NODEIDXDKCTEST
      XML INDEXSPACE       : DBDKCX.VALUES
      XML INDEX            : SYSADM.VALUES
```

RUNSTATS

You can use the RUNSTATS utility to gather statistics for XML objects.

RUNSTATS ignores the following keywords for XML tables and XML indexes:

- COLGROUP
- FREQVAL MOST|LEAST|BOTH
- HISTOGRAM

RUNSTATS INDEX ignores the following keywords for XML indexes:

- KEYCARD
- FREQVAL MOST|LEAST|BOTH
- HISTOGRAM

XML indexes are related to XML tables, and not to the associated base tables. If you specify a base table space and an XML index in the same RUNSTATS control statement, Db2 generates an error. When you run RUNSTATS against a base table, RUNSTATS collects statistics only for indexes on the base table, including the document ID index.

UNLOAD

You can use the UNLOAD utility to unload XML data.

The output data can be in the textual XML format or the binary XML format. Data that is unloaded can be in the delimited or non-delimited format.

When data is unloaded in the binary XML format, UNLOAD does not add whitespace tags.

In the UNLOAD statement, specify the base table space. (You do not have to specify the XML table space.) Also specify the XML keyword in the field specifications for the XML columns.

For example, the following UNLOAD statement specifies that Db2 is to unload data from the XMLSAMP table into the SYSREC data set in delimited format.

```
//STEP3    EXEC DSNUPROC,UID='JUQBU105.UNLD1',
//         UTPROC='',
//         SYSTEM='SSTR'
//UTPRINT  DD SYSOUT=*
//SYSREC   DD DSN=JUQBU105.UNLD1.STEP3.TBQB0501,DISP=(MOD,DELETE,CATLG),
//         UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SYSPUNCH DD DSN=JUQBU105.UNLD1.STEP3.SYSPUNCH
//         DISP=(MOD,CATLG,CATLG)
//         UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SYSIN    DD*
  UNLOAD TABLESPACE DBQB0501.XMLSAMP
         DELIMITED CHARDEL X'22' COLDEL X'2C' DECPT X'2E'
          FROM TABLE ADMF001.BASETBL
          (CATEGORY          POSITION(*) CHAR,
           PURCHASE_ORDER    POSITION(*) XML)
         UNICODE
 /*
```

Assume that the table contains the data in Table 24 on page 99. The output, delimited data is:

```
Shovel,"<product pid=""100-100-01"" xmlns="http://podemo.org"">
            <description>
                 <name>Snow Shovel, Basic 22""</name>
                 <details>Basic Snow Shovel, 22"" wide, straight handle with D-Grip<details>
                 <price>9.99</price>
                 <weight>1 kg<weight>
            </description>
          <product>"
```

```
Shovel,"<product pid=""100-101-01"" xmlns="http://podemo.org"">
            <description>
                 <name>Snow Shovel, Deluxe 24""</name>
                 <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with
                 D-Grip<details>
                 <price>19.99</price>
                 <weight>2 kg<weight>
            </description>
         <product>"
```

For maximum portability, specify UNICODE in the UNLOAD statement and use Unicode delimiter characters. If XML columns are not being unloaded in UTF-8 CCSID 1208, the unloaded column values are prefixed with a standard XML encoding declaration that specifies the encoding that is used.

Unloading XML data with the UNLOAD utility has the following restrictions:

- You cannot specify that the XML data be converted to another data type, such as CHAR or VARCHAR.
- You cannot unload XML data from a copy.

Inline statistics and inline copies

When you request that a utility gathers statistics or make a copy inline, the following restrictions apply:

- You cannot take inline copies of XML table spaces.
- When you request inline statistics or inline copies for the base table space, Db2 does not take copies or gather statistics for any related XML table spaces. You must explicitly specify that you want statistics gathered for any XML table spaces.
- When you request that inline statistics be collected for an XML index, you cannot specify the following statistic keywords:
  - HISTOGRAM
  - KEYCARD
  - FREQVAL NUMCOLS COUNT

Stand-alone utilities

Stand-alone utilities have no specific options to support XML data. However, you can use stand-alone utilities on XML data.

The DSN1COPY utility has the following restriction on use with XML data:

- Do not use DSN1COPY to copy XML table spaces from one subsystem to another.

Related concepts

XML schema validation with an XML type modifier
You can automate XML schema validation by adding an XML type modifier to an XML column definition.

XML versions
Multiple versions of an XML document can coexist in an XML table. The existence of multiple versions of an XML document can lead to improved concurrency through lock avoidance. In addition, multiple versions can save real storage by avoiding a copy of the old values in the document into memory in some cases.

Chapter 5. XML schema management with the XML schema repository (XSR)

A Db2 for z/OS XML schema repository (XSR) is a set of Db2 tables where you can store XML schemas.

Db2 creates the XSR tables during installation or migration. After you add XML schemas to the Db2 XSR, you can use them to validate XML documents before you store them in XML columns.

When you validate an XML document against a schema, Db2 returns a binary representation of the document, which includes default values and normalized text. XML schemas contain data type information for the data that is stored in an XML value, but Db2 for z/OS does not use or store that data type information.

An XML schema consists of a set of XML schema documents. To add an XML schema to the Db2 XSR, you register XML schema documents to Db2. The XML schema documents must be in the Unicode encoding scheme.

You can register an XML schema in any of the following ways:

- Call the following Db2-supplied stored procedures from a Db2 application program:

  SYSPROC.XSR_REGISTER
  Begins registration of an XML schema. You call this stored procedure when you add the first XML schema document to an XML schema.

  SYSPROC.XSR_ADDSCHEMADOC
  Adds additional XML schema documents to an XML schema that you are in the process of registering. You can call SYSPROC.XSR_ADDSCHEMADOC only for an existing XML schema that is not yet complete.

  SYSPROC.XSR_COMPLETE
  Completes the registration of an XML schema.

  During XML schema completion, Db2 resolves references inside XML schema documents to other XML schema documents.

- Invoke the following JDBC method from a Java application program:

  com.ibm.db2.jcc.DB2Connection.registerDB2XmlSchema
  Performs the functions of SYSPROC.XSR_REGISTER, SYSPROC.XSR_ADDSCHEMADOC, and SYSPROC.XSR_COMPLETE.

- Invoke the following commands from the Command Line Processor:

  REGISTER XMLSCHEMA
  Begins registration of an XML schema. You invoke this command when you add the first XML schema document to an XML schema.

  ADD XMLSCHEMA DOCUMENT
  Adds additional XML schema documents to an XML schema that you are in the process of registering. You can invoke -ADD XMLSCHEMA DOCUMENT only for an existing XML schema that is not yet complete.

  COMPLETE XMLSCHEMA
  Completes the registration of an XML schema.

To remove an XML schema from the Db2 XSR, you can use one of the following techniques:

- Call the following Db2-supplied stored procedure SYSPROC.XSR_REMOVE from a Db2 application program.
- Invoke the JDBC method `com.ibm.db2.jcc.DB2Connection.deregisterDB2XMLObject` from a Java application program.

© Copyright IBM Corp. 2007, 2020

Related concepts

The Db2 command line processor (Db2 Commands)

Procedures for XML schema registration and removal that are supplied with Db2

Db2 provides several stored procedures that you can call in your application programs to perform XML schema registration and removal.

Those stored procedures are:

Table 25. XML schema stored procedures

Stored procedure name	Function
XSR_REGISTER	The XSR_REGISTER procedure is the first stored procedure to be called as part of the XML schema registration process.
XSR_ADDSCHEMADOC	The XSR_ADDSCHEMADOC stored procedure is used to add every XML schema other than the primary XML schema document.
XSR_COMPLETE	The XSR_COMPLETE procedure is the final stored procedure to be called as part of the XML schema registration process.
XSR_REMOVE	The XSR_REMOVE procedure is used to remove all components of an XML schema.

Example of XML schema registration and removal using stored procedures

Db2 provides the SYSPROC.XSR_REGISTER, SYSPROC.XSR_ADDSCHEMADOC, SYSPROC.XSR_COMPLETE, and SYSPROC.XSR_REMOVE stored procedures. These stored procedures let you register and remove XML schemas and their components.

To register an XML schema with a single XML schema document, you call SYSPROC.XSR_REGISTER and SYSPROC.XSR_COMPLETE.

To register an XML schema with several XML schema documents, you call SYSPROC.XSR_REGISTER for the first schema document, call SYSPROC.XSR_ADDSCHEMADOC once for each of the other schema documents, and then call SYSPROC.XSR_COMPLETE.

To remove an XML schema, you call SYSPROC.XSR_REMOVE.

To modify the contents of an XML schema for which you have already called SYSPROC.XSR_COMPLETE, you need to call SYSPROC.XSR_REMOVE and begin the registration process again.

Example: The following code performs these steps:

1. Registers an XML schema with an XML schema document.
2. Adds another XML schema document to the XML schema.
3. Completes registration of the XML schema.
4. Drops the XML schema.

```
EXEC SQL BEGIN DECLARE SECTION;
/*********************************************************************/
/* Declare variables:                                                */
/* For SYSPROC.XSR_REGISTER parameters:                              */
/* Parameter name        Value                                       */
/* rschema               Schema of the XML schema (must be 'SYSXSR')*/
/* name                  XML schema name                             */
/* schemaLocation        URI for the XML schema                      */
/* content               XML schema document to be registered        */
/* docProperties         XML schema document information for an      */
/*                        external XML schema registry               */
/* For SYSPROC.XSR_ADDSCHEMADOC parameters:                          */
/* Parameter name        Value                                       */
/* rschema               Same as for SYSPROC.XSR_REGISTER            */
```

```
/* name                  Same as for SYSPROC.XSR_REGISTER          */
/* schemaLocation        Same as for SYSPROC.XSR_REGISTER          */
/* content               Same as for SYSPROC.XSR_REGISTER          */
/* docProperties         Same as for SYSPROC.XSR_REGISTER          */
/* For SYSPROC.XSR_COMPLETE parameters:                            */
/* Parameter name        Value                                     */
/* rschema               Same as for SYSPROC.XSR_REGISTER          */
/* name                  Same as for SYSPROC.XSR_REGISTER          */
/* schemaProperties      XML schema information for an external    */
/*                       XML schema registry                       */
/* issuedForDecomp       Indicator: Must be 0                      */
/* For SYSPROC.XSR_REMOVE parameters:                              */
/* Parameter name        Value                                     */
/* rschema               Same as for SYSPROC.XSR_REGISTER          */
/* name                  Same as for SYSPROC.XSR_REGISTER          */
/*******************************************************************/
struct {
  short len;
  char text[128]; } rschema;
struct {
  short len;
  char text[128]; } name;
struct {
  short len;
  char text[1000]; } schemaLocation;
SQL TYPE IS BLOB (1M) content;
SQL TYPE IS BLOB (1M) docProperties;
SQL TYPE IS BLOB (1M) schemaProperties;
long issuedForDecomp;
EXEC SQL END DECLARE SECTION;
…
main
{
 /******************************************************************/
 /* Assign the schema for the XML schema ('SYSXSR') to rschema.    */
 /******************************************************************/
 strcpy(rschema.text,"SYSXSR");
 rschema.len=strlen(rschema.text);
 /******************************************************************/
 /* Assign the XML schema name 'ORDER' to name.                    */
 /******************************************************************/
 strcpy(name.text,"ORDER");
 name.len=strlen(name.text);
 /******************************************************************/
 /* Assign the XML schema location 'http://posample.org' to        */
 /* schemaLocation.                                                */
 /******************************************************************/
 strcpy(schemaLocation.text,"http://posample.org");
 schemaLocation.len=strlen(schemaLocation.text);
 …
 /******************************************************************/
 /* Read the first XML schema document into host variable content  */
 /* from a file.                                                   */
 /******************************************************************/
 …
 /******************************************************************/
 /* No information for this XML schema document needs to be stored */
 /* for an external registry, so set docProperties to NULL.        */
 /******************************************************************/
 EXEC SQL SET :docProperties = NULL;
 /******************************************************************/
 /* Call the SYSPROC.XSR_REGISTER to register SYSXSR.ORDER.        */
 /******************************************************************/
 EXEC SQL
  CALL SYSPROC.XSR_REGISTER
  (:rschema,:name,:schemaLocation,:content,:docProperties);
 …
 /******************************************************************/
 /* Read the second XML schema document into host variable content */
 /* from a file.                                                   */
 /******************************************************************/
 …
 /******************************************************************/
 /* Copy the contents of the second XML schema document into host  */
 /* host variable docProperties for storage in the XML schema      */
 /* repository.                                                    */
 /******************************************************************/
 EXEC SQL SET :docProperties = :content;
 /******************************************************************/
 /* Call the SYSPROC.XSR_ADDSCHEMADOC to register add the second   */
 /* document to SYSXSR.ORDER.                                      */
 /******************************************************************/
```

```
 EXEC SQL
  CALL SYSPROC.XSR_ADDSCHEMADOC
  (:rschema,:name,:schemaLocation,:content,:docProperties);
 /*********************************************************************/
 /* No information for this XML schema needs to be stored             */
 /* for an external registry, so set schemaProperties to NULL.        */
 /*********************************************************************/
 EXEC SQL SET :schemaProperties = NULL;
 /*********************************************************************/
 /* Set schemaProperties to NULL because we are not going to store    */
 /* any schema properties for SYSXSR.ORDER.                           */
 /*********************************************************************/
 EXEC SQL SET :schemaProperties=NULL;
 /*********************************************************************/
 /* Set issuedForDecomp to 0 because we are not going to use          */
 /* SYSXSR.ORDER for decomposition.                                   */
 /*********************************************************************/
 issuedForDecomp=0;
 /*********************************************************************/
 /* Call the SYSPROC.XSR_COMPLETE to complete registration of         */
 /* SYSXSR.CUSTOMER.                                                  */
 /*********************************************************************/
 EXEC SQL
  CALL SYSPROC.XSR_COMPLETE
  (:rschema,:name,:schemaProperties,:issuedForDecomp);
 /*********************************************************************/
 /* Call the SYSPROC.XSR_REMOVE to delete SYSXSR.CUSTOMER from the    */
 /* XML schema repository.                                            */
 /*********************************************************************/
 EXEC SQL
  CALL SYSPROC.XSR_REMOVE
  (:rschema,:name);

}
```

Chapter 6. Db2 application programming language support for XML

You can write applications to store XML data in Db2 database tables or retrieve XML data from tables. XML parameters for external stored procedures or user-defined functions are not supported.

You can use any of the following languages to write your applications:

- C or C++ (in embedded SQL or Db2 ODBC applications)
- COBOL
- Java (JDBC or SQLJ)
- Assembler
- PL/I

An application program can retrieve an entire document or a fragment of a document from an XML column, or store an entire document or a fragment of a document in an XML column.

When an application provides an XML value to a Db2 database server, the database server converts the data from the textual XML format to the XML hierarchical format, in Unicode UTF-8 encoding.

When an application retrieves data from XML columns, the Db2 database server converts the data from the XML hierarchical format to one of the following formats:

- The textual XML format. For the textual XML format, the database server might need to convert the output data from UTF-8 to the application encoding.
- The XML binary format. This format is an option only for JDBC, SQLJ, or ODBC applications.

When you retrieve XML data, you need to be aware of the effect of code page conversion on data loss. Data loss can occur when characters in the source code page cannot be represented in the target code page.

An application can retrieve an entire XML document or a sequence from an XML column.

When you fetch an entire XML document, you retrieve the document into an application variable.

To retrieve an XML sequence, within an SQL FETCH or single-row SELECT INTO operation, you call the XMLQUERY built-in function, passing an XQuery expression as an argument. XMLQUERY is a scalar function that returns the entire sequence in an application variable.

Related concepts

XML data encoding
The encoding of XML data can be derived from the data itself, which is known as *internally encoded* data, or from external sources, which is known as *externally encoded* data.

XML data in Java applications

In Java applications, you can store XML data in Db2 databases or retrieve XML data from Db2 databases by using JDBC or SQLJ.

Related concepts

Java support for XML schema registration and removal (Db2 Application Programming for Java)
XML data in JDBC applications (Db2 Application Programming for Java)
XML column updates in JDBC applications (Db2 Application Programming for Java)
XML data retrieval in JDBC applications (Db2 Application Programming for Java)
XML data in SQLJ applications (Db2 Application Programming for Java)
XML column updates in SQLJ applications (Db2 Application Programming for Java)
XML data retrieval in SQLJ applications (Db2 Application Programming for Java)
Invocation of routines with XML parameters in Java applications (Db2 Application Programming for Java)

© Copyright IBM Corp. 2007, 2020

XMLCAST in SQLJ applications (Db2 Application Programming for Java)
Binary XML format in Java applications (Db2 Application Programming for Java)

XML data in embedded SQL applications

Embedded SQL applications that are written in assembler language, C, C++, COBOL, or PL/I can update and retrieve data in XML columns.

In embedded SQL applications, you can:

- Store an entire XML document in an XML column using INSERT or UPDATE statements.
- Retrieve an entire XML document from an XML column using SELECT statements.
- Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function within a SELECT or FETCH statement, to retrieve the sequence into a textual XML string in the database, and then retrieve the data into an application variable.

Recommendation: Follow these guidelines when you write embedded SQL applications:

- Avoid using the XMLPARSE and XMLSERIALIZE functions.

  Let Db2 do the conversions between the external and internal XML formats implicitly.
- Use XML host variables for input and output.

  Doing so allows Db2 to process values as XML data instead of character or binary string data. If the application cannot use XML host variables, it should use binary string host variables to minimize character conversion issues.
- Avoid character conversion by using UTF-8 host variables for input and output of XML values whenever possible.

Host variable data types for XML data in embedded SQL applications

Db2 provides XML host variable types for assembler, C, C++, COBOL, and PL/I.

Those types are:

- XML AS BLOB
- XML AS CLOB
- XML AS DBCLOB
- XML AS BLOB_FILE (C, C++, or PL/I) or XML AS BLOB-FILE (COBOL)
- XML AS CLOB_FILE (C, C++, or PL/I) or XML AS CLOB-FILE (COBOL)
- XML AS DBCLOB_FILE (C, C++, or PL/I) or XML AS DBCLOB-FILE (COBOL)

The XML host variable types are compatible only with the XML column data type.

You can use BLOB, CLOB, DBCLOB, CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY host variables to update XML columns. You can convert the host variable data types to the XML type using the XMLPARSE function, or you can let the Db2 database server perform the conversion implicitly.

You can use BLOB, CLOB, DBCLOB, CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY host variables to retrieve data from XML columns. You can convert the XML data to the host variable type using the XMLSERIALIZE function, or you can let the Db2 database server perform the conversion implicitly.

The following examples show you how to declare XML host variables in each supported language. In each table, the left column contains the declaration that you code in your application program. The right column contains the declaration that Db2 generates.

Declarations of XML host variables in assembler

The following table shows assembler language declarations for some typical XML types.

Table 26. Example of assembler XML variable declarations

You declare this variable	Db2 generates this variable

`BLOB_XML SQL TYPE IS XML AS BLOB 1M`	`BLOB_XML DS 0FL4`
	`BLOB_XML_LENGTH DS FL4`
	`BLOB_XML_DATA DS CL65535`"1" on page 109
	` ORG *+(983041)`

`CLOB_XML SQL TYPE IS XML AS CLOB 40000K`	`CLOB_XML DS 0FL4`
	`CLOB_XML_LENGTH DS FL4`
	`CLOB_XML_DATA DS CL65535`"1" on page 109
	` ORG *+(40894465)`

`DBCLOB_XML SQL TYPE IS XML AS DBCLOB 4000K`	`DBCLOB_XML DS 0FL4`
	`DBCLOB_XML_LENGTH DS FL4`
	`DBCLOB_XML_DATA DS GL65534`"2" on page 109
	` ORG *+(4030466)`

`BLOB_XML_FILE SQL TYPE IS XML AS BLOB_FILE`	`BLOB_XML_FILE DS 0FL4`
	`BLOB_XML_FILE_NAME_LENGTH DS FL4`
	`BLOB_XML_FILE_DATA_LENGTH DS FL4`
	`BLOB_XML_FILE_FILE_OPTIONS DS FL4`
	`BLOB_XML_FILE_NAME DS CL255`

`CLOB_XML_FILE SQL TYPE IS XML AS CLOB_FILE`	`CLOB_XML_FILE DS 0FL4`
	`CLOB_XML_FILE_NAME_LENGTH DS FL4`
	`CLOB_XML_FILE_DATA_LENGTH DS FL4`
	`CLOB_XML_FILE_FILE_OPTIONS DS FL4`
	`CLOB_XML_FILE_NAME DS CL255`

`DBCLOB_XML_FILE SQL TYPE IS XML AS DBCLOB_FILE`	`DBCLOB_XML_FILE DS 0FL4`
	`DBCLOB_XML_FILE_NAME_LENGTH DS FL4`
	`DBCLOB_XML_FILE_DATA_LENGTH DS FL4`
	`DBCLOB_XML_FILE_FILE_OPTIONS DS FL4`
	`DBCLOB_XML_FILE_NAME DS CL255`

Notes:

1. Because assembler language allows character declarations of no more than 65535 bytes, Db2 separates the host language declarations for XML AS BLOB and XML AS CLOB host variables that are longer than 65535 bytes into two parts.
2. Because assembler language allows graphic declarations of no more than 65534 bytes, Db2 separates the host language declarations for XML AS DBCLOB host variables that are longer than 65534 bytes into two parts.

Declarations of XML host variables in C

The following table shows C and C++ language declarations that are generated by the Db2 precompiler for some typical XML types. The declarations that the Db2 coprocessor generates might be different.

Table 27. Examples of C language variable declarations

You declare this variable	Db2 generates this variable

`SQL TYPE IS XML AS BLOB (1M) blob_xml;`	`struct`
	`{ unsigned long length;`
	`  char data??(1048576??);`
	`} blob_xml;`

Table 27. Examples of C language variable declarations (continued)

You declare this variable	Db2 generates this variable

`SQL TYPE IS XML AS CLOB(40000K) clob_xml;`	`struct`
	`{ unsigned long length;`
	`  char data??(40960000??);`
	`} clob_xml;`

`SQL TYPE IS XML AS DBCLOB (4000K) dbclob_xml;`	`struct`
	`{ unsigned long length;`
	`  unsigned short data??(4096000??);`
	`} dbclob_xml;`

`SQL TYPE IS XML AS BLOB_FILE blob_xml_file;`	`struct {`
	`unsigned long name_length;`
	`unsigned long data_length;`
	`unsigned long file_options;`
	`char name??(255??);`
	`} blob_xml_file;`

`SQL TYPE IS XML AS CLOB_FILE clob_xml_file;`	`struct {`
	`unsigned long name_length;`
	`unsigned long data_length;`
	`unsigned long file_options;`
	`char name??(255??);`
	`} clob_xml_file;`

`SQL TYPE IS XML AS DBCLOB_FILE dbclob_xml_file;`	`struct {`
	`unsigned long name_length;`
	`unsigned long data_length;`
	`unsigned long file_options;`
	`char name??(255??);`
	`} dbclob_xml_file;`

Declarations of XML host variables in COBOL

The declarations that are generated for COBOL differ, depending on whether you use the Db2 precompiler or the Db2 coprocessor.

The following table shows COBOL declarations that the Db2 precompiler generates for some typical XML types.

Table 28. Examples of COBOL variable declarations by the Db2 precompiler

You declare this variable	Db2 precompiler generates this variable
`01  BLOB-XML USAGE IS` `    SQL TYPE IS XML AS BLOB(1M).`	`01  BLOB-XML.` `  02  BLOB-XML-LENGTH` `      PIC 9(9) COMP.` `  02  BLOB-XML-DATA.` ` 49 FILLER PIC X(32767).``"1" on page 111` ` 49 FILLER PIC X(32767).` ` *Repeat 30 times*` ⋮ ` 49 FILLER` ` PIC X(1048576-32*32767).`

Table 28. Examples of COBOL variable declarations by the Db2 precompiler (continued)

You declare this variable	Db2 precompiler generates this variable
`01  CLOB-XML USAGE IS` `    SQL TYPE IS XML AS CLOB(40000K).`	`01  CLOB-XML.` `  02  CLOB-XML-LENGTH` `      PIC 9(9) COMP.` `  02  CLOB-XML-DATA.` ` 49 FILLER PIC X(32767).``"1" on page 111` ` 49 FILLER PIC X(32767).` ` *Repeat 1248 times*` `⋮` ` 49 FILLER` ` PIC X(40960000-1250*32767).`
`01  DBCLOB-XML USAGE IS` `  SQL TYPE IS XML AS DBCLOB(4000K).`	`01  DBCLOB-XML.` `  02  DBCLOB-XML-LENGTH` `      PIC 9(9) COMP.` `  02  DBCLOB-XML-DATA.` `      49  FILLER PIC G(32767)` ` USAGE DISPLAY-1.``"2" on page 111` ` 49 FILLER PIC G(32767)` ` USAGE DISPLAY-1.` ` *Repeat 123 times*` `⋮` ` 49 FILLER` ` PIC G(4096000-125*32767)` ` USAGE DISPLAY-1.`

`01  BLOB-XML-FILE USAGE IS SQL`
`  TYPE IS XML AS BLOB-FILE.`
	`01  BLOB-XML-FILE.`
	`  49 BLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC.`
	`  49 BLOB-XML-FILE-DATA-LENGTH PIC S9(9) COMP-5.`
	`  49 BLOB-XML-FILE-FILE-OPTION PIC S9(9) COMP-5.`
	`  49 BLOB-XML-FILE-NAME PIC X(255).`

`01  CLOB-XML-FILE USAGE IS SQL`
`    TYPE IS XML AS CLOB-FILE.`
	`01  CLOB-XML-FILE.`
	`  49 CLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC.`
	`  49 CLOB-XML-FILE-DATA-LENGTH PIC S9(9) COMP-5.`
	`  49 CLOB-XML-FILE-FILE-OPTION PIC S9(9) COMP-5.`
	`  49 CLOB-XML-FILE-NAME PIC X(255).`

`01  DBCLOB-XML-FILE USAGE IS SQL`
`    TYPE IS XML AS DBCLOB-FILE.`
	`01  DBCLOB-XML-`
	`FILE.`
	`  49 DBCLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5`
	`SYNC.`
	`  49 DBCLOB-XML-FILE-DATA-LENGTH PIC S9(9)`
	`COMP-5.`
	`  49 DBCLOB-XML-FILE-FILE-OPTION PIC S9(9)`
	`COMP-5.`
	`  49 DBCLOB-XML-FILE-NAME PIC X(255).`

Notes:

1. For XML AS BLOB or XML AS CLOB host variables that are greater than 32767 bytes in length, Db2 creates multiple host language declarations of 32767 or fewer bytes.
2. For XML AS DBCLOB host variables that are greater than 32767 double-byte characters in length, Db2 creates multiple host language declarations of 32767 or fewer double-byte characters.

Declarations of XML host variables in PL/I

The declarations that are generated for PL/I differ, depending on whether you use the Db2 precompiler or the Db2 coprocessor.

The following table shows PL/I declarations that the Db2 precompiler generates for some typical XML types.

Table 29. Examples of PL/I variable declarations

You declare this variable	Db2 precompiler generates this variable

`DCL BLOB_XML`
`  SQL TYPE IS XML AS BLOB (1M);`
	`DCL`
	`   1   BLOB_XML,`
	`      2   BLOB_XML_LENGTH BIN FIXED(31),`
	`      2   BLOB_XML_DATA,`“1” on page 113
	`        3   BLOB_XML_DATA1 (32) CHAR(32767),`
	`        3   BLOB_XML_DATA2 CHAR(32);`

`DCL CLOB_XML`
`  SQL TYPE IS XML AS CLOB (40000K);`
	`DCL`
	`   1   CLOB_XML,`
	`      2   CLOB_XML_LENGTH BIN FIXED(31),`
	`      2   CLOB_XML_DATA,`“1” on page 113
	`        3   CLOB_XML_DATA1 (1250) CHAR(32767),`
	`        3   CLOB_XML_DATA2 CHAR(1250);`

`DCL DBCLOB_XML`
`  SQL TYPE IS XML AS DBCLOB (4000K);`
	`DCL`
	`   1   DBCLOB_XML,`
	`      2   DBCLOB_XML_LENGTH BIN FIXED(31),`
	`      2   DBCLOB_XML_DATA,`“2” on page 113
	`        3   DBCLOB_XML_DATA1 (250  ) GRAPHIC(16383),`
	`        3   DBCLOB_XML_DATA2 GRAPHIC(250);`

`DCL BLOB_XML_FILE`
`  SQL TYPE IS XML AS BLOB_FILE;`
	`DCL`
	`  1`
	`BLOB_XML_FILE,`
	`      2   BLOB_XML_FILE_NAME_LENGTH BIN FIXED(31)`
	`ALIGNED,`
	`      2   BLOB_XML_FILE_DATA_LENGTH BIN`
	`FIXED(31),`
	`      2   BLOB_XML_FILE_FILE_OPTIONS BIN`
	`FIXED(31),`
	`      2   BLOB_XML_FILE_NAME`
	`CHAR(255);`

`DCL CLOB_XML_FILE`
`  SQL TYPE IS XML AS CLOB_FILE;`
	`DCL`
	`  1`
	`CLOB_XML_FILE,`
	`      2   CLOB_XML_FILE_NAME_LENGTH BIN FIXED(31)`
	`ALIGNED,`
	`      2   CLOB_XML_FILE_DATA_LENGTH BIN`
	`FIXED(31),`
	`      2   CLOB_XML_FILE_FILE_OPTIONS BIN`
	`FIXED(31),`
	`      2   CLOB_XML_FILE_NAME`
	`CHAR(255);`

`DCL DBCLOB_XML_FILE SQL TYPE IS XML AS`
`  DBCLOB_FILE;`
	`DCL`
	`  1`
	`DBCLOB_XML_FILE,`
	`      2   DBCLOB_XML_FILE_NAME_LENGTH BIN FIXED(31)`
	`ALIGNED,`
	`      2   DBCLOB_XML_FILE_DATA_LENGTH BIN`
	`FIXED(31),`
	`      2   DBCLOB_XML_FILE_FILE_OPTIONS BIN`
	`FIXED(31),`
	`      2   DBCLOB_XML_FILE_NAME`
	`CHAR(255);`

Table 29. Examples of PL/I variable declarations (continued)

You declare this variable	Db2 precompiler generates this variable

Notes:

1. For XML AS BLOB or XML AS CLOB host variables that are greater than 32767 bytes in length, Db2 creates host language declarations in the following way:
   - If the length of the XML is greater than 32767 bytes and evenly divisible by 32767, Db2 creates an array of 32767-byte strings. The dimension of the array is *length*/32767.
   - If the length of the XML is greater than 32767 bytes but not evenly divisible by 32767, Db2 creates two declarations: The first is an array of 32767 byte strings, where the dimension of the array, *n*, is *length*/32767. The second is a character string of length *length*-*n**32767.
2. For XML AS DBCLOB host variables that are greater than 16383 double-byte characters in length, Db2 creates host language declarations in the following way:
   - If the length of the XML is greater than 16383 characters and evenly divisible by 16383, Db2 creates an array of 16383-character strings. The dimension of the array is *length*/16383.
   - If the length of the XML is greater than 16383 characters but not evenly divisible by 16383, Db2 creates two declarations: The first is an array of 16383 byte strings, where the dimension of the array, *m*, is *length*/16383. The second is a character string of length *length*-*m**16383.

Related concepts

Insertion of rows with XML column values
To insert rows into a table that contains XML columns, you can use the SQL INSERT statement.

Retrieving XML data
You can retrieve entire XML documents from XML columns by using an SQL SELECT statement. Alternatively, you can use SQL with XML extensions (SQL/XML) to retrieve portions of documents.

Updates of XML columns
To update entire documents in an XML column, you can use the SQL UPDATE statement. You can include a WHERE clause when you want to update specific rows. To update portions of XML documents, use the XMLMODIFY function with a basic XQuery updating expression.

XML column updates in embedded SQL applications

When you update or insert data into XML columns of a Db2 table, the input data must be in the textual XML format.

The encoding of XML data can be derived from the data itself, which is known as *internally encoded* data, or from external sources, which is known as *externally encoded* data. XML data that is sent to the database server as binary data is treated as internally encoded data. XML data that is sent to the database server as character data is treated as externally encoded data.

Externally encoded data can have internal encoding. That is, the data might be sent to the database server as character data, but the data contains encoding information. Db2 does not enforce consistency of the internal and external encoding. When the internal and external encoding information differs, the external encoding takes precedence. However, if there is a difference between the external and internal encoding, intervening character conversion might have occurred on the data, and there might be data loss.

Character data in XML columns is stored in UTF-8 encoding. The database server handles conversion of the data from its internal or external encoding to UTF-8.

The following examples demonstrate how to update XML columns in assembler, C, COBOL, and PL/I applications. The examples use a table named MYCUSTOMER, which is a copy of the sample CUSTOMER table.

Example: The following example shows an assembler program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data,

so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```
*********************************************************************
* UPDATE AN XML COLUMN WITH DATA IN AN XML AS CLOB HOST VARIABLE     *
*********************************************************************
         EXEC SQL                                                    +
               UPDATE MYCUSTOMER                                     +
               SET INFO = :XMLBUF                                    +
               WHERE CID = 1000
*********************************************************************
* UPDATE AN XML COLUMN WITH DATA IN AN XML AS BLOB HOST VARIABLE     *
*********************************************************************
         EXEC SQL                                                    +
               UPDATE MYCUSTOMER                                     +
               SET INFO = :XMLBLOB                                   +
               WHERE CID = 1000
*********************************************************************
* UPDATE AN XML COLUMN WITH DATA IN A CLOB HOST VARIABLE. USE        *
* THE XMLPARSE FUNCTION TO CONVERT THE DATA TO THE XML TYPE.         *
*********************************************************************
         EXEC SQL                                                    +
               UPDATE MYCUSTOMER                                     +
               SET INFO = XMLPARSE(DOCUMENT :CLOBBUF)                +
               WHERE CID = 1000
…
          LTORG
*****************************
* HOST VARIABLE DECLARATIONS *
*****************************
XMLBUF   SQL TYPE IS XML AS CLOB 10K
XMLBLOB  SQL TYPE IS XML AS BLOB 10K
CLOBBUF  SQL TYPE IS CLOB 10K
```

Example: The following example shows a C language program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```
/*****************************/
/* Host variable declarations */
/*****************************/
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS XML AS CLOB( 10K ) xmlBuf;
SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
SQL TYPE IS CLOB( 10K ) clobBuf;
EXEC SQL END DECLARE SECTION;
/*******************************************************************/
/* Update an XML column with data in an XML AS CLOB host variable */
/*******************************************************************/
EXEC SQL UPDATE MYCUSTOMER SET INFO = :xmlBuf where CID = 1000;
/*******************************************************************/
/* Update an XML column with data in an XML AS BLOB host variable */
/*******************************************************************/
 EXEC SQL UPDATE MYCUSTOMER SET INFO = :xmlblob where CID = 1000;
/*******************************************************************/
/* Update an XML column with data in a CLOB host variable. Use    */
/* the XMLPARSE function to convert the data to the XML type.     */
/*******************************************************************/
EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :clobBuf) where CID = 1000;
```

Example: The following example shows a COBOL program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```
*****************************
* Host variable declarations *
*****************************
 01 XMLBUF USAGE IS SQL TYPE IS XML as CLOB(10K).
 01 XMLBLOB  USAGE IS SQL TYPE IS XML AS BLOB(10K).
 01 CLOBBUF USAGE IS SQL TYPE IS CLOB(10K).
*****************************************************************
* Update an XML column with data in an XML AS CLOB host variable  *
*****************************************************************
 EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBUF where CID = 1000.
```

```
*********************************************************************
* Update an XML column with data in an XML AS BLOB host variable  *
*********************************************************************
 EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBLOB where CID = 1000.
*********************************************************************
* Update an XML column with data in a CLOB host variable. Use     *
* the XMLPARSE function to convert the data to the XML type.      *
*********************************************************************
 EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :CLOBBUF) where CID = 1000.
```

Example: The following example shows a PL/I program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```
/*****************************/
/* Host variable declarations */
/*****************************/
 DCL
  XMLBUF SQL TYPE IS XML AS CLOB(10K),
  XMLBLOB SQL TYPE IS XML AS BLOB(10K),
  CLOBBUF SQL TYPE IS CLOB(10K);
/*********************************************************************/
/* Update an XML column with data in an XML AS CLOB host variable  */
/*********************************************************************/
 EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBUF where CID = 1000;
/*********************************************************************/
/* Update an XML column with data in an XML AS BLOB host variable  */
/*********************************************************************/
 EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBLOB where CID = 1000;
/*********************************************************************/
/* Update an XML column with data in a CLOB host variable. Use     */
/* the XMLPARSE function to convert the data to the XML type.      */
/*********************************************************************/
 EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :CLOBBUF) where CID = 1000;
```

Insertion of rows with XML column values

To insert rows into a table that contains XML columns, you can use the SQL INSERT statement.

Updates of XML columns

To update entire documents in an XML column, you can use the SQL UPDATE statement. You can include a WHERE clause when you want to update specific rows. To update portions of XML documents, use the XMLMODIFY function with a basic XQuery updating expression.

XML data retrieval in embedded SQL applications

In an embedded SQL application, if you retrieve the data into a character host variable, Db2 converts the data from the UTF-8 encoding scheme to the application encoding scheme. If you retrieve the data into binary host variable, Db2 does not convert the data to another encoding scheme.

The output data is in the textual XML format.

Db2 might add an XML encoding specification to the retrieved data, depending on whether you call the XMLSERIALIZE function when you retrieve the data. If you do not call the XMLSERIALIZE function, Db2 adds the correct XML encoding specification to the retrieved data. If you call the XMLSERIALIZE function, Db2 adds an internal XML encoding declaration for UTF-8 encoding if you specify INCLUDING XMLDECLARATION in the function call. When you use INCLUDING XMLDECLARATION, you need to ensure that the retrieved data is not converted from UTF-8 encoding to another encoding.

The following examples demonstrate how to retrieve data from XML columns in assembler, C, COBOL, and PL/I applications. The examples use a table named MYCUSTOMER, which is a copy of the sample CUSTOMER table.

Example: The following example shows an assembler program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as

character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```
*********************************************************************
* RETRIEVE XML COLUMN DATA INTO AN XML AS CLOB HOST VARIABLE         *
*********************************************************************
         EXEC SQL                                                    +
               SELECT INFO                                           +
               INTO :XMLBUF                                          +
               FROM MYCUSTOMER                                       +
               WHERE CID = 1000
*********************************************************************
* RETRIEVE XML COLUMN DATA INTO AN XML AS BLOB HOST VARIABLE         *
*********************************************************************
         EXEC SQL                                                    +
               SELECT INFO                                           +
               INTO :XMLBLOB                                         +
               FROM MYCUSTOMER                                       +
               WHERE CID = 1000
*********************************************************************
* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE.        *
* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE             *
* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML             *
* TYPE TO THE CLOB TYPE.                                            *
*********************************************************************
         EXEC SQL                                                    +
               SELECT XMLSERIALIZE(INFO AS CLOB(10K))                +
               INTO :CLOBBUF                                         +
               FROM MYCUSTOMER                                       +
               WHERE CID = 1000
…
          LTORG
*****************************
* HOST VARIABLE DECLARATIONS *
*****************************
XMLBUF    SQL TYPE IS XML AS CLOB 10K
XMLBLOB   SQL TYPE IS XML AS BLOB 10K
CLOBBUF   SQL TYPE IS CLOB 10K
```

Example: The following example shows a C language program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```
/*****************************/
/* Host variable declarations */
/*****************************/
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS XML AS CLOB( 10K ) xmlBuf;
SQL TYPE IS XML AS BLOB( 10K ) xmlBlob;
SQL TYPE IS CLOB( 10K ) clobBuf;
EXEC SQL END DECLARE SECTION;
/*********************************************************************/
/* Retrieve data from an XML column into an XML AS CLOB host variable */
/*********************************************************************/
EXEC SQL SELECT INFO   INTO :xmlBuf from myTable where CID = 1000;
/*********************************************************************/
/* Retrieve data from an XML column into an XML AS BLOB host variable */
/*********************************************************************/
EXEC SQL SELECT INFO   INTO :xmlBlob from myTable where CID = 1000;
/*********************************************************************/
/* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE.        */
/* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE             */
/* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML             */
/* TYPE TO THE CLOB TYPE.                                            */
/*********************************************************************/
EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K))
 INTO :clobBuf from myTable where CID = 1000;
```

Example: The following example shows a COBOL program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB

host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```
*****************************
* Host variable declarations *
*****************************
 01 XMLBUF  USAGE IS SQL TYPE IS XML AS CLOB(10K).
 01 XMLBLOB USAGE IS SQL TYPE IS XML AS BLOB(10K).
 01 CLOBBUF USAGE IS SQL TYPE IS CLOB(10K).
*************************************************************************
* Retrieve data from an XML column into an XML AS CLOB host variable *
*************************************************************************
 EXEC SQL SELECT INFO
   INTO :XMLBUF
   FROM MYTABLE
   WHERE CID = 1000
 END-EXEC.
*************************************************************************
* Retrieve data from an XML column into an XML AS BLOB host variable *
*************************************************************************
 EXEC SQL SELECT INFO
   INTO :XMLBLOB
   FROM MYTABLE
   WHERE CID = 1000
 END-EXEC.
*************************************************************************
* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE.        *
* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE             *
* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML             *
* TYPE TO THE CLOB TYPE.                                             *
*************************************************************************
 EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K))
  INTO :CLOBBUF
  FROM MYTABLE
  WHERE CID = 1000
 END-EXEC.
```

Example: The following example shows a PL/I program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```
/*****************************/
/* Host variable declarations */
/*****************************/
 DCL
  XMLBUF SQL TYPE IS XML AS CLOB(10K),
  XMLBLOB SQL TYPE IS XML AS BLOB(10K),
  CLOBBUF SQL TYPE IS CLOB(10K);
/*************************************************************************/
/* Retrieve data from an XML column into an XML AS CLOB host variable */
/*************************************************************************/
 EXEC SQL SELECT INFO   INTO :XMLBUF FROM MYTABLE WHERE CID = 1000;
/*************************************************************************/
/* Retrieve data from an XML column into an XML AS BLOB host variable */
/*************************************************************************/
 EXEC SQL SELECT INFO   INTO :XMLBLOB FROM MYTABLE WHERE CID = 1000;
/*************************************************************************/
/* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE.        */
/* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE             */
/* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML             */
/* TYPE TO THE CLOB TYPE.                                             */
/*************************************************************************/
 EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K))
  INTO :CLOBBUF FROM MYTABLE WHERE CID = 1000;
```

Retrieving XML data
You can retrieve entire XML documents from XML columns by using an SQL SELECT statement. Alternatively, you can use SQL with XML extensions (SQL/XML) to retrieve portions of documents.

XML data in ODBC applications

In Db2 tables, the XML built-in data type is used to store XML data in a column as a structured set of nodes in a tree format.

The ODBC symbolic SQL data type SQL_XML corresponds to the Db2 XML data type. The symbolic C data types that you can use for updating XML columns or retrieving data from XML columns are SQL_C_BINARY, SQL_C_CHAR, SQL_C_DBCHAR or SQL_C_WCHAR. The default C data type is SQL_C_BINARY, which is also the recommended data type because it enables the data to be manipulated in its native format. This data type reduces conversion overhead and minimizes data loss that can result from the conversions.

XML column updates in ODBC applications

When you update or insert data into XML columns of a Db2 table, the input data can be in textual format or Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format (binary XML format)

For XML data, when you use `SQLBindParameter()` or `SQLSetParam()` to bind parameter markers to input data buffers, you can specify the data type of the input data buffer (*fCType*) as one of the following types:

- SQL_C_BINARY
- SQL_C_BINARYXML
- SQL_C_CHAR
- SQL_C_DBCHAR
- SQL_C_WCHAR.

When you bind a data buffer that contains XML data as SQL_C_BINARY, ODBC processes the XML data as internally encoded data. This is the preferred method because it avoids the overhead and potential data loss of character conversion.

Important: If the XML data is encoded in an encoding scheme and CCSID other than the application encoding scheme, you need to include internal encoding in the data and bind the data as SQL_C_BINARY to avoid character conversion.

When you bind a data buffer that contains XML data as SQL_C_CHAR, SQL_C_DBCHAR or SQL_C_WCHAR, ODBC processes the XML data as externally encoded data. ODBC determines the encoding of the data as follows:

- If the *fCType* value is SQL_C_WCHAR, ODBC assumes that the data is encoded as UCS-2.
- If the *fCType* value is SQL_C_CHAR or SQL_C_DBCHAR, ODBC assumes that the data is encoded in the application encoding scheme.

SQL_C_BINARYXML is neither internally encoded nor externally encoded. SQL_C_BINARYXML is in binary XML format, as opposed to textual XML format, and it has no encoding.

If you want Db2 to do an implicit XMLPARSE on the data before storing it in an XML column, the parameter marker data type in `SQLBindParameter()` or `SQLSetParam()` (*fsqlType*) must be specified as SQL_XML.

If you do an explicit XMLPARSE on the data, the parameter marker data type in `SQLBindParameter()` or `SQLSetParam()` (*fsqlType*) can be specified as any character or binary data type.

Example of inserting XML data into an XML column: The following example shows how to insert XML data into an XML column by using various C and SQL data types.

```
/* Variables for input XML data                 */
SQLCHAR     HVCHAR[32768];
SQLWCHAR    HVWCHAR[32768];
```

```
/* Variables for input XML data lengths          */
SQLINTEGER  LEN_HVCHAR;
SQLINTEGER  LEN_HVWCHAR;
/* SQL statement buffer                          */
SQLCHAR     sqlstmt[250];
/* Return code for ODBC calls                    */
SQLRETURN   rc = SQL_SUCCESS;
/* Prepare an INSERT statement for inserting     */
/* data into an XML column. The input parameter  */
/* type is SQL_XML, so DB2 does an implicit      */
/* XMLPARSE.                                     */
strcpy((char *)sqlstmt,
  "INSERT INTO MYTABLE(XMLCOL) VALUES(?)");
/* Bind input XML data with the SQL_C_CHAR type, */
/* to an SQL_XML SQL type.                       */
/* The data is assumed to be externally encoded, */
/* in the application encoding scheme.           */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_XML,
  0, 0, HVCHAR, sizeof(HVCHAR), &LEN_HVCHAR);
/* Execute the INSERT statement */
rc = SQLExecute(hstmt);
/* Bind input XML data with the SQL_C_WCHAR type, */
/* to an SQL_XML SQL type.                        */
/* The data is assumed to be externally encoded,  */
/* in the UCS-2 encoding scheme.                  */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_WCHAR, SQL_XML,
  0, 0, HVWCHAR, sizeof(HVWCHAR0, &LEN_HVWCHAR);
/* Execute the INSERT statement */
rc = SQLExecute(hstmt);
/* Prepare an INSERT statement for inserting     */
/* data into an XML column. The input parameter  */
/* type is SQL_CLOB, so the application must     */
/* do an explicit XMLPARSE.                      */
strcpy((char *)sqlstmt,
  "INSERT INTO MYTABLE (XMLCOL) ");
strcat((char *)sqlstmt,
  "VALUES(XMLPARSE(DOCUMENT CAST ? AS CLOB))");
/* Bind input XML data with the SQL_C_CHAR type,  */
/* to an SQL_CLOB SQL type.                       */
/* An explicit XMLPARSE is required for inserting */
/* character data into an XML column.             */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CLOB,
  32768, 0, HVCHAR, sizeof(HVCHAR), &LEN_HVCHAR);
/* Execute the INSERT statement */
rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);
```

Figure 5. Example of inserting XML data into an XML column

Example of inserting binary XML data into an XML column: The following example shows how to insert binary XML data into an XML column by using the SQL_C_BINARYXML data type.

```
CREATE TABLE MYTABLE ( XML_COL XML );

/* Declare variables for binary XML data */
SQLCHAR HV1BINARYXML[100];
SQLINTEGER LEN_HV1BINARYXML;
SQLCHAR sqlstmt[250];
SQLRETURN rc = SQL_SUCCESS;

/* Assume that HV1BINARYXML contains XML data in binary format
   and LEN_HV1BINARYXML contains the length of data in bytes */

/* Prepare insert statement */
strcpy((char *)sqlstmt, "insert into mytable values(?)");
rc = SQLPrepare(hstmt, sqlstmt, SQL_NTS);

/* Bind XML_COL column as SQL_C_BINARYXML */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARYXML, SQL_XML,
               0, 0, HV1BINARYXML, sizeof(HV1BINARYXML), &LEN_HV1BINARYXML);

/* Execute insert */
rc = SQLExecute(hstmt);
```

Figure 6. Example of inserting binary XML data into an XML column

XML data retrieval in ODBC applications

When you select data from XML columns in a Db2 table, the output data is in textual format or Extensible Dynamic Binary XML Db2 Client/Server Binary XML Format (binary XML format).

For XML data, when you use `SQLBindCol()` to bind columns in a query result set to application variables, you can specify the data type of the application variables (*fCType*) as one of the following types:

- SQL_C_BINARY
- SQL_C_BINARYXML
- SQL_C_CHAR
- SQL_C_DBCHAR
- SQL_C_WCHAR.

The data is returned as internally encoded data.

ODBC determines the encoding of the data as follows:

- If the *fCType* value is SQL_C_BINARY, ODBC returns the data in the UTF-8 encoding scheme.
- If the *fCType* value is SQL_C_BINARYXML, ODBC returns the data in binary XML format.
- If the *fCType* value is SQL_C_CHAR or SQL_C_DBCHAR, ODBC returns the data in the application encoding scheme.
- If the *fCType* value is SQL_C_WCHAR, ODBC returns the data in the UCS-2 encoding scheme.

Db2 performs an implicit XMLSERIALIZE on the data before returning it to your application.

For applications that use the SQL_C_BINARYXML data type, set LIMITEDBLOCKFETCH to 0. Otherwise, if you attempt to use the SQLGetData() function to retrieve XML data and have LIMITEDBLOCKFETCH set to 1, the function call fails.

Example of retrieving XML data from an XML column: The following example shows how to retrieve XML data from an XML column into application variables with various C data types.

```
/* Variables for output XML data                   */
SQLCHAR     HVBINARY[32768];
SQLCHAR     HVCHAR[32768];
SQLDBCHAR   HVDBCHAR[32768];
SQLWCHAR    HVWCHAR[32768];
/* Variables for output XML data lengths           */
SQLINTEGER  LEN_HVBINARY;
SQLINTEGER  LEN_HVCHAR;
SQLINTEGER  LEN_HVDBCHAR;
SQLINTEGER  LEN_HVWCHAR;
/* SQL statement buffer                            */
SQLCHAR     sqlstmt[250];
/* Return code for ODBC calls                      */
SQLRETURN   rc = SQL_SUCCESS;
/* Prepare an SELECT statement for retrieving      */
/* data from XML columns.                          */
strcpy((char *)sqlstmt,
  "SELECT XMLCOL1, XMLCOL2, XMLCOL3, XMLCOL4 ");
strcat((char *)sqlstmt,
  "FROM MYTABLE");
/* Bind data for first XML column as SQL_C_BINARY. */
/* This data will be retrieved as internally       */
/* encoded, in the UTF-8 encoding scheme.          */
rc = SQLBindCol(hstmt, 1, SQL_C_BINARY, HVBINARY, sizeof(HVBINARY), &LEN_HVBINARY);
/* Bind data for second XML column as              */
/* SQL_C_CHAR. This data will be retrieved as      */
/* internally encoded, in the application encoding */
/* scheme.                                         */
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, HVCHAR, sizeof(HVCHAR), &LEN_HVCHAR);
/* Bind data for third XML column as SQL_C_DBCHAR. */
/* This data will be retrieved as internally       */
/* encoded, in the application encoding scheme.    */
rc = SQLBindCol(hstmt, 3, SQL_C_DBCHAR, HVDBCHAR, sizeof(HVDBCHAR), &LEN_HVDBCHAR);
/* Bind data for fourth XML column as SQL_C_WCHAR. */
/* This data will be retrieved as internally       */
/* encoded, in the UCS-2 encoding scheme.          */
rc = SQLBindCol(hstmt, 4, SQL_C_WCHAR, HVWCHAR, sizeof(HVWCHAR), &LEN_HVWCHAR);
/* Execute the SELECT statement and fetch a row    */
/* from the result set                             */
rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);
rc = SQLFetch(hstmt);
```

Figure 7. Example of retrieving XML data from an XML column

Example of retrieving binary XML data from an XML column: The following example shows how to retrieve binary XML data from an XML column into application variables by using type SQL_C_BINARYXML.

```
CREATE TABLE MYTABLE ( XML_COL XML );

/* Declare variables for binary XML data */
SQLCHAR HV1BINARYXML[100];
SQLINTEGER LEN_HV1BINARYXML;
SQLCHAR sqlstmt[250];
SQLRETURN rc = SQL_SUCCESS;

/* Prepare select statement */
strcpy((char *)sqlstmt, "select * from mytable");
rc = SQLPrepare(hstmt, sqlstmt, SQL_NTS);

/* Bind column data as SQL_C_BINARYXML */
rc = SQLBindCol(hstmt, 1, SQL_C_BINARYXML, sizeof(HV1BINARYXML), &LEN_HV1BINARYXML);

/* Execute select */
rc = SQLExecute(hstmt);
/* Fetch result set column as binary XML */
rc = SQLFetch(hstmt);
```

Figure 8. Example of retrieving binary XML data from an XML column

Data types for archiving XML documents

Use a non-XML column type to store an XML document in a Db2 table only for archiving.

When you store the data in an XML column, the Db2 database server transforms the data into an internal format. When you retrieve the document, the retrieved data is not exactly the same as the original textual XML document. Therefore, the XML column type is not appropriate for archiving.

The best column data type for archiving XML data is a binary data type, such as BLOB. A BLOB type does not store the external encoding of the data, so the XML documents need to have an internal encoding declaration that indicates the original encoding.

Related concepts

XML data encoding

The encoding of XML data can be derived from the data itself, which is known as *internally encoded* data, or from external sources, which is known as *externally encoded* data.

Chapter 7. XML data encoding

The encoding of XML data can be derived from the data itself, which is known as *internally encoded* data, or from external sources, which is known as *externally encoded* data.

The application data type that you use to exchange the XML data between the application and the XML column determines how the encoding is derived.

- XML data that is in character or graphic application data types is considered to be externally encoded. Like character and graphic data, XML data that is in these data types is considered to be encoded in:
  - The application code page, if the encoding is not specified in an SQLDA
  - The value that is specified by the CCSID, if a CCSID is specified in an SQLDA
- XML data that is in a binary application data type or has a bit data subtype is considered to be internally encoded.

Externally coded XML data might contain internal encoding, such as when an XML document in a character data type contains an encoding declaration. When you send externally encoded data to a Db2 database, the database manager ignores internal encoding.

Related concepts

Db2 application programming language support for XML
You can write applications to store XML data in Db2 database tables or retrieve XML data from tables. XML parameters for external stored procedures or user-defined functions are not supported.

Background information on XML internal encoding

XML data in a binary application data type has internal encoding. With internal encoding, the content of the data determines the encoding.

The Db2 database system derives the internal encoding from the document content according to the XML standard.

Internal encoding is derived from three components:

Unicode Byte Order Mark (BOM)
A byte sequence that consists of a Unicode character code at the beginning of XML data. The BOM indicates the byte order of the following text. The Db2 database manager recognizes a BOM only for XML data. For XML data that is stored in a non-XML column, the database manager treats a BOM value like any other character or binary value.

XML declaration
A special tag at the beginning of an XML document. The declaration provides specific details about the remainder of the XML.

Encoding declaration
An optional part of the XML declaration that specifies the encoding for the characters in the document.

The Db2 database manager uses the following procedure to determine the encoding:

1. If the data contains a Unicode BOM, the BOM determines the encoding. The following table lists the BOM types and the resultant data encoding:

Table 30. Byte order marks and resultant document encoding

BOM type	BOM value	Encoding
UTF-8	X'EFBBBF'	UTF-8
UTF-16 Big Endian	X'FEFF'	UTF-16
UTF-16 Little Endian	X'FFFE'	UTF-16

© Copyright IBM Corp. 2007, 2020

2. If the data contains an XML declaration, the encoding depends on whether there is an encoding declaration:
   - If there is an encoding declaration, the encoding is the value of the encoding attribute. For example, the encoding is EUC-JP for XML data with the following XML declaration:

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

   - If there is an encoding declaration and a BOM, the encoding declaration must match the encoding from the BOM. Otherwise, an error occurs.
   - If there is no encoding declaration and no BOM, the database manager determines the encoding from the encoding of the XML declaration:
     - If the XML declaration is in single-byte ASCII characters, the encoding of the document is UTF-8.
     - If the XML declaration is in double-byte ASCII characters, the encoding of the document is UTF-16.
3. If there is no XML declaration and no BOM, the encoding of the document is UTF-8.

Related information

XML encoding considerations
Following some guidelines helps you to avoid encoding issues when you store or retrieve XML data in Db2 tables.

XML encoding scenarios
Internal or external encoding and implicit or explicit XML serialization can affect data conversion during storage and retrieval of XML values in Db2 tables.

XML encoding considerations

Following some guidelines helps you to avoid encoding issues when you store or retrieve XML data in Db2 tables.

Related concepts

Background information on XML internal encoding
XML data in a binary application data type has internal encoding. With internal encoding, the content of the data determines the encoding.

Related information

XML encoding scenarios
Internal or external encoding and implicit or explicit XML serialization can affect data conversion during storage and retrieval of XML values in Db2 tables.

Encoding considerations for input of XML data to a Db2 table

When you update XML data in a Db2 table, you need to avoid data loss.

When you store XML data in a Db2 table, observe the following rules:

- For externally encoded XML data (data that is sent to the database server using character data types), any internally encoded declaration needs to match the external encoding.
- For internally encoded XML data (data that is sent to the database server using binary data types), the application *must* ensure that the data contains accurate encoding information.

Related concepts

Background information on XML internal encoding
XML data in a binary application data type has internal encoding. With internal encoding, the content of the data determines the encoding.

Related reference

Mappings of encoding names to effective CCSIDs for stored XML data

Each encoding name maps to a specific CCSID.

Related information

XML encoding scenarios
Internal or external encoding and implicit or explicit XML serialization can affect data conversion during storage and retrieval of XML values in Db2 tables.

Encoding considerations for retrieval of XML data from a Db2 table

When you retrieve XML data from a Db2 table, you need to avoid data loss and truncation.

Data loss can occur when characters in the source data cannot be represented in the encoding of the target data. Truncation can occur when:

- Conversion to the target data type results in expansion of the data.
- The application host variable length is not set or is set too small.

  Because DESCRIBE does not return a length for XML columns, applications that use an SQLDA must set a length in the SQLDA that reflects the amount of application storage available for each application variable. Applications can provide enough storage to hold the largest retrieved documents, or use FETCH CONTINUE to retrieve large XML documents in pieces.

Data loss is less of a problem for Java applications than for other types of applications because Java string data types use Unicode UTF-16 or UCS2 encoding. Truncation is possible because expansion can occur when UTF-8 characters are converted to UTF-16 or UCS-2 encoding.

Related concepts

Background information on XML internal encoding
XML data in a binary application data type has internal encoding. With internal encoding, the content of the data determines the encoding.

Related reference

Mappings of CCSIDs to encoding names for textual XML output data
As part of an implicit or explicit XMLSERIALIZE operation, Db2 might add an encoding declaration at the beginning of textual XML output data.

Related information

XML encoding scenarios
Internal or external encoding and implicit or explicit XML serialization can affect data conversion during storage and retrieval of XML values in Db2 tables.

XML data encoding in JDBC and SQLJ applications

In general, Java applications have fewer XML encoding issues than Db2 ODBC or embedded SQL applications because the application code page is always Unicode.

Although the encoding considerations for internally encoded XML data are the same for all applications, the situation is simplified for externally encoded data in Java applications.

General recommendations for input of XML data in Java applications

- If the input data is in a file, read the data in as a binary stream (`setBinaryStream`) so that the database manager processes it as internally encoded data.
- If the input data is in a Java application variable, your choice of application variable type determines whether the Db2 database manager uses any internal encoding. If you input the data as a character type (for example, `setString`), the database manager converts the data from UTF-16 (the application code page) to UTF-8 before parsing and storing it.

General recommendations for output of XML data in Java applications

- If you output XML data to a file as non-binary data, you should add XML internal encoding to the output data.

The encoding for the file system might not be Unicode, so string data can undergo conversion when it is stored in the file. If you write data to a file as binary data, conversion does not occur.

For Java applications, the database server does not add an explicit declaration for an implicit XML serialize operation. If you cast the output data as the `com.ibm.db2.jcc.DB2Xml` type, and invoke one of the `getDB2Xml`*xxx* methods, the JDBC driver adds an encoding declaration, as shown in the following table.

`getDB2Xml`*xxx*	**Encoding in declaration**
`getDB2XmlString`	ISO-10646-UCS-2
`getDB2XmlBytes(String` *`targetEncoding`*`)`	Encoding specified by *targetEncoding*
`getDB2XmlAsciiStream`	US-ASCII
`getDB2XmlCharacterStream`	ISO-10646-UCS-2
`getDB2XmlBinaryStream(String` *`targetEncoding`*`)`	Encoding specified by *targetEncoding*

For an explicit XMLSERIALIZE function with INCLUDING XMLDECLARATION, the database server adds encoding, and the JDBC driver does not modify it. The explicit encoding that the database server adds is UTF-8 encoding. Depending on how the value is retrieved by the application, the actual encoding of the data might not match the explicit internal encoding.

- If the application sends the output data to an XML parser, you should retrieve the data in a binary application variable, with UTF-8, UCS-2, or UTF-16 encoding.

Related information

XML data in Java applications
In Java applications, you can store XML data in Db2 databases or retrieve XML data from Db2 databases by using JDBC or SQLJ.

XML encoding scenarios

Internal or external encoding and implicit or explicit XML serialization can affect data conversion during storage and retrieval of XML values in Db2 tables.

Related concepts

Background information on XML internal encoding
XML data in a binary application data type has internal encoding. With internal encoding, the content of the data determines the encoding.

Related information

XML encoding considerations
Following some guidelines helps you to avoid encoding issues when you store or retrieve XML data in Db2 tables.

Encoding scenarios for input of internally encoded XML data to a Db2 table

Internal encoding can affect data conversion and truncation during input of XML data to an XML column.

In general, use of a binary application data type minimizes code page conversion problems during input to a Db2 table. The following examples demonstrate the effects of internal encoding during input.

Scenario 1

Encoding source	**Value**
Data encoding	UTF-8 Unicode input data, with or without a UTF-8 BOM or XML encoding declaration
Application data type	Binary

Encoding source	Value
Application code page	Not applicable

Example input statements:

```
INSERT INTO T1 (XMLCOL) VALUES (?)
INSERT INTO T1 (XMLCOL) VALUES
  (XMLPARSE(DOCUMENT CAST(? AS BLOB) PRESERVE WHITESPACE))
INSERT INTO T1 (XMLCOL) VALUES (XMLPARSE(DOCUMENT :HV))
```

Character conversion: None.

Data loss: None.

Truncation: None.

Scenario 2

Encoding source	Value
Data encoding	UTF-16 Unicode input data containing a UTF-16 BOM or XML encoding declaration
Application data type	Binary
Application code page	Not applicable

Example input statements:

```
INSERT INTO T1 (XMLCOL) VALUES (?)
INSERT INTO T1 (XMLCOL) VALUES
  (XMLPARSE(DOCUMENT CAST(? AS BLOB) PRESERVE WHITESPACE))
INSERT INTO T1 (XMLCOL) VALUES (XMLPARSE(DOCUMENT :HV))
```

Character conversion: The Db2 database system converts the data from UTF-16 to UTF-8 when it performs the XML parse for storage in the XML column.

Data loss or truncation: No data loss occurs. Truncation can occur during conversion from UTF-16 to UTF-8, due to expansion.

Scenario 3

Encoding source	Value
Data encoding	ISO-8859-1 input data containing an XML encoding declaration
Application data type	Binary
Application code page	Not applicable

Example input statements:

```
INSERT INTO T1 (XMLCOL) VALUES (?)
INSERT INTO T1 (XMLCOL) VALUES
  (XMLPARSE(DOCUMENT CAST(? AS BLOB) PRESERVE WHITESPACE))
INSERT INTO T1 VALUES (XMLPARSE(DOCUMENT :HV))
```

Character conversion: The Db2 database system converts the data from CCSID 819 to UTF-8 when it performs the XML parse for storage in the XML column.

Data loss: None.

Truncation: None.

Scenario 4

Encoding source	Value
Data encoding	Shift_JIS input data containing an XML encoding declaration
Application data type	Binary
Application code page	Not applicable

Example input statements:

```
INSERT INTO T1 (XMLCOL) VALUES (?)
INSERT INTO T1 (XMLCOL) VALUES
  (XMLPARSE(DOCUMENT CAST(? AS BLOB) PRESERVE WHITESPACE))
INSERT INTO T1 VALUES (XMLPARSE(DOCUMENT :HV))
```

Character conversion: The Db2 database system converts the data from CCSID 943 to UTF-8 when it performs the XML parse for storage in the XML column.

Data loss: None.

Truncation: None.

Related reference

Mappings of encoding names to effective CCSIDs for stored XML data
Each encoding name maps to a specific CCSID.

Related information

XML encoding considerations
Following some guidelines helps you to avoid encoding issues when you store or retrieve XML data in Db2 tables.

Encoding scenarios for input of externally encoded XML data to a database

In general, when you use a character application data type, problems with code page conversion do not occur during input to a database.

The following examples demonstrate how external encoding affects data conversion and truncation during input of XML data to an XML column.

Only scenario 1 and scenario 2 apply to Java and .NET applications, because the application code page for Java and .NET applications is always Unicode.

Scenario 1

Encoding source	Value
Data encoding	UTF-8 Unicode input data, with or without an appropriate encoding declaration or BOM
Application data type	Character
Application code page	1208 (UTF-8)

Example input statements:

```
INSERT INTO T1 (XMLCOL) VALUES (?)
INSERT INTO T1 (XMLCOL) VALUES
```

```
  (XMLPARSE(DOCUMENT CAST(? AS CLOB) PRESERVE WHITESPACE))
INSERT INTO T1 (XMLCOL) VALUES (XMLPARSE(DOCUMENT :HV))
```

Character conversion: None.

Data loss: None.

Truncation: None.

Scenario 2

Encoding source	Value
Data encoding	UTF-16 Unicode input data, with or without an appropriate encoding declaration or BOM
Application data type	Graphic
Application code page	Any SBCS code page or CCSID 1208

Example input statements:

```
INSERT INTO T1 (XMLCOL) VALUES (?)
INSERT INTO T1 (XMLCOL) VALUES
  (XMLPARSE(DOCUMENT CAST(? AS DBCLOB) PRESERVE WHITESPACE))
INSERT INTO T1 (XMLCOL) VALUES (XMLPARSE(DOCUMENT :HV))
```

Character conversion: The Db2 database system converts the data from UTF-16 to UTF-8 when it performs the XML parse for storage in the XML column.

Data loss: None.

Truncation: Truncation can occur during conversion from UTF-16 to UTF-8, due to expansion.

Scenario 3

Encoding source	Value
Data encoding	ISO-8859-1 input data, with or without an appropriate encoding declaration
Application data type	Character
Application code page	819

Example input statements:

```
INSERT INTO T1 (XMLCOL) VALUES (?)
INSERT INTO T1 (XMLCOL) VALUES
  (XMLPARSE(DOCUMENT CAST(? AS CLOB) PRESERVE WHITESPACE))
INSERT INTO T1 (XMLCOL) VALUES (XMLPARSE(DOCUMENT :HV))
```

Character conversion: The Db2 database system converts the data from CCSID 819 to UTF-8 when it performs the XML parse for storage in the XML column.

Data loss: None.

Truncation: None.

Scenario 4

Encoding source	Value
Data encoding	Shift_JIS input data, with or without an appropriate encoding declaration
Application data type	Graphic
Application code page	943

Example input statements:

```
INSERT INTO T1 VALUES (?)
INSERT INTO T1 VALUES
  (XMLPARSE(DOCUMENT CAST(? AS DBCLOB)))
INSERT INTO T1 VALUES (XMLPARSE(DOCUMENT :HV))
```

Character conversion: The Db2 database system converts the data from CCSID 943 to UTF-8 when it performs the XML parse for storage in the XML column.

Data loss: None.

Truncation: None.

Related reference

Mappings of encoding names to effective CCSIDs for stored XML data
Each encoding name maps to a specific CCSID.

Related information

XML encoding considerations
Following some guidelines helps you to avoid encoding issues when you store or retrieve XML data in Db2 tables.

Encoding scenarios for retrieval of XML data with implicit serialization

The target encoding and application code page can affect data conversion, truncation due to expansion, and internal encoding during XML data retrieval with implicit serialization.

The following examples demonstrate these interactions.

Only scenario 1 and scenario 2 apply to Java applications, because the application code page for Java applications is always Unicode. In general, code page conversion is not a problem for Java applications.

Scenario 1

Encoding source	Value
Target data encoding	UTF-8 Unicode
Target application data type	Binary
Application code page	Not applicable

Example output statements:

```
SELECT XMLCOL FROM T1
```

Character conversion: None.

Data loss: None.

Truncation due to expansion: None.

Internal encoding in the textual XML data: For applications other than Java applications, the data is prefixed by the following XML declaration:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

For Java applications, no encoding declaration is added, unless you cast the data as the `com.ibm.db2.jcc.DB2Xml` type, and use a `getDB2Xml`*xxx* method to retrieve the data. The declaration that is added depends on the `getDB2Xml`*xxx* that you use.

Scenario 2

Encoding source	Value
Target data encoding	UTF-16 Unicode
Target application data type	Graphic
Application code page	CCSID 1208

Example output statements:

```
SELECT XMLCOL FROM T1
```

Character conversion: Data is converted from UTF-8 to UTF-16.

Data loss: None.

Truncation due to expansion: Truncation can occur during conversion from UTF-8 to UTF-16, due to expansion.

Internal encoding in the textual XML data: For applications other than Java applications, the data is prefixed by a UTF-16 Byte Order Mark (BOM) and the following XML declaration:

```
<?xml version="1.0" encoding="UTF-16" ?>
```

For Java applications, no encoding declaration is added, unless you cast the data as the `com.ibm.db2.jcc.DB2Xml` type, and use a `getDB2Xml`*xxx* method to retrieve the data. The declaration that is added depends on the `getDB2Xml`*xxx* that you use.

Scenario 3

Encoding source	Value
Target data encoding	ISO-8859-1 data
Target application data type	Character
Application code page	819

Example output statements:

```
SELECT XMLCOL FROM T1
```

Character conversion: Data is converted from UTF-8 to CCSID 819.

Data loss: Possible data loss. Some UTF-8 characters cannot be represented in CCSID 819. The Db2 database system generates an error.

Truncation due to expansion: None.

Internal encoding in the textual XML data: The data is prefixed by the following XML declaration:

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

Scenario 4

Encoding source	Value
Target data encoding	Windows-31J data (superset of Shift_JIS)
Target application data type	Graphic
Application code page	943

Example output statements:

```
SELECT XMLCOL FROM T1
```

Character conversion: Data is converted from UTF-8 to CCSID 943.

Data loss: Possible data loss. Some UTF-8 characters cannot be represented in CCSID 943. The Db2 database system generates an error.

Truncation due to expansion: Truncation can occur during conversion from UTF-8 to CCSID 943 due to expansion.

Internal encoding in the textual XML data: The data is prefixed by the following XML declaration:

```
<?xml version="1.0" encoding="Windows-31J" ?>
```

Related concepts

Encoding considerations for retrieval of XML data from a Db2 table
When you retrieve XML data from a Db2 table, you need to avoid data loss and truncation.

Related reference

Mappings of CCSIDs to encoding names for textual XML output data
As part of an implicit or explicit XMLSERIALIZE operation, Db2 might add an encoding declaration at the beginning of textual XML output data.

Encoding scenarios for retrieval of XML data with explicit XMLSERIALIZE

The target encoding scheme and application code page can affect data conversion, truncation, and internal encoding during XML data retrieval with an explicit XMLSERIALIZE invocation.

The following examples demonstrate these interactions.

Data loss does not occur during conversion of the XML data to the type that is specified in the XMLSERIALIZE function because the input and output data is Unicode data. Data loss can occur during conversion of the result of the XMLSERIALIZE operation to the application data type. This data loss results in an SQL warning.

Truncation can occur at two points during data retrieval with an explicit XMLSERIALIZE:

- During conversion to the type that is specified in the XMLSERIALIZE function

  This truncation can occur because the size that you specify in the XMLSERIALIZE function for the output data type is too small. This truncation results in an SQL error.

- During conversion of the result of the XMLSERIALIZE operation to the application data type.

  This truncation can occur because the size that you specify for the host variable is too small. This truncation results in an SQL warning.

The following examples discuss only truncation that occurs because the size of a document increases during conversion to the output encoding.

Only scenario 1 and scenario 2 apply to Java applications, because the application code page for Java applications is always Unicode.

Scenario 1

Encoding source	Value
Target data encoding	UTF-8 Unicode
Target application data type	Binary
Application code page	Not applicable

Example output statements:

```
SELECT XMLSERIALIZE(XMLCOL AS BLOB(1M) INCLUDING XMLDECLARATION) FROM T1
```

Character conversion: None.

Data loss: None.

Truncation due to expansion: None.

Internal encoding in the textual XML data: The data is prefixed by the following XML declaration:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Scenario 2

Encoding source	Value
Target data encoding	UTF-16 Unicode
Target application data type	Graphic
Application code page	CCSID 1208

Example output statements:

```
SELECT XMLSERIALIZE(XMLCOL AS DBCLOB(1M) EXCLUDING XMLDECLARATION) FROM T1
```

Character conversion: Data is converted from UTF-8 to UTF-16.

Data loss: None.

Truncation due to expansion: Truncation can occur during conversion from UTF-8 to UTF-16, due to expansion.

Internal encoding in the textual XML data: None, because EXCLUDING XMLDECLARATION is specified. If INCLUDING XMLDECLARATION is specified, the internal encoding indicates UTF-8 instead of UTF-16.

This can result in XML data that cannot be parsed by application processes that rely on the encoding name.

Scenario 3

Encoding source	Value
Target data encoding	ISO-8859-1 data
Target application data type	Character
Application code page	819

Example output statements:

```
SELECT XMLSERIALIZE(XMLCOL AS CLOB(1M) EXCLUDING XMLDECLARATION) FROM T1
```

Character conversion: Data is converted from UTF-8 to CCSID 819.

Data loss: Possible data loss. Some UTF-8 characters cannot be represented in CCSID 819. If a character cannot be represented in CCSID 819, the Db2 database manager inserts a substitution character in the output and issues a warning.

Truncation due to expansion: None.

Internal encoding in the textual XML data: None, because EXCLUDING XMLDECLARATION is specified. If INCLUDING XMLDECLARATION is specified, the database manager adds internal encoding for UTF-8 instead of ISO-8859-1. This can result in XML data that cannot be parsed by application processes that rely on the encoding name.

Scenario 4

Encoding source	Value
Target data encoding	Windows-31J data (superset of Shift_JIS)
Target application data type	Graphic
Application code page	943

Example output statements:

```
SELECT XMLSERIALIZE(XMLCOL AS CLOB(1M) EXCLUDING XMLDECLARATION) FROM T1
```

Character conversion: Data is converted from UTF-8 to CCSID 943.

Data loss: Possible data loss. Some UTF-8 characters cannot be represented in CCSID 943. If a character cannot be represented in CCSID 943, the database manager inserts a substitution character in the output and issues a warning.

Truncation due to expansion: Truncation can occur during conversion from UTF-8 to CCSID 943 due to expansion.

Internal encoding in the textual XML data: None, because EXCLUDING XMLDECLARATION is specified. If INCLUDING XMLDECLARATION is specified, the internal encoding indicates UTF-8 instead of Windows-31J. This can result in XML data that cannot be parsed by application processes that rely on the encoding name.

Mappings of encoding names to effective CCSIDs for stored XML data

Each encoding name maps to a specific CCSID.

Db2 examines data that is in a binary application variable or an internally encoded XML type to determine the encoding before storing the data in an XML column. If the data has an encoding declaration, Db2 maps the encoding name to a CCSID.

The following table lists these mappings. If an encoding name is not in that table, Db2 returns an error.

The normalized encoding name in the first column of the following table is the result of converting the encoding name to uppercase, and stripping out all hyphens, plus signs, underscores, colons, periods, and spaces. For example, ISO88591 is the normalized encoding name for ISO 8859-1, ISO-8859-1, and iso-8859-1.

Table 31. Encoding names and effective CCSIDs

Normalized encoding name	CCSID
437	437
646	367
813	813
819	819
850	850
852	852
855	855
857	857
862	862
863	863
866	866
869	869
885913	901
885915	923
88591	819
88592	912
88595	915
88597	813
88598	62210
88599	920
904	904
912	912
915	915
916	916
920	920
923	923

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
ANSI1251	1251
ANSIX341968	367
ANSIX341986	367
ARABIC	1089
ASCII7	367
ASCII	367
ASMO708	1089
BIG5	950
CCSID00858	858
CCSID00924	924
CCSID01140	1140
CCSID01141	1141
CCSID01142	1142
CCSID01143	1143
CCSID01144	1144
CCSID01145	1145
CCSID01146	1146
CCSID01147	1147
CCSID01148	1148
CCSID01149	1149
CP00858	858
CP00924	924
CP01140	1140
CP01141	1141
CP01142	1142
CP01143	1143
CP01144	1144
CP01145	1145
CP01146	1146
CP01147	1147
CP01148	1148
CP01149	1149
CP037	37
CP1026	1026

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
CP1140	1140
CP1141	1141
CP1142	1142
CP1143	1143
CP1144	1144
CP1145	1145
CP1146	1146
CP1147	1147
CP1148	1148
CP1149	1149
CP1250	1250
CP1251	1251
CP1252	1252
CP1253	1253
CP1254	1254
CP1255	1255
CP1256	1256
CP1257	1257
CP1258	1258
CP1363	1363
CP1383	1383
CP1386	1386
CP273	273
CP277	277
CP278	278
CP280	280
CP284	284
CP285	285
CP297	297
CP33722	954
CP33722C	954
CP367	367
CP420	420
CP423	423

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
CP424	424
CP437	437
CP500	500
CP5346	5346
CP5347	5347
CP5348	5348
CP5349	5349
CP5350	5350
CP5353	5353
CP813	813
CP819	819
CP838	838
CP850	850
CP852	852
CP855	855
CP857	857
CP858	858
CP862	862
CP863	863
CP864	864
CP866	866
CP869	869
CP870	870
CP871	871
CP874	874
CP904	904
CP912	912
CP915	915
CP916	916
CP920	920
CP921	921
CP922	922
CP923	923
CP936	1386

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
CP943	943
CP943C	943
CP949	970
CP950	950
CP964	964
CP970	970
CPGR	869
CSASCII	367
CSBIG5	950
CSEBCDICCAFR	500
CSEBCDICDKNO	277
CSEBCDICES	284
CSEBCDICFISE	278
CSEBCDICFR	297
CSEBCDICIT	280
CSEBCDICPT	37
CSEBCDICUK	285
CSEBCDICUS	37
CSEUCKR	970
CSEUCPKDFMTJAPANESE	954
CSGB2312	1383
CSHPROMAN8	1051
CSIBM037	37
CSIBM1026	1026
CSIBM273	273
CSIBM277	277
CSIBM278	278
CSIBM280	280
CSIBM284	284
CSIBM285	285
CSIBM297	297
CSIBM420	420
CSIBM423	423
CSIBM424	424

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
CSIBM500	500
CSIBM855	855
CSIBM857	857
CSIBM863	863
CSIBM864	864
CSIBM866	866
CSIBM869	869
CSIBM870	870
CSIBM871	871
CSIBM904	904
CSIBMEBCDICATDE	273
CSIBMTHAI	838
CSISO128T101G2	920
CSISO146SERBIAN	915
CSISO147MACEDONIAN	915
CSISO2INTLREFVERSION	367
CSISO646BASIC1983	367
CSISO88596I	1089
CSISO88598I	916
CSISOLATIN0	923
CSISOLATIN1	819
CSISOLATIN2	912
CSISOLATIN5	920
CSISOLATIN9	923
CSISOLATINARABIC	1089
CSISOLATINCYRILLIC	915
CSISOLATINGREEK	813
CSISOLATINHEBREW	62210
CSKOI8R	878
CSKSC56011987	970
CSMACINTOSH	1275
CSMICROSOFTPUBLISHING	1004
CSPC850MULTILINGUAL	850
CSPC862LATINHEBREW	862

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
CSPC8CODEPAGE437	437
CSPCP852	852
CSSHIFTJIS	943
CSUCS4	1236
CSUNICODE11	1204
CSUNICODE	1204
CSUNICODEASCII	1204
CSUNICODELATIN1	1204
CSVISCII	1129
CSWINDOWS31J	943
CYRILLIC	915
DEFAULT	367
EBCDICATDE	273
EBCDICCAFR	500
EBCDICCPAR1	420
EBCDICCPBE	500
EBCDICCPCA	37
EBCDICCPCH	500
EBCDICCPDK	277
EBCDICCPES	284
EBCDICCPFI	278
EBCDICCPFR	297
EBCDICCPGB	285
EBCDICCPGR	423
EBCDICCPHE	424
EBCDICCPIS	871
EBCDICCPIT	280
EBCDICCPNL	37
EBCDICCPNO	277
EBCDICCPROECE	870
EBCDICCPSE	278
EBCDICCPUS	37
EBCDICCPWT	37
EBCDICCPYU	870

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
EBCDICDE273EURO	1141
EBCDICDK277EURO	1142
EBCDICDKNO	277
EBCDICES284EURO	1145
EBCDICES	284
EBCDICFI278EURO	1143
EBCDICFISE	278
EBCDICFR297EURO	1147
EBCDICFR	297
EBCDICGB285EURO	1146
EBCDICINTERNATIONAL500EURO	1148
EBCDICIS871EURO	1149
EBCDICIT280EURO	1144
EBCDICIT	280
EBCDICLATIN9EURO	924
EBCDICNO277EURO	1142
EBCDICPT	37
EBCDICSE278EURO	1143
EBCDICUK	285
EBCDICUS37EURO	1140
EBCDICUS	37
ECMA114	1089
ECMA118	813
ELOT928	813
EUCCN	1383
EUCJP	954
EUCKR	970
EUCTW	964
EXTENDEDUNIXCODEPACKEDFORMATFORJAPANESE	954
GB18030	1392
GB2312	1383
GBK	1386
GREEK8	813
GREEK	813

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
HEBREW	62210
HPROMAN8	1051
IBM00858	858
IBM00924	924
IBM01140	1140
IBM01141	1141
IBM01142	1142
IBM01143	1143
IBM01144	1144
IBM01145	1145
IBM01146	1146
IBM01147	1147
IBM01148	1148
IBM01149	1149
IBM01153	1153
IBM01155	1155
IBM01160	1160
IBM037	37
IBM1026	1026
IBM1043	1043
IBM1047	1047
IBM1252	1252
IBM273	273
IBM277	277
IBM278	278
IBM280	280
IBM284	284
IBM285	285
IBM297	297
IBM367	367
IBM420	420
IBM423	423
IBM424	424
IBM437	437

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
IBM500	500
IBM808	808
IBM813	813
IBM819	819
IBM850	850
IBM852	852
IBM855	855
IBM857	857
IBM862	862
IBM863	863
IBM864	864
IBM866	866
IBM867	867
IBM869	869
IBM870	870
IBM871	871
IBM872	872
IBM902	902
IBM904	904
IBM912	912
IBM915	915
IBM916	916
IBM920	920
IBM921	921
IBM922	922
IBM923	923
IBMTHAI	838
IRV	367
ISO10646	1204
ISO10646UCS2	1200
ISO10646UCS4	1232
ISO10646UCSBASIC	1204
ISO10646UNICODELATIN1	1204
ISO646BASIC1983	367

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
ISO646IRV1983	367
ISO646IRV1991	367
ISO646US	367
ISO885911987	819
ISO885913	901
ISO885915	923
ISO885915FDIS	923
ISO88591	819
ISO885921987	912
ISO88592	912
ISO885951988	915
ISO88595	915
ISO885961987	1089
ISO88596	1089
ISO88596I	1089
ISO885971987	813
ISO88597	813
ISO885981988	62210
ISO88598	62210
ISO88598I	916
ISO885991989	920
ISO88599	920
ISOIR100	819
ISOIR101	912
ISOIR126	813
ISOIR127	1089
ISOIR128	920
ISOIR138	62210
ISOIR144	915
ISOIR146	915
ISOIR147	915
ISOIR148	920
ISOIR149	970
ISOIR2	367

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
ISOIR6	367
JUSIB1003MAC	915
JUSIB1003SERB	915
KOI8	878
KOI8R	878
KOI8U	1168
KOREAN	970
KSC56011987	970
KSC56011989	970
KSC5601	970
L1	819
L2	912
L5	920
L9	923
LATIN0	923
LATIN1	819
LATIN2	912
LATIN5	920
LATIN9	923
MAC	1275
MACEDONIAN	915
MACINTOSH	1275
MICROSOFTPUBLISHING	1004
MS1386	1386
MS932	943
MS936	1386
MS949	970
MSKANJI	943
PCMULTILINGUAL850EURO	858
R8	1051
REF	367
ROMAN8	1051
SERBIAN	915
SHIFTJIS	943

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
SJIS	943
SUNEUGREEK	813
T101G2	920
TIS20	874
TIS620	874
UNICODE11	1204
UNICODE11UTF8	1208
UNICODEBIGUNMARKED	1200
UNICODELITTLEUNMARKED	1202
US	367
USASCII	367
UTF16	1204
UTF16BE	1200
UTF16LE	1202
UTF32	1236
UTF32BE	1232
UTF32LE	1234
UTF8	1208
VISCII	1129
WINDOWS1250	5346
WINDOWS1251	5347
WINDOWS1252	5348
WINDOWS1253	5349
WINDOWS1254	5350
WINDOWS1255	5351
WINDOWS1256	5352
WINDOWS1257	5353
WINDOWS1258	5354
WINDOWS28598	62210
WINDOWS31J	943
WINDOWS936	1386
XEUCTW	964
XMSWIN936	1386
XUTF16BE	1200

Table 31. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
XUTF16LE	1202
XWINDOWS949	970

Mappings of CCSIDs to encoding names for textual XML output data

As part of an implicit or explicit XMLSERIALIZE operation, Db2 might add an encoding declaration at the beginning of textual XML output data.

The encoding declaration has the following form:

```
<?xml version="1.0" encoding="encoding-name"?>
```

In general, the character set identifier in the encoding declaration describes the encoding of the characters in the output string. For example, when XML data is serialized to the CCSID that corresponds to the target application data type, the encoding declaration describes the target application variable CCSID. An exception is the case where the application performs an explicit XMLSERIALIZE function with INCLUDING XMLDECLARATION. When you specify INCLUDING XMLDECLARATION, the database manager generates an encoding declaration for UTF-8. If the target data type is a CLOB or DBCLOB type, additional code page conversion might occur, which can make the encoding information inaccurate. If the data is further parsed in the application, data corruption can result.

Where possible, Db2 chooses the IANA registry name for the CCSID, as prescribed by the XML standard.

If an application has a target CCSID that is not in the following list, Db2 generates an encoding name as follows:

- If the CCSID has one to three digits, the generated encoding name is IBM*nnn*, where *nnn* is the CCSID, which is left-padded with zeroes for a one-digit or two-digit CCSID.
- If the CCSID has four to five digits, the generated encoding name is IBM*nnnnn*, where *nnnnn* is the CCSID, which is left-padded with a zero for a four-digit CCSID.

Some parsers might not be able to parse retrieved documents that have the generated encoding names.

Table 32. CCSIDs and corresponding encoding names

CCSID	Encoding name
37	IBM037
273	IBM273
277	IBM277
278	IBM278
280	IBM280
284	IBM284
285	IBM285
297	IBM297
367	US-ASCII
420	IBM420
423	IBM423
424	IBM424
437	IBM437

Table 32. CCSIDs and corresponding encoding names (continued)

CCSID	Encoding name
500	IBM500
808	IBM808
813	ISO-8859-7
819	ISO-8859-1
838	IBM-Thai
850	IBM850
852	IBM852
855	IBM855
857	IBM857
858	IBM00858
862	IBM862
863	IBM863
864	IBM864
866	IBM866
867	IBM867
869	IBM869
870	IBM870
871	IBM871
872	IBM872
874	TIS-620
878	KOI8-R
901	ISO-8859-13
902	IBM902
904	IBM904
912	ISO-8859-2
915	ISO-8859-5
916	ISO-8859-8-I
920	ISO-8859-9
921	ISO-8859-13
922	IBM922
923	ISO-8859-15
924	IBM00924
932	Shift_JIS
943	Windows-31J

Table 32. CCSIDs and corresponding encoding names (continued)

CCSID	Encoding name
949	EUC-KR
950	Big5
954	EUC-JP
964	EUC-TW
970	EUC-KR
1004	Microsoft-Publish
1026	IBM1026
1043	IBM1043
1047	IBM1047
1051	hp-roman8
1089	ISO-8859-6
1129	VISCII
1140	IBM01140
1141	IBM01141
1142	IBM01142
1143	IBM01143
1144	IBM01144
1145	IBM01145
1146	IBM01146
1147	IBM01147
1148	IBM01148
1149	IBM01149
1153	IBM01153
1155	IBM01155
1160	IBM-Thai
1161	TIS-620
1162	TIS-620
1163	VISCII
1168	KOI8-U
1200	UTF-16
1202	UTF-16
1204	UTF-16
1208	UTF-8
1232	UTF-32

Table 32. CCSIDs and corresponding encoding names (continued)

CCSID	Encoding name
1234	UTF-32
1236	UTF-32
1250	windows-1250
1251	windows-1251
1252	windows-1252
1253	windows-1253
1254	windows-1254
1255	windows-1255
1256	windows-1256
1257	windows-1257
1258	windows-1258
1275	MACINTOSH
1363	KSC_5601
1370	Big5
1381	GB2312
1383	GB2312
1386	GBK
1392	GB18030
4909	ISO-8859-7
5039	Shift_JIS
5346	windows-1250
5347	windows-1251
5348	windows-1252
5349	windows-1253
5350	windows-1254
5351	windows-1255
5352	windows-1256
5353	windows-1257
5354	windows-1258
5488	GB18030
8612	IBM420
8616	IBM424
9005	ISO-8859-7
12712	IBM424

Table 32. CCSIDs and corresponding encoding names (continued)

CCSID	Encoding name
13488	UTF-16
13490	UTF-16
16840	IBM420
17248	IBM864
17584	UTF-16
17586	UTF-16
62209	IBM862
62210	ISO-8859-8
62211	IBM424
62213	IBM862
62215	ISO-8859-8
62218	IBM864
62221	IBM862
62222	ISO-8859-8
62223	windows-1255
62224	IBM420
62225	IBM864
62227	ISO-8859-6
62228	windows-1256
62229	IBM424
62231	IBM862
62232	ISO-8859-8
62233	IBM420
62234	IBM420
62235	IBM424
62237	windows-1255
62238	ISO-8859-8-I
62239	windows-1255
62240	IBM424
62242	IBM862
62243	ISO-8859-8-I
62244	windows-1255
62245	IBM424
62250	IBM420

Related concepts

Encoding scenarios for retrieval of XML data with explicit XMLSERIALIZE

The target encoding scheme and application code page can affect data conversion, truncation, and internal encoding during XML data retrieval with an explicit XMLSERIALIZE invocation.

Chapter 8. Overview of XQuery

XQuery is a functional programming language that was designed by the World Wide Web Consortium (W3C) to meet specific requirements for querying and modifying XML data.

The XQuery language provides several kinds of expressions that can be constructed from keywords, symbols, and operands. In most cases, the operands of various expressions, operators, and functions must conform to the expected types. Db2 ignores type errors in certain situations.

XQuery supports a subset of the language constructs in the W3C recommendation.

XQuery can be used in the following contexts:

- As an argument to the XMLQUERY SQL built-in function, which extracts data from an XML column
- As an argument to the XMLEXISTS SQL predicate, which is used for evaluation of data in an XML column

XQuery expressions

The basic building block of XQuery is the expression. XQuery provides several kinds of expressions for working with XML data:

- Primary expressions, which include the basic primitives of the language, such as literals, variable references, and function calls
- Path expressions for locating nodes within a document tree
- Arithmetic expressions for addition, subtraction, multiplication, division, and modulus
- Comparison expressions for comparing two values
- Logical expressions for using boolean logic

XQuery expressions can be composed with full generality, which means that where an expression is expected, any kind of expression can be used. In general, the operands of an expression are other expressions. In the following example, the operands of a logical expression are the comparison expressions `1 = 1` and `2 = 2`:

```
1 = 1 and 2 = 2
```

XQuery processing

An XQuery expression consists of an optional *prolog* that establishes the processing environment and an *expression* that generates a result. XQuery processing occurs in two phases: the static analysis phase and the dynamic evaluation phase.

During the static analysis phase, the expression is parsed and augmented based on information that is defined in the prolog. The static context is used to resolve type names, function names, and variable names that are used by the expression. The *static context* includes all information that is available prior to evaluating an expression. The static phase occurs during the Db2 BIND process or during PREPARE. If a required name is not found in the static context, an error is raised.

The dynamic evaluation phase occurs if no errors are detected during the static analysis phase. During the dynamic evaluation phase, the value of the expression is computed. A dynamic type is associated with each value as the value is computed. If an operand of an expression has a dynamic type that does not match the expected type, a type error is raised. If the evaluation generates no errors, a result is returned. The *dynamic context* includes information that is available at the time the expression is evaluated.

The result of an XQuery expression is, in general, a heterogeneous sequence of XML nodes and atomic values. More specifically, the result of an XQuery expression is an instance of the XQuery data model.

© Copyright IBM Corp. 2007, 2020

The XPath 2.0 and XQuery 1.0 data model

The XPath 2.0 and XQuery 1.0 data model represents an XML document as a hierarchy (tree) of nodes that represent XML elements and attributes. Each value of the data model is a sequence that can contain zero, one, or more items. The items can be atomic values or nodes. Every XQuery expression takes as its input an instance of the XPath 2.0 and XQuery 1.0 data model and returns an instance of the XPath 2.0 and XQuery 1.0 data model.

XQuery data types

XQuery supports the following data types:

- xs:integer
- xs:decimal
- xs:double
- xs:string
- xs:boolean
- xs:untypedAtomic
- xs:date
- xs:dateTime
- xs:time
- xs:duration
- xs:yearMonthDuration
- xs:dayTimeDuration

Db2 checks data types during the dynamic evaluation phase and the static analysis phase. When an expression encounters an inappropriate type, a type error is raised. For example, an XQuery expression that uses the plus operator (+) to add two strings together results in a type error because the plus operator is used in arithmetic expression to add numeric values only. Implicit type conversions and type substitutions occur, when possible, to provide the type that is expected by an expression.

The built-in function library

XQuery provides a library of built-in functions for working with XML data. The library includes the following types of functions:

- String functions
- Numeric functions
- Date and time functions
- Functions that operate on boolean values
- Functions that operate on sequences

These built-in functions are in the namespace with URI `http://www.w3.org/2005/xpath-functions`, which by default is associated with the prefix `fn`. The default function namespace is set to `fn` by default, which means that you can call functions in this namespace without specifying a prefix.

Function calls can be used anywhere in an XQuery expression where an expression is expected.

Best applications for XQuery or XPath

XPath provides a subset of the XQuery language. You can write better pureXML applications if you understand when it is better to use XQuery and when it is better to use XPath.

When to use XPath

If you need only to identify qualifying documents in an XML column using an XMLEXISTS predicate, XPath is more efficient than XQuery. XPath has the necessary function to identify the documents. In addition, you can use an XML index with an XPath expression in an XMLEXISTS predicate, but you cannot use an XML index with an XQuery expression.

When to use XQuery

Use XQuery when you need constructors or you need to use conditional logic.

XQuery code is more readable than:

- Code that uses XPath with the XMLDOCUMENT, XMLELEMENT, and XMLATTRIBUTES functions to construct documents
- Code that uses XPath with the XMLTABLE, XMLDOCUMENT, XMLELEMENT, and XMLATTRIBUTES, and XMLAGG functions to perform conditional logic

Example: Construction of documents with XPath and XQuery:

Suppose that the PURCHASEORDERS table is defined like this:

```
CREATE TABLE PURCHASEORDERS (
 PONUMBER VARCHAR(10) NOT NULL,
 STATUS VARCHAR(10) NOT NULL WITH DEFAULT 'New',
 XMLPO XML)
```

The PORDER column of the PURCHASEORDER table contains this document, which is associated with PONUMBER '200300001':

```
<purchaseOrder
   xmlns="xmlns="http://posample.org"
   orderDate="2009-12-01">
  <shipTo exportCode="1">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo>
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
```

```
    </item>
  </items>
</purchaseOrder>
```

You need to construct this document:

```
<invoice invoiceNo="12345">
 <name xmlns="http://posample.org">Robert Smith</name>
 <purchaseOrderNo>200300001</purchaseOrderNo>
 <amount>278.94</amount>
</invoice>
```

XPath code to construct the document looks similar to this code:

```
SELECT XMLDOCUMENT(
  XMLELEMENT(NAME "invoice",
   XMLATTRIBUTES( '12345' as "invoiceNo"),
    XMLQUERY('declare default element namespace "http://posample.org";
     /purchaseOrder/billTo/name' PASSING XMLPO),
     XMLELEMENT(NAME “purchaseOrderNo”,
      PONUMBER),
     XMLELEMENT(NAME "amount",
      XMLQUERY('declare default element namespace "http://posample.org";
       fn:sum(/purchaseOrder/items/item/xs:decimal(USPrice))'
       PASSING XMLPO)
      )
  )
 )
 FROM PURCHASEORDERS PO
 WHERE PONUMBER = '200300001'
```

You can construct the same document using the following XQuery code, which is easier to interpret:

```
SELECT XMLQUERY(
 'let $x := /purchaseOrder
   return
    <invoice invoiceNo= "12345">
      {$x/billTo/name}
      <purchaseOrderNo> { $y } </purchaseOrderNo>
      <amount> { fn:sum($x/items/item/xs:decimal(USPrice)) } </amount>
    </invoice>' PASSING XMLPO, PO.PONUMBER as "y")
FROM PurchaseOrders PO
WHERE PONUMBER = '200300001'
```

XML namespaces and qualified names in XQuery

XQuery uses XML namespaces to prevent naming collisions. An *XML namespace* is a collection of names that is identified by a namespace URI. Namespaces provide a way of qualifying names that are used for elements, attributes, data types, and functions in XQuery.

Names in XQuery are called QNames (qualified names) and conform to the syntax that is specified in a recommendation by the World Wide Web Consortium (W3C). A *QName* consists of an optional namespace prefix and a local name. The namespace prefix, if present, is bound to a URI and provides a shortened form of the URI. During query processing, XQuery expands the QName by resolving the URI that is bound to the namespace prefix. The expanded QName includes the namespace URI and a local name. Two QNames are equal if they have the same namespace URI and local name. This means that two QNames can match even if they have different prefixes provided that the prefixes are bound to the same namespace URI.

Using QNames in XQuery allows expressions to refer to element types or attribute names that have the same local name, but might be associated with different DTDs or XML Schemas. In the following XML data, `pfx1` is a prefix that is bound to some URI. `pfx2` is a prefix that is bound to a different URI. `c` is the local name for all three elements:

```
<a xmlns:pfx1="uri1" xmlns:pfx2="uri2">
<b>
 <pfx1:c>C</pfx1:c>
 <pfx2:c>B</pfx2:c>
 <c>A</c>
```

```
</b>
</a>
```

The elements in this example share the same local name, `c`, but naming conflicts do not occur because the elements exist in different namespaces. During expression processing, the name `pfx1:c` is expanded into a name that includes the URI bound to `pfx1` (`uri1`) and the local name, `c`. Likewise, the name `pfx2:c` is expanded into a name that includes the URI bound to `pfx2` (`uri2`) and the local name, `c`. The element `c`, which has an empty prefix, is bound to the default element namespace because no prefix is specified. An error is raised if a name uses a prefix that is not bound to a URI.

The namespace prefix must be an NCName (non-colonized name). An XML NCName is similar to an XML Name except that NCName cannot include a colon.

Some namespaces are predeclared; others can be added through declarations in the XQuery expression prolog. XQuery includes the following predeclared namespace prefixes:

Prefix	URI	Description
xs	http://www.w3.org/2001/XMLSchema	XML Schema namespace
xsi	http://www.w3.org/2001/XMLSchema-instance	XML Schema instance namespace
fn	http://www.w3.org/2005/xpath-functions	Default function namespace
xdt	http://www.w3.org/2005/xpath-datatypes	XQuery type namespace

In addition to the predeclared namespaces, namespaces can be provided in the following ways:

- The following namespace information is available in the static context:
  - *In-scope namespaces* are a set of prefix and URI pairs. The in-scope namespaces are used for resolving prefixes that are used in QNames in an XQuery expression. In-scope namespaces come from the following sources:
    - Namespace declarations in an XQuery expression
    - The XMLNAMESPACES Db2 built-in function in the XMLELEMENT or XMLFOREST Db2 built-in function

      If the XMLQUERY Db2 built-in function is an argument to XMLELEMENT or XMLFOREST, the namespaces that are declared in that XMLELEMENT or XMLFOREST invocation become part of the static context for the XQuery expression in the XMLQUERY invocation.
  - *Default element or type namespace* is the namespace that is used for any unprefixed QName that appears where an element or type name is expected. The initial default element or type namespace is the default namespace that is provided by a `declare default element namespace` clause in the prolog of an XQuery expression.
  - *Default function namespace* is the namespace that is associated with built-in functions: http://www.w3.org/2003/11/xpath-functions. There are no user-defined functions in XQuery.

Related information

Namespaces in XML

Case sensitivity in XQuery

XQuery is a case-sensitive language.

Keywords in XQuery use lowercase characters and are not reserved. Names in XQuery expressions are allowed to be the same as language keywords.

Related concepts

XML namespaces and qualified names in XQuery

XQuery uses XML namespaces to prevent naming collisions. An *XML namespace* is a collection of names that is identified by a namespace URI. Namespaces provide a way of qualifying names that are used for elements, attributes, data types, and functions in XQuery.

Whitespace in XQuery

Whitespace is allowed in most XQuery expressions to improve readability even if whitespace is not part of the syntax for the expression. Whitespace consists of space characters (U+0020), carriage returns (U+000D), line feeds (U+000A), and tabs (U+0009).

In general, whitespace is not significant in an XQuery expression, except in the following situations where whitespace is preserved:

- The whitespace is in a string literal.
- The whitespace clarifies an expression by preventing two adjacent tokens from being mistakenly recognized as one.
- The whitespace is in an element constructor. The boundary-space declaration in the prolog determines whether to preserve or strip whitespace in element constructors.

The following examples include expressions that require whitespace for clarity:

- `one- two` results in a syntax error. The parser recognizes one- as a single QName (qualified name) and raises an error when no operator is found.
- `one -two` does not result in a syntax error. The parser recognizes one as a QName, the minus sign (-) as an operator, and then `two` as another QName.
- `one-two` does not result in a syntax error. However, the expression parses as a single QName because a hyphen (-) is a valid character in a QName.
- The following expressions all result in syntax errors:
  - `5 div2`
  - `5div2`

  In these expressions, whitespace is required for the parser to recognize each token separately. Notice that `5div 2` does not result in a syntax error.

Comments in XQuery

Comments are allowed in an XQuery expression, wherever nonessential whitespace is allowed. Comments do not affect expression processing.

A comment is a string that is delimited by the symbols `(:` and `:)`. The following example is a comment in XQuery:

```
(: This is a comment. It makes code easier to understand. :)
```

The following general rules apply to using comments in XQuery:

- Comments can be used wherever nonessential whitespace is allowed. *Nonessential whitespace* is whitespace that is not part of the syntax of an XQuery expression.
- Comments can nest within each other, but each nested comment must have open and close delimiters, `(:` and `:)`.

The following examples illustrate legal comments and comments that result in errors:

- `(: is this a comment? ::)` is a legal comment.
- `(: is this a comment? ::) or an error? :)` results in an error because there is an unbalanced nesting of the symbols `(:` and `:)`.
- `(: commenting out a (: comment :) may be confusing, but often helpful :)` is a legal comment because a balanced nesting of comments is allowed.
- `"this is just a string :)"` is a legal expression.

- `(: "this is just a string :)" :)` results in a syntax error. Likewise, `"this is another string (:"` is a legal expression, but `(: "this is another string (:" :)` results in a syntax error. Literal content can result in an unbalanced nesting of comments.

Chapter 9. XQuery type system

XQuery is a strongly typed language in which the operands of various expressions, operators, and functions conform to expected types.

The type system for XQuery includes a subset of the built-in types of XML schema and the predefined types of XQuery.

The built-in types of XML Schema are in the namespace `http://www.w3.org/2001/XMLSchema`, which has the predeclared namespace prefix `xs`. Some examples of built-in schema types include `xs:integer` and `xs:string`.

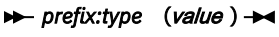
Overview of the type system

The type system for XQuery includes simple atomic types and complex types. A *simple atomic type* is a primitive or derived atomic type that does not contain elements or attributes. A *complex type* can contain mixed content or element-only content.

Constructor functions for built-in data types

Every built-in atomic type that is defined in the XML Schema Definition language has an associated constructor function.

Syntax



prefix
The prefix that is bound to the namespace for the data type. This is not the prefix that is bound to the default function namespace.

type
The unqualified name of the target data type.

value
The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

If *value* is illegal for the target data type, the constructor function returns an error.

Returned value

If *value* is not the empty sequence, the returned value is an instance of *prefix:type*.

If *value* is the empty sequence, a constructor function returns the empty sequence.

Example:

The constructor function `xs:integer(100)` or the constructor function `xs:integer("100")` returns the `xs:integer` value 100. A constructor function whose argument is a node with the typed value 100 also returns the typed value 100.

Related reference

xs:date
The date type xs:date represents an interval of exactly one day that begins on the first moment of a given day.

xs:dateTime
The data type xs:dateTime represents an instant in time.

xs:dayTimeDuration

© Copyright IBM Corp. 2007, 2020

The data type xs:dayTimeDuration represents a duration of time that is expressed by days, hours, minutes, and seconds components. xs:dayTimeDuration is derived from data type xs:duration.

xs:decimal
The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

xs:double
The data type `xs:double` is supported in XQuery by the IEEE 64-bit decimal floating point.

xs:duration
The data type xs:duration represents a duration of time that is expressed by the Gregorian year, month, day, hour, minute, and second components. xs:duration is derived from data type xs:anyAtomicType.

xs:integer
The data type `xs:integer` represents a decimal number that does not include a trailing decimal point. The base type of `xs:integer` is `xs:decimal`.

xs:string
The data type `xs:string` represents character strings in XML. Because `xs:string` is a simple type, it cannot contain any children.

xs:time
The data type xs:time represents an instant of time that recurs every day.

xs:untypedAtomic
The data type `xs:untypedAtomic` serves as a special type annotation to indicate atomic values that have not been validated by an XML schema or a DTD.

xs:yearMonthDuration
The data type xs:yearMonthDuration represents a duration of time that is expressed by the Gregorian year and month components. xs:yearMonthDuration is derived from data type xs:duration.

Generic data types

Generic data types support data that is not strongly typed.

xs:anyType

The data type `xs:anyType` is the base type for all data types that are defined in the XML Schema Definition language.

Related concepts

Data model generation in XQuery
Before an XQuery expression can be processed, the input documents must be represented in the pureXML data model.

Related reference

xs:anySimpleType
The data type `xs:anySimpleType` is the base type for all primitive types that are defined in the XML Schema Definition language.

xs:untyped
The data type `xs:untyped` serves as a special type annotation to indicate types that have not been validated by an XML schema or a DTD. The data type `xs:untyped` can be used (for example, in a function signature) to define a required type to indicate that only an untyped value is acceptable. The base type of `xs:untyped` is `xs:anyType`.

xs:anySimpleType

The data type `xs:anySimpleType` is the base type for all primitive types that are defined in the XML Schema Definition language.

`xs:anySimpleType` is used to define a required type (for example, in a function signature) to indicate that any simple type is acceptable. The base type of `xs:anySimpleType` is `xs:anyType`.

Casting is not supported to or from `xs:anySimpleType`.

Lexical form
`xs:anySimpleType` can have any lexical form.
Related concepts
Data model generation in XQuery
Before an XQuery expression can be processed, the input documents must be represented in the pureXML data model.

xs:anyAtomicType

The data type `xs:anyAtomicType` is the base type for all primitive atomic types that are defined in the XML Schema Definition language.

Lexical form
The data type `xs:anyAtomicType` can be used to define a required type (for example, in a function signature) to indicate that any of the primitive atomic types or `xs:untypedAtomic` is acceptable. The base type of `xs:anyAtomicType` is `xs:anySimpleType`.

`xs:anyAtomicType` can have any lexical form.

Related reference
xs:untypedAtomic
The data type `xs:untypedAtomic` serves as a special type annotation to indicate atomic values that have not been validated by an XML schema or a DTD.

Data types for untyped data

The xs:untyped and xs:untypedAtomic data types support untyped data.

xs:untyped

The data type `xs:untyped` serves as a special type annotation to indicate types that have not been validated by an XML schema or a DTD. The data type `xs:untyped` can be used (for example, in a function signature) to define a required type to indicate that only an untyped value is acceptable. The base type of `xs:untyped` is `xs:anyType`.

If an element node is annotated as `xs:untyped`, all of its descendant element nodes are also annotated as `xs:untyped`

Related reference
xs:anyType
The data type `xs:anyType` is the base type for all data types that are defined in the XML Schema Definition language.

xs:untypedAtomic

The data type `xs:untypedAtomic` serves as a special type annotation to indicate atomic values that have not been validated by an XML schema or a DTD.

An attribute that has an unknown type is represented in the data model by an attribute node with the type `xs:untypedAtomic`. The data type `xs:untypedAtomic` can be used (for example, in a function signature) to define a required type to indicate that only an untyped atomic value is acceptable. The base type of `xs:untypedAtomic` is `xs:anyAtomicType`.

Lexical form

`xs:untypedAtomic` can have any lexical form.

Constructor

Use the following syntax to construct an instance of `xs:untypedAtomic`:



value
The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

If *value* is illegal for the target data type, the constructor function returns an error.

Related reference

xs:anyAtomicType
The data type `xs:anyAtomicType` is the base type for all primitive atomic types that are defined in the XML Schema Definition language.

xs:string

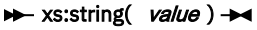
The data type `xs:string` represents character strings in XML. Because `xs:string` is a simple type, it cannot contain any children.

Lexical form

The lexical form of `xs:string` is a sequence of characters that can include any character that is in the range of legal characters for XML.

Constructor

Use the following syntax to construct an instance of `xs:string`:



value
The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

If *value* is illegal for the target data type, the constructor function returns an error.

Numeric data types

The xs:decimal, xs:double, and xs:integer data types support numeric data.

xs:decimal

The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

Lexical form

The lexical form of `xs:decimal` is a finite-length sequence of decimal digits (0 to 9) that are separated by a period as a decimal indicator. An optional leading sign is allowed. If the sign is omitted, a positive sign (+) is assumed. Leading and trailing zeroes are optional. If the fractional part is zero, the period and any following zeroes can be omitted. The following numbers are all valid examples of a decimal: `-1.23, 12678967.543233, +100000.00, 210.`.

Constructor

Use the following syntax to construct an instance of `xs:decimal`:



value
The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

If *value* is illegal for the target data type, the constructor function returns an error.

xs:double

The data type `xs:double` is supported in XQuery by the IEEE 64-bit decimal floating point.

Lexical form

The lexical form of `xs:double` is a mantissa followed, optionally, by the character E or e, followed by an exponent. The exponent must be an integer. The mantissa must be a decimal number. The representations for exponent and mantissa must follow the lexical rules for `xs:integer` and `xs:decimal`. If the E or e and the exponent that follows are omitted, an exponent value of 0 is assumed.

The special values positive infinity, negative infinity, and not-a-number have the lexical representations INF, -INF and NaN, respectively. Lexical representations for zero can take a positive or negative sign. The following literals are all valid examples of a double: `-1E4, 1267.43233E12, 12.78e-2, 12 , -0, 0 and INF`.

Constructor

Use the following syntax to construct an instance of `xs:double`:



value
The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

If *value* is illegal for the target data type, the constructor function returns an error.

Related reference

xs:decimal
The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

xs:integer
The data type `xs:integer` represents a decimal number that does not include a trailing decimal point. The base type of `xs:integer` is `xs:decimal`.

xs:integer

The data type `xs:integer` represents a decimal number that does not include a trailing decimal point. The base type of `xs:integer` is `xs:decimal`.

Lexical form

The lexical form of `xs:integer` is a finite-length sequence of decimal digits (0 to 9) with an optional leading sign. If the sign is omitted, a positive sign (+) is assumed. The following numbers are all valid examples of integers: `-1, 0, 12678967543233, +100000`.

Constructor

Use the following syntax to construct an instance of `xs:integer`:



value
The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

If *value* is illegal for the target data type, the constructor function returns an error.

Related reference

xs:decimal
The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

Range limits for numeric types

XQuery has range limits for numeric data types.

The following table lists the range limit and SQL equivalent for each XQuery numeric data type.

Table 33. Range limits for numeric types

XML type	Db2 XML range	SQL type mapping
xs:double	34 digits of precision and an exponent range of 10**-6143 to 10**+6144	DECFLOAT
xs:decimal	Up to 34 digits of precision, and a range of 1-10**34 to 10**34 -1	DECIMAL
xs:integer	-9223372036854775808 to 9223372036854775807	BIGINT

Related reference

xs:decimal
The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

xs:double
The data type `xs:double` is supported in XQuery by the IEEE 64-bit decimal floating point.

xs:integer
The data type `xs:integer` represents a decimal number that does not include a trailing decimal point. The base type of `xs:integer` is `xs:decimal`.

xs:boolean

The data type `xs:boolean` supports the mathematical concept of binary-valued logic: true or false.

Lexical form

The lexical form of the data type `xs:boolean` can be one of the literal values true, false, 1, or 0.

Constructor

Use the following syntax to construct an instance of `xs:boolean`:

value
The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

If *value* is illegal for the target data type, the constructor function returns an error.

Date and time data types

The xs:date, xs:time, and xs:dateTime data types support date and time data.

xs:date

The date type xs:date represents an interval of exactly one day that begins on the first moment of a given day.

Lexical form

The lexical form of xs:date is a finite-length sequence of characters of the following form: *yyyy-mm-ddzzzzzz*. The following abbreviations describe this form:

yyyy
A four-digit numeral that represents the year.

The value cannot begin with a negative (-) sign or a plus (+) sign.

0001 is the lexical representation of the year 1 of the Common Era (also known as 1 AD).

The value cannot be 0000.

-
Separators between parts of the date.

mm
A two-digit numeral that represents the month.

dd
A two-digit numeral that represents the day.

zzzzzz
Optional. If present, represents the time zone.

Timezone indicator

The lexical form for the time zone indicator is a string that includes one of the following forms:

- A positive (+) or negative (-) sign that is followed by hh`:`mm, where the following abbreviations are used:

  hh
  A two-digit numeral (with leading zeros as required) that represents the hours. The value must be between -14 and +14, inclusive.

  mm
  A two-digit numeral that represents the minutes. The value of the minutes property must be zero when the hours property is equal to 14.

  +
  Indicates that the specified time instant is in a time zone that is ahead of the UTC time by *hh* hours and *mm* minutes.

  -
  Indicates that the specified time instant is in a time zone that is behind UTC time by *hh* hours and *mm* minutes.
- The literal Z, which represents the time in UTC (Z represents Zulu time, which is equivalent to UTC). Specifying Z for the time zone is equivalent to specifying `+00:00` or `-00:00`.

Example

The following form indicates 10 October 2009, Eastern Standard Time in the United States:

```
2009-10-10-05:00
```

This date is expressed in UTC as `2009-10-10T05:00:00Z`..

xs:dateTime

The data type xs:dateTime represents an instant in time.

The xs:dateTime data type has several properties. The following properties:

- year
- month
- day
- hour
- minute
- second
- time zone (optional)

The year, month, day, hour, and minute properties are expressed as integer values. The seconds property is expressed as a decimal value. The time zone property is expressed as a time zone indicator.

Lexical form

The lexical form of xs:dateTime is a finite-length sequence of characters of the following form: *yyyy*-*mm*-*dd*T*hh*:*mm*:*ss*.*sssssssssssszzzzzz*. The following abbreviations describe this form:

yyyy
A four-digit numeral that represents the year.

The value cannot begin with a negative (-) sign or a plus (+) sign.

0001 is the lexical representation of the year 1 of the Common Era (also known as 1 AD).

The value cannot be 0000.

-
Separators between parts of the date portion

mm
A two-digit numeral that represents the month.

dd
A two-digit numeral that represents the day.

T
A separator to indicate that the time of day follows.

hh
A two-digit numeral (with leading zeros as required) that represents the hours. The value must be between -14 and +14, inclusive.

:
A separator between parts of the time portion.

mm
A two-digit numeral that represents the minute.

ss
A two-digit numeral that represents the whole seconds.

.sssssssssss
Optional. If present, a 1-to-12 digit numeral that represents the fractional seconds.

zzzzzz
Optional. If present, represents the time zone offset from UTC (Coordinated Universal Time). If not specified, an implicit time of UTC (Coordinated Universal Time) is used.

Each part of the datetime value that is expressed as a numeric value is constrained to the maximum value within the interval that is determined by the next-higher part of the datetime value. For example, the day value can never be 32 and cannot be 29 for month 02 and year 2002 (February 2002).

Timezone indicator

The lexical form for the time zone indicator is a string that includes one of the following forms:

- A positive (+) or negative (-) sign that is followed by hh:mm, where the following abbreviations are used:

  hh
  A two-digit numeral (with leading zeros as required) that represents the hours. The value must be between -14 and +14, inclusive.

  mm
  A two-digit numeral that represents the minutes. The value of the minutes property must be zero when the hours property is equal to 14.

  +
  Indicates that the specified time instant is in a time zone that is ahead of the UTC time by *hh* hours and *mm* minutes.

  -
  Indicates that the specified time instant is in a time zone that is behind UTC time by *hh* hours and *mm* minutes.
- The literal Z, which represents the time in UTC (Z represents Zulu time, which is equivalent to UTC). Specifying Z for the time zone is equivalent to specifying +00:00 or -00:00.

Example

The following form indicates noon on 10 October 2009, Eastern Standard Time in the United States:

```
2009-10-10T12:00:00-05:00
```

This time is expressed in UTC as 2009-10-10T17:00:00Z.

xs:dayTimeDuration

The data type xs:dayTimeDuration represents a duration of time that is expressed by days, hours, minutes, and seconds components. xs:dayTimeDuration is derived from data type xs:duration.

The range that can be represented by this data type is from -P11574074073DT23H163M219.999999999999S to P11574074073DT23H163M219.999999999999S (or -999999999999999.999999999999 seconds to 999999999999999.999999999999 seconds).

The lexical form of xs:dayTimeDuration is P*n*DT*n*H*n*M*n*S, which is a reduced form of the ISO 8601 format. The following abbreviations describe this form:

P
The duration designator.

***n*D**
n is an unsigned integer that represents the number of days.

T
The date and time separator.

***n*H**
n is an unsigned integer that represents the number of hours.

***n*M**
n is an unsigned integer that represents the number of minutes.

***n*S**
n is an unsigned decimal that represents the number of seconds. If a decimal point appears, it must be followed by one to twelve digits that represent fractional seconds.

For example, the following form indicates a duration of 3 days, 10 hours, and 30 minutes:

```
P3DT10H30M
```

The following form indicates a duration of negative 120 days:

```
-P120D
```

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:

- If the number of days, hours, minutes, or seconds in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator must be present.
- The seconds part can have a decimal fraction.
- The designator T must be absent if and only if all of the time items are absent. The designator P must always be present.

For example, the following forms are allowed:

```
P13D
PT47H
P3DT2H
-PT35.89S
P4DT251M
```

The form `P-134D` is not allowed, but the form `-P1347D` is allowed.

Db2 database system stores xs:dayTimeDuration values in a normalized form. In the normalized form, the seconds and minutes components are less than 60, and the hours component is less than 24. Db2 converts each multiple of 60 seconds to one minute, each multiple of 60 minutes to one hour, and each multiple of 24 hours to one day. For example, the following XQuery expression invokes a constructor function specifying a dayTimeDuration of 63 days, 55 hours, and 81 seconds:

```
xs:dayTimeDuration("P63DT55H81S")
```

Db2 converts 55 hours to 2 days and 7 hours, and 81 seconds to 1 minute and 21 seconds. The expression returns the normalized dayTimeDuration value P65DT7H1M21S.

xs:duration

The data type xs:duration represents a duration of time that is expressed by the Gregorian year, month, day, hour, minute, and second components. xs:duration is derived from data type xs:anyAtomicType.

The range that can be represented by this data type is from -P83333333333333Y3M11574074074DT1H46M39.999999999999S to P83333333333333Y3M11574074074DT1H46M39.999999999999S (or -999999999999999 months and -999999999999999.999999999999 seconds to 999999999999999 months and 999999999999999.999999999999 seconds).

The lexical form of xs:duration is the ISO 8601 extended format P*n*Y*n*M*n*DT*n*H*n*M*n*S. The following abbreviations describe the extended format:

P
The duration designator.

***n*Y**
n is an unsigned integer that represents the number of years.

***n*M**
n is an unsigned integer that represents the number of months.

***n*D**
n is an unsigned integer that represents the number of days.

T
The date and time separator.

***n*H**
n is an unsigned integer that represents the number of hours.

***n*M**
n is an unsigned integer that represents the number of minutes.

***n*S**
n is an unsigned decimal that represents the number of seconds. If a decimal point appears, it must be followed by one to twelve digits that represent fractional seconds.

For example, the following form indicates a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes:

```
P1Y2M3DT10H30M
```

The following form indicates a duration of negative 120 days:

```
-P120D
```

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:

- If the number of years, months, days, hours, minutes, or seconds in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator must be present.
- The seconds part can have a decimal fraction.
- The designator T must be absent if and only if all of the time items are absent.
- The designator P must always be present.

For example, the following forms are allowed:

```
P1347Y
P1347M
P1Y2MT2H
P0Y1347M
P0Y1347M0D
```

The form P1Y2MT is not allowed because no time items are present. The form P-1347M is not allowed, but the form -P1347M is allowed.

The Db2 database system stores xs:duration values in a normalized form. In the normalized form, the seconds and minutes components are less than 60, the hours component is less than 24, and the months component is less than 12. Db2 converts each multiple of 60 seconds to one minute, each multiple of 60 minutes to one hour, each multiple of 24 hours to one day, and each multiple of 12 months to one year. For example, the following XQuery expression invokes a constructor function that specifies a duration of 2 months, 63 days, 55 hours, and 91 minutes:

```
xs:duration("P2M63DT55H91M")
```

Db2 converts 55 hours to 2 days and 7 hours, and 91 minutes to 1 hour and 31 minutes. The expression returns the normalized duration value P2M65DT8H31M.

xs:time

The data type xs:time represents an instant of time that recurs every day.

Lexical form

The lexical form of the data type xs:time is *hh*:*mm*:*ss.sssssssssssszzzzzz*.

The following abbreviations describe this form:

hh
A two-digit numeral (with leading zeros as required) that represents the hours.

:
A separator between parts of the time portion.

mm
A two-digit numeral that represents the minute.

ss
A two-digit numeral that represents the whole seconds.

.sssssssssss
Optional. If present, a 1-to-12 digit numeral that represents the fractional seconds.

zzzzzz
Optional. If present, represents the time zone.

Example

The following form, which includes an optional time zone indicator, represents 1:20 p.m. Eastern Standard Time, which is five hours behind than Coordinated Universal Time (UTC):

```
13:20:00-05:00
```

xs:yearMonthDuration

The data type xs:yearMonthDuration represents a duration of time that is expressed by the Gregorian year and month components. xs:yearMonthDuration is derived from data type xs:duration.

The range that can be represented by this data type is from -P83333333333333Y3M to P83333333333333Y3M (or -999999999999999 to 999999999999999 months).

The lexical form of xs:yearMonthDuration is P*n*Y*n*M, which is a reduced form of the ISO 8601 format. The following abbreviations describe this form:

P
The duration designator.

***n*Y**
n is an unsigned integer that represents the number of years.

***n*M**
n is an unsigned integer that represents the number of months.

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

For example, the following form indicates a duration of 1 year and 2 months:

```
P1Y2M
```

The following form indicates a duration of negative 13 months:

```
-P13M
```

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:

- The designator P must always be present.
- If the number of years or months in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator (Y or M) must be present.

For example, the following forms are allowed:

```
P1347Y
P1347M
```

The form `P-1347M` is not allowed, but the form `-P1347M` is allowed. The form `P24YM` is not allowed because `M` must have one preceding digit. `PY43M` is not allowed because `Y` must have at least one preceding digit.

Db2 stores xs:yearMonthDuration values in a normalized form. In the normalized form, the months component is less than 12. Db2 converts each multiple of 12 months to one year. For example, the following XQuery expression invokes a constructor function that specifies a yearMonthDuration of 20 years and 30 months:

```
xs:yearMonthDuration("P20Y30M")
```

Db2 converts 30 months to 2 years and 6 months. The expression returns the normalized yearMonthDuration value P22Y6M.

Casts between XML schema data types

You can use data type constructor functions to cast a value to a specific data type. Specify the value that you want to cast and the type to which you want to cast it.

The following table lists the compatible types for casting. You can cast values only of the listed input types to each target type.

Table 34. Compatible types for casting

Target type	Source type	Comments
xs:untypedAtomic	Any type	

Table 34. Compatible types for casting (continued)

Target type	Source type	Comments
xs:string	Any type	• If the source type is xs:boolean, the result is `true` or `false`. • If the source type is xs:integer, the result is the canonical lexical representation of the value, as defined in the XML Schema specification. • If the source type is xs:decimal: – If the value has no significant digits after the decimal point, the decimal point and the zeroes that follow the decimal point are deleted, and the rules for casting from xs:integer apply. – Otherwise, the result is the canonical lexical representation of the value, as defined in the XML Schema specification. • If the source type is xs:double: – If .000001<=*value*<=1000000, the value is converted to xs:decimal, and the rules for casting from xs:decimal apply. – If *value*=+0, or *value*=-0, the result is '0'. – Otherwise, the result is the canonical lexical representation of the value, as defined in the XML Schema specification. • If the source type is xs:yearMonthDuration or xs:dayTimeDuration, the result is the canonical lexical representation of the value. • If the source type is xs:date, xs:dateTime, or xs:time, the result is the lexical representation of the value, with no adjustment for the time zone. If the value has no time zone, the result has no time zone. If the time zone is +00:00 or -00:00, the result has the UTC time zone "Z".
xs:boolean	xs:untypedAtomic, xs:string, xs:boolean, xs:double, xs:decimal, xs:integer	• If the source type is numeric, a value of 0 or NaN is cast to type xs:boolean with a value of false. • If the source type is xs:string or xs:untypedAtomic, the value "true" and the value "1" are cast to the xs:boolean value `true`. The value "false" and the value "0" are cast to the xs:boolean value `false`. All other values are invalid, and result in an error.
xs:dayTimeDuration	xs:duration, xs:untypedAtomic, xs:string	A cast from xs:duration to xs:dayTimeDuration results in information loss. To avoid information loss, cast the xs:duration value to an xs:yearMonthDuration value and an xs:dayTimeDuration value and work with both values.

Table 34. Compatible types for casting (continued)

Target type	Source type	Comments
xs:decimal	Numeric types, xs:untypedAtomic, xs:string, xs:boolean	Values of numeric types are converted to a value that is within the set of possible values for type xs:decimal and is numerically closest to the source. If two values are equally close, the one that is closest to zero is chosen. The source value cannot be +INF, -INF, NaN, or outside of the range of type xs:decimal. For values of type xs:boolean, true is converted to 1.0, and false is converted to 0.0.
xs:double	Numeric types, xs:untypedAtomic, xs:string, xs:boolean	If the source is of type xs:decimal, or xs:integer, the cast is performed as xs:double(*SV* cast as xs:string) where *SV* is the source value. If the source is of type xs:boolean, true is cast to a value of 1.0E0, and false is cast to a value of 0.0E0.
xs:duration	xs:dayTimeDuration, xs:yearMonthDuration, xs:untypedAtomic, xs:string	• If the source type is xs:dayTimeDuration, the target value has the same days, hours, minutes and seconds components as the source value. The year component and the month component of the target value are 0. • If the source type is xs:yearMonthDuration, the target value has the same years and months components as the source value. The days, hours, minutes and seconds components are 0.
xs:integer	Numeric types, xs:untypedAtomic, xs:string, xs:boolean	If the source type is a numeric type other than integer, the result is the source value with the fractional part discarded. The source cannot be outside of the range of type xs:integer. For values of type xs:boolean, true is converted to 1, and false is converted to 0.
xs:date	xs:dateTime, xs:untypedAtomic, xs:string	The time portion of the source value is not used in the conversion.
xs:time	xs:dateTime, xs:untypedAtomic, xs:string	The date portion of the source value is not used in the conversion.
xs:dateTime	xs:date, xs:untypedAtomic, xs:string	The time portion of the target value is the first moment of the day. The value is not adjustment for the time zone.
xs:yearMonthDuration	xs:duration, xs:untypedAtomic, xs:string	A cast from xs:duration to xs:yearMonthDuration results in information loss. To avoid information loss, cast the xs:duration value to an xs:yearMonthDuration value and an xs:dayTimeDuration value and work with both values.

Example

The following XQuery expression returns purchase orders that contain more than one item. The xs:integer constructor function casts the value of the quantity element to an integer. That integer can then be compared to the integer 1.

```
declare namespace ipo="http://www.example.com/IPO";
/ipo:purchaseOrder[items/item/quantity/xs:integer(.) > 1]
```

If the result of the path expression is not explicitly cast to an integer, Db2 converts both operands to type xs:double to make the numeric comparison. The cast ensures that the values are compared as values of type xs:integer.

Related reference

Castable expressions
Castable expressions test whether a value can be cast to a specific data type. If the value can be cast to the data type, the castable expression returns true. Otherwise, the expression returns false.

xs:date
The date type xs:date represents an interval of exactly one day that begins on the first moment of a given day.

xs:dateTime
The data type xs:dateTime represents an instant in time.

xs:decimal
The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

xs:double
The data type `xs:double` is supported in XQuery by the IEEE 64-bit decimal floating point.

xs:integer
The data type `xs:integer` represents a decimal number that does not include a trailing decimal point. The base type of `xs:integer` is `xs:decimal`.

xs:string
The data type `xs:string` represents character strings in XML. Because `xs:string` is a simple type, it cannot contain any children.

xs:time
The data type xs:time represents an instant of time that recurs every day.

xs:untypedAtomic
The data type `xs:untypedAtomic` serves as a special type annotation to indicate atomic values that have not been validated by an XML schema or a DTD.

Chapter 10. XQuery prologs and expressions

In XQuery, an XQuery expression consists of an optional prolog that is followed by an expression. The *prolog* contains a series of declarations that define the processing environment for the expression. The *path expression* consists of an expression that defines the result of the XQuery expression.

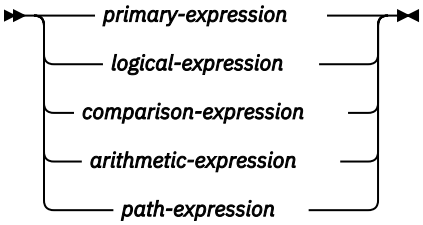
Syntax

The following diagrams show the general format of an XQuery expression.

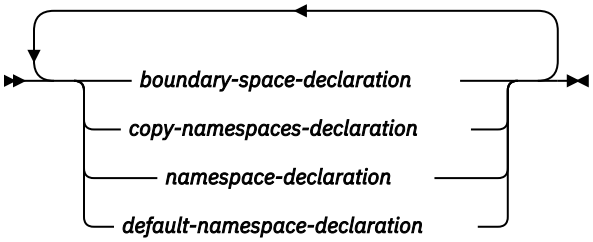
XQuery expression



expression



prolog



Example

The following example illustrates the structure of a typical expression in XQuery. In this example, the prolog contains a namespace declaration that binds the prefix ns1 to a URI. The expression body contains an expression that returns all documents in the XMLPO column where the name attribute on the shipTo node is "Jane" and the name attribute on the billTo node is "Jason":

```
SELECT XMLQUERY ('declare namespace ipo="http://www.example.com/IPO";
/ipo:purchaseOrder[shipTo/name = "Jane" and
billTo/name = "Jason"]'
PASSING XMLPO)
FROM T1;
```

Figure 9. Structure of a typical expression in XQuery

Prologs

The *prolog* consists of a declaration that defines the processing environment for an XQuery expression. A declaration in the prolog is followed by a semicolon (;). The prolog is an optional part of the XQuery expression.

The prolog can contain zero or one boundary space declaration, zero or one copy namespaces declaration, zero or more namespace declarations, and zero or one default namespace declaration.

© Copyright IBM Corp. 2007, 2020

Related concepts

XML namespaces and qualified names in XQuery
XQuery uses XML namespaces to prevent naming collisions. An *XML namespace* is a collection of names that is identified by a namespace URI. Namespaces provide a way of qualifying names that are used for elements, attributes, data types, and functions in XQuery.

Related reference

Default namespace declarations
Default namespace declarations are optional declarations in the XQuery expression prolog that specify the namespaces to use for unprefixed QNames (qualified names).

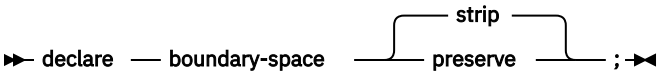
Boundary-space declaration

A boundary-space declaration in the query prolog sets the boundary-space policy for the query. The *boundary-space policy* controls how boundary whitespace is processed by element constructors. *Boundary whitespace* includes all whitespace characters that occur by themselves in the boundaries between tags or enclosed expressions in element constructors.

The boundary-space policy can specify that boundary whitespace is either preserved or stripped (removed) when elements are constructed. If no boundary-space declaration is specified, the default behavior is to strip boundary whitespace when elements are constructed.

The prolog can contain only one boundary-space declaration for a query.

Syntax
boundary-space-declaration



strip
Specifies that boundary whitespace is removed when elements are constructed.

preserve
Specifies that boundary whitespace is preserved when elements are constructed.

Example

The following boundary-space declaration specifies that boundary whitespace is preserved when elements are constructed:

```
declare boundary-space preserve;
```

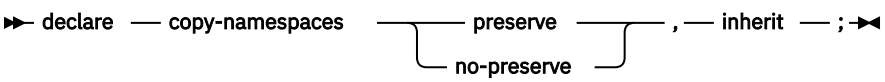
Copy-namespaces declaration

The copy-namespaces declaration in the query prolog sets the policy for the query. The copy-namespaces policy controls how the namespace bindings are assigned when an existing element node is copied by an element constructor or document node constructor.

The setting **preserve** specifies that all in-scope-namespaces of the original element are retained in the new copy. The copied node preserves its default namespace or absence of a default namespace. The setting **no-preserve** specifies that unreferenced in-scope-namespaces of the original element are not retained. The setting **inherit** specifies that the copied node inherits in-scope namespaces from the constructed node. In case of a conflict, the namespace bindings that were preserved from the original node take precedence.

A copy-namespaces declaration that specifies values other than **preserve** or **no-preserve** and **inherit** results in an error. The prolog can contain only one copy-namespaces declaration for a query.

Syntax



preserve
Specifies that all in-scope namespaces of the original element are retained in the new copy.

no-preserve
Specifies that unreferenced in-scope namespaces of the original element are not retained in the new copy.

inherit
Specifies that the copied node inherits in-scope namespaces from the constructed node.

Example

The following copy-namespace declaration is valid, but does not change the default behavior for element construction:

```
declare copy-namespaces preserve, inherit;
```

Namespace declarations

A *namespace declaration* is an optional declaration in the XQuery expression prolog that declares a namespace prefix and associates the prefix with a namespace URI.

The declaration adds the prefix-URI pair to the set of statically known namespaces for the expression. The *statically known namespaces* include all of the namespaces that are known during the static processing of an expression. The namespace declaration is in scope throughout the XQuery expression in which it is declared. Multiple declarations of the same namespace prefix in the query prolog result in an error.

Restriction: The prefixes `xmlns` and `xml` are reserved and cannot be specified as a prefix in a namespace declaration.

Syntax
namespace-declaration

prefix
Specifies a namespace prefix that is bound to the URI. The namespace prefix is used in qualified names (QNames) to identify the namespace for an element, attribute, data type, or function.

stringLiteral
Specifies a string literal that represents the URI to which the prefix is bound. The string literal value must be a valid URI and cannot be a zero-length string.

You can override predeclared namespace prefixes by specifying a namespace declarations for those prefixes. However, you cannot override the URI that is associated with the prefix `xml`.

The string literal cannot be http://www.w3.org/XML/1998/namespace or http://www.w3.org/2000/xmlns/.

Example

The following namespace declaration declares the namespace prefix `ns1` and associates it with the namespace URI `http://posample.org`:

```
declare namespace ns1 = "http://posample.org";
/ns1:purchaseOrder[shipTo/name = "Jane" and billTo/name = "Jason"]
```

When the expression in the example executes, the namespace prefix ns1 is associated with the namespace URI http://posample.org. The instance of the purchase order document to which the expression refers is the instance with the namespace URI `http://posample.org`.

Related concepts

XML namespaces and qualified names in XQuery
XQuery uses XML namespaces to prevent naming collisions. An *XML namespace* is a collection of names that is identified by a namespace URI. Namespaces provide a way of qualifying names that are used for elements, attributes, data types, and functions in XQuery.

Default namespace declarations

Default namespace declarations are optional declarations in the XQuery expression prolog that specify the namespaces to use for unprefixed QNames (qualified names).

An XQuery expression prolog can include a default element namespace declaration.

The default element namespace declaration specifies a namespace URI that is used for unprefixed element names. The XQuery expression prolog can contain one default element namespace declaration only. This declaration is in scope throughout the expression in which it is declared. If no default element namespace is declared, unqualified element names are in no namespace.

Syntax
default-namespace-declaration



namespace
Specifies a string literal that represents the URI for the namespace. The string literal must be a valid URI or a zero-length string.

If *namespace* is a zero-length string, unprefixed element names are in no namespace.

The string literal cannot be http://www.w3.org/XML/1998/namespace or http://www.w3.org/2000/xmlns/.

Example

The following declaration specifies that the default namespace for element names is the namespace that is associated with the URI `http://posample.org`

```
declare default element namespace "http://posample.org";
```

When the query in the example executes, all element nodes in this expression (purchaseOrder, shipTo, billTo, and name) are associated with the namespace URI `http://posample.org`.

```
declare default element namespace "http://posample.org";
/purchaseOrder[shipTo/name = "Jane" and billTo/name = "Jason"]
```

When the expression in the example executes, the namespace URI http://posample.org is associated with all unprefixed element names in the expression.

Related concepts

XML namespaces and qualified names in XQuery

XQuery uses XML namespaces to prevent naming collisions. An *XML namespace* is a collection of names that is identified by a namespace URI. Namespaces provide a way of qualifying names that are used for elements, attributes, data types, and functions in XQuery.

Expressions

XQuery supports several kinds of expressions for working with XML data.

Expression evaluation and processing

A number of operations are often included in the processing of expressions. These operations include extracting atomic values from nodes and using type promotion and subtype substitution to obtain values of an expected type.

Atomization

Atomization is the process of converting a sequence of items into a sequence of atomic values. Atomization is used by expressions whenever a sequence of atomic values is required.

Each item in a sequence is converted to an atomic value by applying the following rules:

- If the item is an atomic value, then the atomic value is returned.
- If the item is a node, its typed value is returned. The *typed value* of a node is a sequence of zero or more atomic values that can be extracted from the node. If the node has no typed value, then an error is returned.

Implicit atomization of a sequence produces the same result as invoking the fn:data function explicitly on a sequence.

For example, the following sequence contains a combination of nodes and atomic values:

```
("Some text", <anElement>More text</anElement>, 1001)
```

Applying atomization to this sequence results in the following sequence of atomic values:

```
("Some text", "More text", 1001)
```

The following XQuery expressions use atomization to convert items into atomic values:

- Arithmetic expressions
- Comparison expressions
- Function calls with arguments whose expected types are atomic
- Cast expressions

Type promotion

Type promotion is a process that converts an atomic value from its original type to the type that is expected by an expression. XQuery uses type promotion during the evaluation of function calls and operators that accept numeric or string operands.

XQuery permits the following type promotions:

Numeric type promotion:

A value of type xs:decimal (or any type that is derived by restriction from xs:decimal) can be promoted to xs:double. The result of this promotion is created by casting the original value to the required type.

In the following example, a sequence that contains the xs:double value 13.54e-2 and the xs:decimal value 100 is passed to the fn:sum function, which returns a value of type xs:double:

```
fn:sum(xs:double(13.54e-2), xs:decimal(100))
```

Type promotion and subtype substitution differ in the following ways:

- For type promotion, the atomic value is actually converted from its original type to the type that is expected by an expression.

- For subtype substitution, an expression that expects a specific type can be invoked with a value that is derived from that type. However, the value retains its original type.

Subtype substitution
Subtype substitution is the use of a value whose type is derived from an expected type.

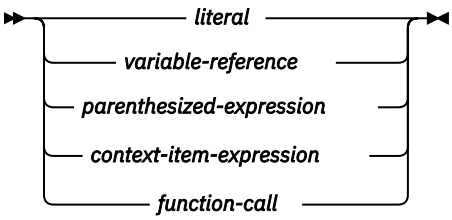
Subtype substitution does not change the actual type of a value. For example, if an xs:integer value is used where an xs:decimal value is expected, the value retains its type as xs:integer.

Subtype substitution is used whenever a value that is derived from an expected type is passed to an expression.

Primary expressions

Primary expressions are basic primitives of the language.

Syntax
primary-expression



Literals
XQuery supports two kinds of literals: numeric literals and string literals.

A *numeric literal* is an atomic value of type `xs:integer`, `xs:decimal`, or `xs:double`. A numeric literal that contains no decimal point (.) and no e or E character is an atomic value of the type `xs:integer`. A numeric literal that contains a decimal point (.), but no e or E character is an atomic value of type `xs:decimal`. A numeric literal that contains an e or E character is an atomic value of type `xs:double`. Values of numeric literals are interpreted according to the rules of XML Schema.

A *string literal* is an atomic value of type `xs:string` that is enclosed in delimiting apostrophes or quotation marks. String literals can include predefined entity references and character references.

To include an apostrophe within a string literal that is delimited by apostrophes, specify two adjacent apostrophes. Similarly, to include a quotation mark within a string literal that is delimited by quotation marks, specify two adjacent quotation marks.

If a string literal is used in an XQuery expression within the value of an XML attribute, the characters that are used to delimit the literal must be different from the characters that are used to delimit the attribute.

Examples

Example of an XQuery expression with numeric literals:

```
SELECT XMLQUERY ('7635') AS XSINTVAL,
 XMLQUERY ('8735.98834') AS XSDECVAL,
 XMLQUERY ('93948.87E+77') AS XSDOUBLEVAL
FROM T1;
```

This is the result:

```
XSINTVAL XSDECVAL   XSDOUBLEVAL
-------- --------   -----------
7635     8735.98834 93948.87E+77
```

Example of an XQuery expression that contains a string literal with an embedded double quotation mark:

```
SELECT XMLQUERY ('"string literal double-quote "" in the middle"')
FROM T1
```

This is the result:

```
string literal double-quote " in the middle
```

Related concepts

XMLQUERY function for retrieval of portions of an XML document
XMLQUERY is an SQL scalar function that lets you execute an XQuery expression from within an SQL context.

Related reference

xs:decimal
The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

xs:double
The data type `xs:double` is supported in XQuery by the IEEE 64-bit decimal floating point.

xs:integer
The data type `xs:integer` represents a decimal number that does not include a trailing decimal point. The base type of `xs:integer` is `xs:decimal`.

xs:string
The data type `xs:string` represents character strings in XML. Because `xs:string` is a simple type, it cannot contain any children.

Predefined entity references

A *predefined entity reference* is a short sequence of characters that represents a character that has some syntactic significance in XQuery.

A predefined entity reference begins with an ampersand (&) and ends with a semicolon (;). When a string literal is processed, each predefined entity reference is replaced by the character that it represents. The following table lists the predefined entity references that XQuery recognizes:

Table 35. Predefined entity references in XQuery

Entity reference	Character represented
<	<
>	>
&	&
"	"
'	'

Related concepts

Storage structure for XML data
The storage structure for XML data is similar to the storage structure for LOB data.

Character references

A *character reference* is an XML-style reference to a Unicode character that is identified by its decimal or hexadecimal code point.

A character reference begins with either &#x or &# and ends with a semicolon (;). If the character reference begins with &#x, the digits and letters up to the terminating semicolon (;) provide a hexadecimal representation of the character's code point in ISO/IEC 10646. If the character reference

begins with &#, the digits up to the terminating semicolon (;) provide a decimal representation of the character's code point.

Example

The character reference € represents the Euro symbol (€).

Variable references in XQuery
A variable reference is a QName that is preceded by a dollar sign ($). When an XQuery expression is evaluated, each variable reference resolves to the value of the expression that is bound to the variable.

Every variable reference must match a name in the in-scope variables for the XQuery expression. In-scope variables are bound from the SQL context that invokes the XQuery expression (XMLQUERY or XMLEXISTS).

Two variable references are equivalent if their local names are the same and their namespace prefixes are bound to the same namespace URI in the in-scope namespaces. An variable reference with no prefix is in no namespace.

Examples

In the following example, the XMLQUERY function binds the value of the host variable :IHV to $PARTNUMBER, and the value of column C1 to $QTY.

```
SELECT XMLQUERY('//item[@partNum = $PARTNUMBER and quantity=$QTY])'
  PASSING XMLPO, :IHV AS PARTNUMBER, C1 AS QTY)
FROM T1;
```

Related concepts
XMLEXISTS predicate for querying XML data
The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.

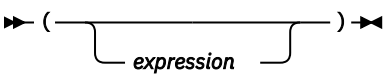
XMLQUERY function for retrieval of portions of an XML document
XMLQUERY is an SQL scalar function that lets you execute an XQuery expression from within an SQL context.

Parenthesized expression
Parentheses can be used to enforce a particular order of evaluation in expressions that contain multiple operators.

Use a parenthesized expression to explicitly specify the order of operations in a complex arithmetic expression.

Empty parentheses are used to denote an empty sequence.

Syntax
parenthesized-expression



Examples

In the following example, the parenthesized expressions 5+5 and 6+4 are evaluated first.

```
SELECT XMLQUERY (' ((5+5) * (6+4)) div 5') FROM T1
```

The result is 20.

Context item expressions

A context item expression consists of a single period (.). A context item expression evaluates to the item that is currently being processed, which is known as the *context item*. The context item can be either a node or an atomic value.

Example

The following example contains a context item expression that identifies nylon pants in the products document:

```
SELECT XMLQUERY ('declare namespace ipo="http://www.example.com/IPO";
 /ipo:products/product/name[. = "Nylon pants"]'
 PASSING XMLPO)
FROM T1
```

The result is:

```
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:ipo="http://www.example.com/IPO">Nylon pants</name>
```

Related concepts

Document nodes
A document node encapsulates an XML document.

XMLQUERY function for retrieval of portions of an XML document
XMLQUERY is an SQL scalar function that lets you execute an XQuery expression from within an SQL context.

Related reference

Axis steps
An *axis step* consists of three parts: an optional axis, a node test, and zero or more predicates.

Function calls in XQuery

XQuery supports calls to built-in XQuery functions.

Built-in XQuery functions are in the namespace `http://www.w3.org/2003/11/xpath-functions`. If the function name in the function call has no namespace prefix, the function is considered to be in the default function namespace.

XQuery uses the following process to evaluate functions:

1. XQuery evaluates each expression that is passed as an argument in the function call and returns a value for each expression.
2. The value that is returned for each argument is converted to the data type that is expected for that argument. When the expected type is a sequence of zero or more atomic types, XQuery uses the following rules to convert the value to its expected type:
   a. The given value is atomized into a sequence of atomic values.
   b. Each item in the atomic sequence that is of type `xs:untypedAtomic` is cast to the expected atomic type. For built-in functions where the expected type is specified as `numeric`, arguments of type `xs:untypedAtomic` are cast to `xs:double`.
   c. Numeric type promotion is applied to any numeric item in the atomic sequence that can be promoted to the expected atomic type through numeric type promotion. Numeric items include items of type `xs:integer`, `xs:decimal`, or `xs:double`.
3. The function is evaluated using the converted values of its arguments. The result of the function call is either an instance of the function's declared return type or an error.

Examples

The following function call retrieves the first three characters of the pid attribute of a product document:

```
SELECT XMLQUERY ('declare namespace pos="http://posample.org";
  fn:substring(/pos:product/@pid, 1, 3)'
  PASSING DESCRIPTION)
FROM T1;
```

Related information

Descriptions of XQuery functions

The XQuery functions are a subset of the XPath 2.0 and XQuery 1.0 functions and operators.

Path expressions

Path expressions locate nodes within an XML tree. Path expressions in XQuery are based on the syntax of XPath 2.0.

A path expression consists of a series of one or more steps that are separated by a slash character (/) or two slash characters (//). The path can begin with a slash character (/), two slash characters(//), or a step. Two slash characters (//) in a path expression are expanded as /descendant-or-self::node()/ , which leaves a sequence of steps separated by a slash character (/). A step generates a sequence of items. The steps in a path expression are evaluated from left to right. The sequence of items that a step generates are used as context nodes for the step that follows. For example, in the expression `description/name`, the first step generates a sequence of nodes that includes all `description` elements. The final step evaluates the `name` element once for each `description` item in the sequence. Each time a `name` element is evaluated, it is evaluated with a different focus, until all `name` elements have been evaluated. The sequences that result from each evaluation of the step are combined, and duplicate nodes are eliminated based on node identity.

A slash character (/) at the beginning of a path expression means that the path is to begin at the root node of the tree that contains the context node. That root node must be a document node.

Restriction: In Db2, XQuery path expressions cannot contain the comma operator.

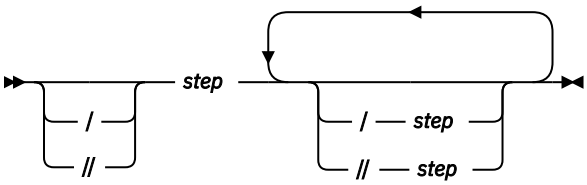
Recommendation: Because the slash character can be used as both an operator and an operand, use parentheses to clarify the meaning of the slash character when it is used as the first character of an operator. For example, to specify an empty path expression as the left operand of a multiplication operation use `(/)*5` instead of `/*5`. The later expression causes an error. Because path expressions have the higher precedence, Db2 interprets this expression as a path expression with a wildcard for a name test (`/*`) that is followed by the token 5.

Two slash characters (//) at the beginning of a path expression establishes an initial node sequence that contains the root of the tree in which the context node is found and all nodes descended from this root. This node sequence is used as the input to subsequent steps in the path expression. That root node must be a document node.

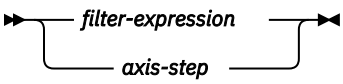
The value of the path expression is the combined sequence of items that results from the final step in the path. This value is a sequence of nodes or an atomic value. A path expression that returns a mixture of nodes and atomic values results in an error.

A step consists of an axis step or a filter expression.

Syntax
path-expression



step



Example

Use a path expression to determine which stocks have at least one bid for which the price is greater than the price of some offer on that stock.

```
//stock[bid/xs:double(price) > offer/price]/@stock_id
```

Related concepts

XMLEXISTS predicate for querying XML data

The XMLEXISTS predicate can be used to restrict the set of rows that a query returns, based on the values in XML columns.

Axis steps

An *axis step* consists of three parts: an optional axis, a node test, and zero or more predicates.

The *node test* specifies the criteria used to select nodes. The *predicates* filter the sequence that is returned by the axis step.

The result of an axis step is always a sequence of zero or more nodes, and these nodes are returned in document order. An axis step can be either a *forward step*, which starts at the context node and moves down through the XML tree, or a *reverse step*, which starts at the context node and moves up through the XML tree. If the context item is not a node, then the expression results in a type error.

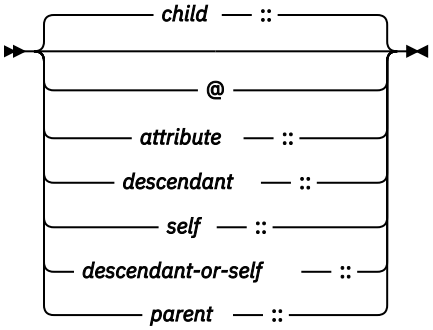
The unabbreviated syntax for an axis step consists of an axis name and node test that are separated by a double colon. The syntax of an axis expression can be abbreviated by omitting the axis and using shorthand notations.

Syntax

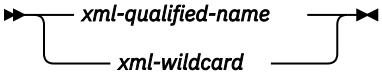
axis-step



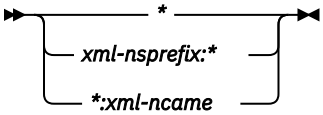
axis



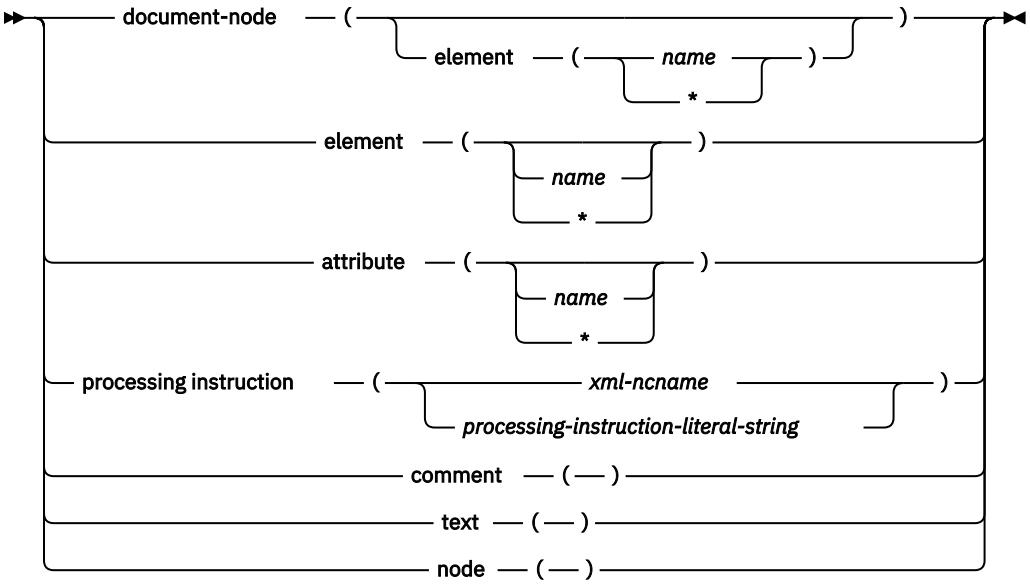
xmlname-test



xml-wildcard



xmlkind-test



predicate-list



Example

In the following example, `child` is the name of the axis and `price` is the name of the element nodes to be selected on this axis.

```
child::price
```

The axis step in this example selects all `price` elements that are children of the context node.

Related reference

Abbreviated syntax for path expressions
XQuery provides an abbreviated syntax for expressing axes in path expressions.

Node tests
A *node test* is a condition that must be true for each node that is selected by an axis step. The node test can be expressed as a name test or a kind test. A *name test* selects nodes based on the name of the node. A *kind test* selects nodes based on the kind of node.

Predicates
A *predicate* consists of an expression, called a predicate expression, that is enclosed in square brackets ([]). A predicate filters a sequence by retaining some items and discarding others.

Axes

An *axis* is an optional part of an axis step that specifies a direction of movement through an XML document. XQuery supports all the axes except the optional axes that are defined by XQuery.

Table 36 on page 191 describes the axes that are supported in XQuery.

Table 36. Supported axes in XQuery

Axis	Description	Notes
child	Returns the children of the context node. This axis is the default.	Document nodes and element nodes are the only nodes that have children. If the context node is any other kind of node, or if the context node is an empty document or element node, then the child axis is an empty sequence. The children of a document node or element node may be element, processing instruction, comment, or text nodes. Attribute and document nodes can never appear as children.
descendant	Returns the descendants of the context node (the children, the children of the children, and so on).	
attribute	Returns the attributes of the context node.	This axis is empty if the context node is not an element node.
self	Returns the context node only.	
descendant-or-self	Returns the context node and the descendants of the context node.	
parent	Returns the parent of the context node, or an empty sequence if the context node has no parent.	An element node can be the parent of an attribute node even though an attribute node is never a child of an element node.

Important: XQuery does not support the full axis feature of XQuery.

An axis can be either a forward or reverse axis. A *forward axis* contains the context node and nodes that are after the context node in document order. A *reverse axis* contains the context node and nodes that are before the context node in document order. In XQuery, the forward axes include: child, descendant, attribute, self, and descendant-or-self. The only supported reverse axis is the parent axis.

When an axis step selects a sequence of nodes, each node is assigned a context position that corresponds to its position in the sequence. If the axis is a forward axis, context positions are assigned to the nodes in document order, starting with 1. If the axis is a reverse axis, context positions are assigned to the nodes in reverse document order, starting with 1.

Related reference

Abbreviated syntax for path expressions
XQuery provides an abbreviated syntax for expressing axes in path expressions.

Node tests

A *node test* is a condition that must be true for each node that is selected by an axis step. The node test can be expressed as a name test or a kind test. A *name test* selects nodes based on the name of the node. A *kind test* selects nodes based on the kind of node.

Name tests

A name test consists of a QName or a wildcard. When a name test is specified in an axis step, the step selects the nodes on the specified axis that match the QName or wildcard. If the name test is specified on the attribute axis, the step selects any attributes that match the name test. Otherwise, on all other axes, the step selects any elements that match the name test. For the QNames to match, the expanded QName of the node must be equal (on a codepoint basis) to the expanded QName that is specified in the name test. Two expanded QNames are equal if their namespace URIs are equal and their local names are equal (even if their namespace prefixes are not equal).

Important: Any prefix that is specified in a name test must correspond to one of the statically known namespaces for the expression. For name tests that are performed on the attribute axis, unprefixed QNames have no namespace URI. For name tests that are performed on all other axes, unprefixed QNames have the namespace URI of the default element namespace.

Table 37 on page 192 describes the name tests that are supported in XQuery.

Table 37. Supported name tests in XQuery

Test	Description	Examples
QName	Matches any nodes (on the specified axis) whose QName is equal to the specified QName. If the axis is an attribute axis, this test matches attribute nodes that are equal to the specified QName. On all other axes, this test matches element nodes that are equal to the specified QName.	In the expression `child::para`, the name test `para` selects all of the `para` elements on the child axis.
*	Matches all nodes on the specified axis. If the axis is an attribute axis, this test matches all attribute nodes. On all other axes, this test matches all element nodes.	In the expression, `child::*`, the name test * matches all elements on the child axis.

Kind tests

When a kind test is specified in an axis step, the step selects only those nodes on the specified axis that match the kind test. Table 38 on page 192 describes the kind tests that are supported in XQuery.

Table 38. Supported kind tests in XQuery

Test	Description	Examples
node()	Matches any node on the specified axis.	In the expression `child::node()`, the kind test `node()` selects any nodes on the child axis.
text()	Matches any text node on the specified axis.	In the expression `child::text()`, the kind test `text()` selects any text nodes on the child axis.
comment()	Matches any comment node on the specified axis.	In the expression `child::comment()`, the kind test `comment()` selects any comment nodes on the child axis.
processing-instruction(NCName)	Matches any processing-instruction node (on the specified axis) whose name (called its "PITarget" in XML) matches the NCName that is specified in this name test.	In the expression `child::processing-instruction( xml-stylesheet)`, the kind test `processing-instruction( xml-stylesheet)` selects any processing instruction nodes on the child axis whose PITarget is xml-stylesheet.

Table 38. Supported kind tests in XQuery (continued)

Test	Description	Examples
processing-instruction(*StringLiteral*)	Matches any processing-instruction node (on the specified axis) whose name matches the string literal that is specified in this test. This node test provides backwards compatibility with XQuery 1.0.	In the expression `child::processing-instruction("xml-stylesheet")`, the kind test `processing-instruction("xml-stylesheet")` selects any processing instruction nodes on the child axis whose PITarget is xml-stylesheet.
element()	Matches any element node on the specified axis.	In the expression `child::element()`, the kind test `element()` selects any element nodes on the child axis.
element(*QName*)	Matches any element node (on the specified axis) whose name matches the qualified name that is specified in this test.	In the expression `child::element("price")`, the kind test `element("price")` selects any element nodes on the child axis whose name is price.
element(*)	Matches any element node on the specified axis.	In the expression `child::element(*)`, the kind test `element(*)` selects any element nodes on the child axis.
attribute()	Matches any attribute node on the specified axis.	In the expression `child::attribute()`, the kind test `attribute()` selects any attribute nodes on the child axis.
attribute(*QName*)	Matches any attribute node (on the specified axis) whose name matches the qualified name that is specified in this test.	In the expression `child::attribute("price")`, the kind test `attribute("price")` selects any attribute nodes on the child axis whose name is price.
attribute(*)	Matches any attribute node on the specified axis.	In the expression `child::attribute(*)`, the kind test `attribute(*)` selects any attribute nodes on the child axis.
document-node()	Matches any document node on the specified axis.	In the expression `child::document-node()`, the kind test `document-node()` selects any document nodes on the child axis.
document-node(element(*QName*))	Matches any document node on the specified axis that has only one element node (on the specified axis), and the name of the node matches the qualified name that is specified in this test.	In the expression `child::document-node(element("price"))`, the kind test `document-node(element("price"))` selects any document nodes on the child axis that have a single element whose name is price.

Table 38. Supported kind tests in XQuery (continued)

Test	Description	Examples
document-node(element(*))	Matches any document node on the specified axis that has element nodes.	In the expression `child::document-node(element(*))`, the kind test `document-node(element(*))` selects any document nodes on the child axis that have element nodes.

Related concepts

Nodes

A *node* conforms to one of the types of nodes that are defined for XQuery. These node types include: document, element, attribute, text, processing instruction, comment, and namespace nodes.

Related reference

Axis steps

An *axis step* consists of three parts: an optional axis, a node test, and zero or more predicates.

Predicates

A *predicate* consists of an expression, called a predicate expression, that is enclosed in square brackets ([]). A predicate filters a sequence by retaining some items and discarding others.

The predicate expression is evaluated once for each item in the sequence. The result of the predicate expression is an `xs:boolean` value called the *predicate truth value*. Those items for which the predicate truth value is true are retained, and those for which the predicate truth value is false are discarded. The value of the predicate expression must not be a numeric value. For all other data types, the predicate truth value is the effective boolean value of the predicate expression. The effective boolean value is `false` if the predicate expression evaluates to an empty sequence or `false`. Otherwise, the effective boolean value is `true`.

Examples

The following examples are axis steps that include predicates:

- `descendant::phone[attribute::type = "work"]` selects all the descendants of the context node that are elements named phone and whose type attribute has the value "work".
- `child::address[prov-state][pcode-zip]` selects all the `address` children of the context node that have a `prov-state` child element and a `pcode-zip` child element.

Related reference

xs:boolean

The data type `xs:boolean` supports the mathematical concept of binary-valued logic: true or false.

Abbreviated syntax for path expressions

XQuery provides an abbreviated syntax for expressing axes in path expressions.

Table 39 on page 194 describes the abbreviations that are allowed in path expressions.

Table 39. Abbreviated syntax for path expressions

Abbreviated syntax	Description
no axis specified	Shorthand abbreviation for `child::`, except when the axis step specifies `attribute()` for the node test. When the axis step specifies an attribute test, an omitted axis is shorthand for `attribute::`.
@	Shorthand abbreviation for `attribute::` .

Table 39. Abbreviated syntax for path expressions (continued)

Abbreviated syntax	Description
//	Shorthand abbreviation for `/descendant-or-self::node()/`, except when this abbreviation appears at the beginning of the path expression. When this abbreviation appears at the beginning of the path expression, the axis step selects an initial node sequence that contains the root of the tree in which the context node is found, plus all nodes that are descended from this root. This expression raises an error if the root node is not a document node.
..	Shorthand abbreviation for `parent::node()`.

Examples of abbreviated and unabbreviated syntax

Table 40 on page 195 provides examples of abbreviated and unabbreviated syntax.

Table 40. Unabbreviated and abbreviated syntax compared

Unabbreviated syntax	Abbreviated syntax	Result
`child::para`	`para`	Selects the `para` elements that are children of the context node.
`child::*`	`*`	Selects all elements that are children of the context node.
`child::text()`	`text()`	Selects all text nodes that are children of the context node.
`child::node()`	`node()`	Selects all of the children of the context node. This expression returns no attribute nodes, because attributes are not children of a node.
`attribute::name`	`@name`	Selects the `name` attribute of the context node
`attribute::*`	`@*`	Selects all of the attributes of the context node.
`child::para[attribute::type="warning"]`	`para[@type="warning"]`	Selects all `para` children of the context node that have a type attribute with the value `warning`.
`child::chapter[child::title="Introduction"]`	`chapter[title="Introduction"]`	Selects the `chapter` children of the context node that have one or more `title` children whose typed value is equal to the string `Introduction`.
`child::chapter[child::title]`	`chapter[title]`	Selects the `chapter` children of the context node that have one or more `title` children.

Related concepts

Nodes

A *node* conforms to one of the types of nodes that are defined for XQuery. These node types include: document, element, attribute, text, processing instruction, comment, and namespace nodes.

Sequence expressions

Sequence expressions construct, filter, and combine sequences of items. Sequences are never nested. For example, combining the values 1, (2, 3), and ( ) into a single sequence results in the sequence (1, 2, 3).

Expressions that construct sequences

You can use the comma operator to construct sequences.

Comma operators

To construct a sequence by using the comma operator, specify two or more operands (expressions) that are separated by commas. When XQuery evaluates the sequence expression, it evaluates the operands of each comma operator and concatenates the resulting sequences, in order, into a single result sequence. For example, the following expression results in a sequence that contains five integers:

```
(15, 1, 3, 5, 7)
```

Restriction: The operands (expressions) of the comma operator cannot contain an FLWOR expression.

Restriction: XQuery path expressions cannot contain the comma operator.

A sequence can contain duplicate atomic values and nodes. However, a sequence is never an item in another sequence. When a new sequence is created by concatenating two or more input sequences, the new sequence contains all of the items of the input sequences. The length of the sequence is the sum of the lengths of the input sequences.

Examples: The following expressions use the comma operator for sequence construction:

- This expression combines four sequences of length one, two, zero, and two, respectively, into a single sequence of length five. The result of this expression is the sequence 10, 1, 2, 3, 4.

  ```
  (10, (1, 2), (), (3, 4))
  ```

- The result of this expression is a sequence that contains all `salary` elements that are children of the context node, followed by all `bonus` elements that are children of the context node.

  ```
  (salary, bonus)
  ```

- Assuming that the variable $price is bound to the value 10.50, the result of this expression is the sequence 10.50, 10.50.

  ```
  ($price, $price)
  ```

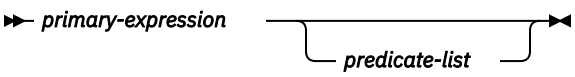
Filter expressions

A filter expression consists of a primary expression that is followed by zero or more predicates. The predicates, if present, filter the result of the primary expression.

The result of a filter expression consists of all the items that are returned by the primary expression for which all the predicates are true. If no predicates are specified, the result is the result of the primary expression. This result can contain nodes, atomic values, or a combination of nodes and atomic values. The ordering of the items that are returned by a filter expression is the same as their order in the result of the primary expression. Context positions are assigned to items based on their ordinal position in the result sequence. The first context position is 1.

Restriction: An XPath step node or XPath predicate cannot contain FLWOR expressions, conditional expressions, or comma operators.

Syntax
filter-expression



predicate-list



Examples

The following example uses a filter expression that returns a value if there is a customerinfo element anywhere in the document that is specified by $x:

```
SELECT XMLQUERY ('declare default element namespace "http://posample.org";
  $x[.//customerinfo]'
  PASSING PASSING INFO AS "x")
FROM CUSTOMER
```

Arithmetic expressions

Arithmetic expressions perform operations that involve addition, subtraction, multiplication, division, and modulus.

The following table describes the arithmetic operators and lists them in order of operator precedence from highest to lowest. Unary operators have a higher precedence than binary operators unless parentheses are used to force the evaluation of the binary operator.

Table 41. Arithmetic operators in XQuery

Operator	Purpose	Associativity
-(unary), + (unary)	negates value of operand, maintains value of operand	right-to-left
*, div, idiv, mod	multiplication, division, integer division, modulus	left-to-right
+, -	addition, subtraction	left-to-right

Note: A subtraction operator must be preceded by whitespace if the operator could otherwise be interpreted as part of a previous token. For example, a-b is interpreted as a name, but a - b and a -b are interpreted as arithmetic operations.

Restriction: The operand of an arithmetic operator cannot contain an FLWOR expression.

The result of an arithmetic expression is one of the following items:

- A numeric value
- A date or time value
- A duration value
- An empty sequence
- An error

XQuery uses the following process to evaluate an arithmetic expression.

1. Atomizes each operand into a sequence of atomic values.
2. Uses the following rules to evaluate the operands in the arithmetic expression:

- If the atomized operand is an empty sequence, the result of the arithmetic expression is an empty sequence.
- If the atomized operand is a sequence that contains more than one value, an error is returned.
- If the atomized operand is an untyped atomic value (xs:untypedAtomic), XQuery casts the value to xs:double. If the cast fails, XQuery returns an error.

3. If the types of the operands are a valid combination for the arithmetic operator, XQuery applies the operator to the atomized values. The result of this operation is an atomic value or a dynamic error (for example, an error might result from dividing by zero).
4. If the types of the operands are not a valid combination for the arithmetic operator, XQuery raises a type error.

The following table identifies valid combinations of types for arithmetic operators. In this table, the letter A represents the first operand in the expression, and the letter B represents the second operand. The term numeric denotes the types xs:integer, xs:decimal, xs:double, or any types derived from one of these types. If the result type of an operator is listed as numeric, the result type will be the first type in the ordered list (xs:integer, xs:decimal, xs:double) into which all operands can be converted by subtype substitution and type promotion.

Table 42. Valid types for operands of arithmetic expressions

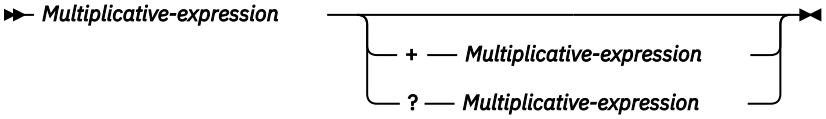
Operator with operands	Type of operand A	Type of operand B	Result type
A + B	numeric	numeric	numeric
	xs:date	xs:yearMonthDuration	xs:date
	xs:yearMonthDuration	xs:date	xs:date
	xs:date	xs:dayTimeDuration	xs:date
	xs:dayTimeDuration	xs:date	xs:date
	xs:time	xs:dayTimeDuration	xs:time
	xs:dayTimeDuration	xs:time	xs:time
	xs:dateTime	xs:yearMonthDuration	xs:dateTime
	xs:yearMonthDuration	xs:dateTime	xs:dateTime
	xs:dateTime	xs:dayTimeDuration	xs:dateTime
	xs:dayTimeDuration	xs:dateTime	xs:dateTime
	xs:yearMonthDuration	xs:yearMonthDuration	xs:yearMonthDuration
	xs:dayTimeDuration	xs:dayTimeDuration	xs:dayTimeDuration

Table 42. Valid types for operands of arithmetic expressions (continued)

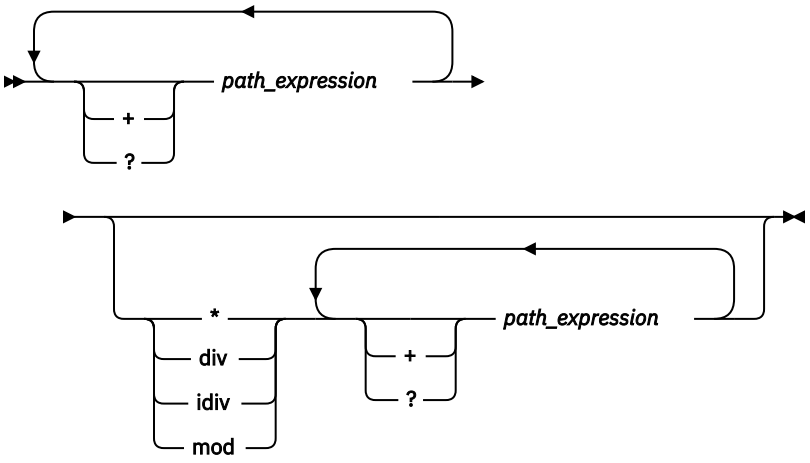
Operator with operands	Type of operand A	Type of operand B	Result type
A - B	numeric	numeric	numeric
	xs:date	xs:date	xs:dayTimeDuration
	xs:date	xs:yearMonthDuration	xs:date
	xs:date	xs:dayTimeDuration	xs:date
	xs:time	xs:time	xs:dayTimeDuration
	xs:time	xs:dayTimeDuration	xs:time
	xs:dateTime	xs:dateTime	xs:dayTimeDuration
	xs:dateTime	xs:yearMonthDuration	xs:dateTime
	xs:dateTime	xs:dayTimeDuration	xs:dateTime
	xs:yearMonthDuration	xs:yearMonthDuration	xs:yearMonthDuration
	xs:dayTimeDuration	xs:dayTimeDuration	xs:dayTimeDuration
A * B	numeric	numeric	numeric
	xs:yearMonthDuration	numeric	xs:yearMonthDuration
	numeric	xs:yearMonthDuration	xs:yearMonthDuration
	xs:dayTimeDuration	numeric	xs:dayTimeDuration
	numeric	xs:dayTimeDuration	xs:dayTimeDuration
A idiv B	numeric	numeric	xs:integer
A div B	numeric	numeric	numeric; but xs:decimal if both operands are xs:integer
	xs:yearMonthDuration	numeric	xs:yearMonthDuration
	xs:dayTimeDuration	numeric	xs:dayTimeDuration
	xs:yearMonthDuration	xs:yearMonthDuration	xs:decimal
	xs:dayTimeDuration	xs:dayTimeDuration	xs:decimal
A mod B	numeric	numeric	numeric

Syntax

arithmetic expression



multiplicative expression



Examples

The following query uses an arithmetic expression to calculate the amount that buyers pay in taxes on a product, at a rate of 8.25%, and selects the description elements for which the tax is greater than one unit of currency.

```
SELECT XMLQUERY ('declare namespace pos="http://posample.org";
  /pos:product/description[price * .0825 > 1]'
  PASSING DESCRIPTION)
FROM T1
```

The following query subtracts two xs:date values, which results in the xs:yearMonthDuration value P8559D:

```
SELECT XMLQUERY(' xs:date("2005-10-10")
  - xs:date("1982-05-05")')
 FROM SYSIBM.SYSDUMMY1
```

Related reference

Parenthesized expression
Parentheses can be used to enforce a particular order of evaluation in expressions that contain multiple operators.

Comparison expressions

Comparison expressions compare two values. XQuery provides general comparisons, value comparisons, and node comparisons.

Value comparisons

Value comparisons compare two atomic values. The value comparison operators include **eq**, **ne**, **lt**, **le**, **gt**, and **ge**.

The following table describes these operators.

Table 43. Value comparison operators in XQuery

Operator	Purpose
eq	Returns true if the first value is equal to the second value.
ne	Returns true if the first value is not equal to the second value.
lt	Returns true if the first value is less than the second value.
le	Returns true if the first value is less than or equal to the second value.

Table 43. Value comparison operators in XQuery (continued)

Operator	Purpose
gt	Returns true if the first value is greater than the second value.
ge	Returns true if the first value is greater than or equal to the second value.

Restriction: The operand of a value comparison cannot contain an FLWOR expression.

Two values can be compared if they have the same type or if the type of one operand is a subtype of the other operand's type. Two operands of numeric types (types xs:integer, xs:decimal, xs:double, and types derived from these) can be compared.

Special values: For xs:double values, positive zero and negative zero compare equal. INF equals INF, and -INF equals -INF. NaN does not equal itself. Positive infinity is greater than all other non-NaN values; negative infinity is less than all other non-NaN values. NaN **ne** NaN is true, and any other comparison involving a NaN value is false.

The result of a value comparison can be a boolean value, an empty sequence, or an error. When a value comparison is evaluated, each operand is *atomized* (converted into an atomic value), and the following rules are applied:

- If either atomized operand is an empty sequence, the result of the value comparison is an empty sequence.
- If either atomized operand is a sequence that contains more than one value, an error is returned.
- If either atomized operand is an untyped atomic value (xs:untypedAtomic), that value is cast to xs:string.

  Casting values of type xs:untypedAtomic to xs:string allows value comparisons to be transitive. In contrast, general comparisons follow a different rule for casting untyped data and are therefore not transitive.
- If the types of the operands, after evaluation, are a valid combination for the operator, the operator is applied to the atomized operands, and the result of the comparison is either true or false. If the types of the operands are not a valid combination for the comparison operator, an error is returned.

The following types can be compared with value comparison operators. The term numeric refers to the types xs:integer, xs:decimal, xs:double, and any type derived from one of these types. During comparisons that involve numeric values, subtype substitution and numeric type promotion are used to convert the operands into the first type in the ordered list (xs:integer, xs:decimal, xs:double) into which all operands can be converted.

- Numeric
- xs:boolean
- xs:string
- xs:date
- xs:time
- xs:dateTime
- xs:yearMonthDuration
- xs:dayTimeDuration
- xs:duration (eq and ne only)

Examples

- The following comparison atomizes the nodes that are returned by the expression $book/author. The comparison is true only if the result of atomization is the value "Kennedy" as an instance of xs:string or xs:untypedAtomic. If the result of atomization is a sequence that contains more than one value, an error is returned

```
$book1/author eq "Kennedy"
```

- The following path expression contains a predicate that selects products whose weight is greater than 100. For any product that does not have a weight subelement, the value of the predicate is the empty sequence, and the product is not selected:

```
//product[xs:decimal(weight) gt 100]
```

- The following comparisons are true because, in each case, the two constructed nodes have the same value after atomization, even though they have different identities or names:

```
<a>5</a> eq <a>5</a>
<a>5</a> eq <b>5</b>
```

General comparisons in XQuery

General comparisons compare two sequences of any length to determine if a comparison is true for at least one item in both sequences. The general comparison operators include =, !=, <, <=, >, and >=.

The following table describes these operators, listed in order of operator precedence from highest to lowest.

Table 44. General comparison operators in XQuery

Operator	Purpose
=	Returns true if any value in the first sequence is equal to any value in the second sequence.
!=	Returns true if any value in the first sequence is not equal to any value in the second sequence.
<	Returns true if any value in the first sequence is less than any value in the second sequence.
<=	Returns true if any value in the first sequence is less than or equal to any value in the second sequence.
>	Returns true if any value in the first sequence is greater than any value in the second sequence.
>=	Returns true if any value in the first sequence is greater than or equal to any value in the second sequence.

Restriction: The operand of a general comparison cannot contain an FLWOR expression.

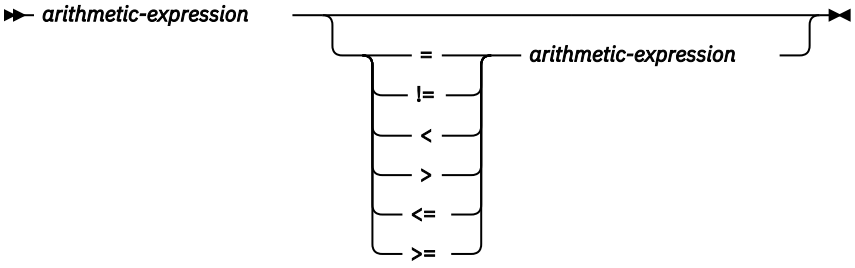
The result of a general comparison expression is a boolean value or an error. XQuery uses the following process to evaluate a general comparison expression.

1. Atomizes each operand into a sequence of atomic values.
2. Compares each of the values in the first sequence to each of the values in the second sequence. For each comparison:
   - If one of the atomic values is an instance of `xs:untypedAtomic` and the other is an instance of a numeric type (xs:integer, xs:decimal, or xs:double), the untyped value is cast to the type `xs:double`.
   - If one of the atomic values is an instance of `xs:untypedAtomic` and the other is an instance of `xs:untypedAtomic` or `xs:string`, the `xs:untypedAtomic` values are cast to the type `xs:string`.
   - If one of the atomic values is an instance of `xs:untypedAtomic` and the other is not an instance of `xs:string`, `xs:untypedAtomic`, or any numeric type, the `xs:untypedAtomic` value is cast to the dynamic type of the other value.

3. If at least one of the values in the first sequence and at least one of the values in the second sequence meet the conditions of the comparison, the general comparison is true.

Syntax

comparison expression



Examples

The following query uses a general comparison expression to find the descriptions of products that cost less than 20 units.

```
SELECT XMLQUERY ('declare namespace pos="http://posample.org";
  /pos:product/description[price < 20]'
  PASSING DESCRIPTION)
FROM T1;
```

Related concepts

XMLQUERY function for retrieval of portions of an XML document
XMLQUERY is an SQL scalar function that lets you execute an XQuery expression from within an SQL context.

Related reference

xs:date
The date type xs:date represents an interval of exactly one day that begins on the first moment of a given day.

xs:dateTime
The data type xs:dateTime represents an instant in time.

xs:dayTimeDuration
The data type xs:dayTimeDuration represents a duration of time that is expressed by days, hours, minutes, and seconds components. xs:dayTimeDuration is derived from data type xs:duration.

xs:decimal
The data type `xs:decimal` represents a subset of the real numbers that can be represented by decimal numerals.

xs:double
The data type `xs:double` is supported in XQuery by the IEEE 64-bit decimal floating point.

xs:duration
The data type xs:duration represents a duration of time that is expressed by the Gregorian year, month, day, hour, minute, and second components. xs:duration is derived from data type xs:anyAtomicType.

xs:integer
The data type `xs:integer` represents a decimal number that does not include a trailing decimal point. The base type of `xs:integer` is `xs:decimal`.

xs:string
The data type `xs:string` represents character strings in XML. Because `xs:string` is a simple type, it cannot contain any children.

xs:time

The data type xs:time represents an instant of time that recurs every day.

xs:untypedAtomic
The data type `xs:untypedAtomic` serves as a special type annotation to indicate atomic values that have not been validated by an XML schema or a DTD.

xs:yearMonthDuration
The data type xs:yearMonthDuration represents a duration of time that is expressed by the Gregorian year and month components. xs:yearMonthDuration is derived from data type xs:duration.

Node comparisons
Node comparisons compare two nodes. Nodes can be compared to determine if they share the same identity or if one node precedes or follows another node in document order.

The following table describes the node comparison operators that are available in XQuery.

Table 45. Node comparison operators in XQuery

Operator	Purpose
is	Returns true if the two nodes that are compared have the same identity.
<<	Returns true if the first operand node precedes the second operand node in document order.
>>	Returns true if the first operand node follows the second operand node in document order.

Restriction: The operand of a node comparison cannot contain an FLWOR expression.

The result of a node comparison is either a Boolean value, an empty sequence, or an error. The result of a node comparison is defined by the following rules:

- Each operand must be either a single node or an empty sequence; otherwise, an error is returned.
- If either operand is an empty sequence, the result of the comparison is an empty sequence.
- A comparison that uses the **is** operator is true when the two nodes that are compared have the same identity; otherwise, the comparison is false.
- A comparison that uses the << operator returns true when the left operand node precedes the right operand node in document order; otherwise, the comparison returns false. If the nodes are from two different documents, the document ID determines the precedence.
- A comparison that uses the >> operator returns true when the left operand node follows the right operand node in document order; otherwise, the comparison returns false. If the nodes are from two different documents, the document ID determines the precedence.

Examples

- The following comparison is true only if both the left operand and right operand evaluate to exactly the same single node:

```
/books/book[isbn="1558604820"] is /books/book[call="QA76.9 C3845"]
```

- The following comparison is false because each constructed node has its own identity:

```
<a>5</a> is <a>5</a>
```

- The following comparison is true only if the node that is identified by the left operand occurs before the node that is identified by the right operand in document order:

```
/transactions/purchase[parcel="28-451"] << /transactions/sale[parcel="33-870"]
```

Logical expressions

Logical expressions return the boolean value true if both of two expressions are true, or if one or both of two expressions are true. The operators that are used in logical expressions include `and` and `or`.

The following table describes these operators, listed in order of operator precedence from highest to lowest.

Table 46. Logical expression operators in XQuery

Operator	Purpose
and	Returns true if both expressions are true.
or	Returns true if one or both expressions are true.

Restriction: The operand of a logical expression cannot contain an FLWOR expression.

The result of a logical expression is a boolean value or an error. XQuery uses the following process to evaluate a logical expression.

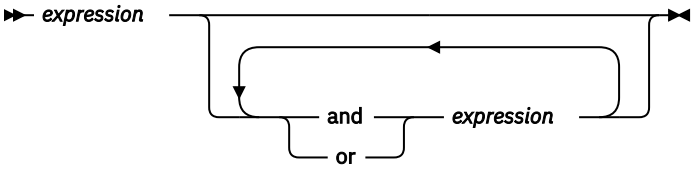
1. Determines the effective boolean value (EBV) of each operand.
2. Applies the operator to the effective boolean values of the operands. The result is a boolean value or an error. Table 47 on page 205 shows the results that are returned by a logical expression based on the EBV of its operands and any errors that are encountered during the evaluation of an operand.

Table 47. Results of logical expressions based on effective boolean values (EBVs) of operands

EBV of operand 1	Operator	EBV of operand 2	Result
true	and	true	true
true	and	false	false
false	and	true	false
false	and	false	false
true	and	error	error
error	and	true	error
false	and	error	false or error
error	and	false	false or error
error	and	error	error
true	or	true	true
false	or	false	false
true	or	false	true
false	or	true	true
true	or	error	true or error
error	or	true	true or error
false	or	error	error
error	or	false	error
error	or	error	error

Syntax

logical expression



Examples

The following query uses a logical expression to retrieve records from a table for 22-inch snow shovels or 24-inch snow shovels.

```
SELECT XMLQUERY ('declare namespace pos="http://posample.org";
  /pos:product/description[name = "Snow Shovel, Deluxe 24"""
  or name = "Snow Shovel, Basic 22"""]'
  PASSING DESCRIPTION)
FROM T1;
```

Related reference

fn:boolean function
The fn:boolean function returns the effective boolean value of a sequence.

fn:not function
The fn:not function returns false if the effective boolean value of an item is true. fn:not returns true if the effective boolean value of an item is false.

XQuery constructors

XQuery constructors create XML structures within a query. .

Enclosed expressions in constructors
Enclosed expressions are used in constructors to provide computed values for element and attribute content. These expressions are evaluated and replaced by their value when the constructor is processed. Enclosed expressions are enclosed in curly braces ({}) to distinguish them from literal text.

Enclosed expressions can be used in the following constructors to provide computed values:

- Direct element constructors:
  - An attribute value in the start tag of a direct element constructor can include an enclosed expression.
  - The content of a direct element constructor can include an enclosed expression that computes both the content and the attributes of the constructed node.
- Document node constructors:
  - An enclosed expression can be used to generate the content of the node.

For example, the following direct element constructor includes an enclosed expression:

```
<example>
  <p> Here is a query. </p>
  <eg> $b/title </eg>
  <p> Here is the result of the query. </p>
  <eg>{ $b/title }</eg>
</example>
```

When this constructor is evaluated, it might produce the following result (whitespace is added to this example to improve readability):

```
<example>
  <p> Here is a query. </p>
  <eg> $b/title </eg>
```

```
  <p> Here is the result of the query. </p>
  <eg><title>Harold and the Purple Crayon</title></eg>
</example>
```

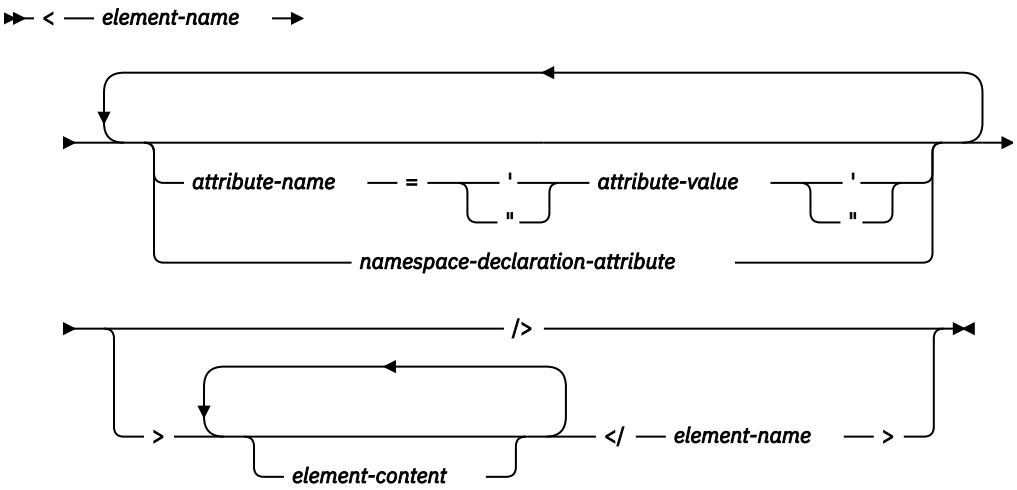
Tip: To use a curly brace as an ordinary character within the content of an element or attribute, you can either include a pair of identical curly braces or use character references. For example, you can use the pair {{ to represent the character {. Likewise, you can use the pair }} to represent }. Alternatively, you can use the character references { and } to denote curly brace characters. A single left curly brace ({) is interpreted as the beginning delimiter for an enclosed expression. A single right curly brace (}) without a matching left curly brace results in an error.

Direct element constructors

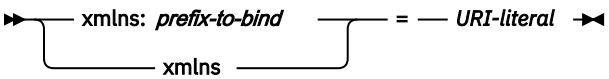
Direct element constructors use an XML-like notation to create element nodes. The constructed node can be a simple element or a complex element that contains attributes, text content, and nested elements.

The result of a direct element constructor is a new element node that has its own node identity. All of the attribute and descendant nodes of the new element node are also new nodes that have their own identities.

Syntax



namespace-declaration-attribute



element-name

An XML qualified name (QName) that represents the name of the element to construct. The name that is used for *element-name* in the end tag must exactly match the name that is used in the corresponding start tag, including the prefix or absence of a prefix. If *element-name* includes a namespace prefix, the prefix is resolved to a namespace URI by using the statically known namespaces. If *element-name* has no namespace prefix, the name is implicitly qualified by the default element namespace. The expanded QName that results from evaluating *element-name* becomes the name of the constructed element node.

attribute-name

A QName that represents the name of the attribute to construct. If *attribute-name* includes a namespace prefix, the prefix is resolved to a namespace URI by using the statically known namespaces. If *attribute-name* has no namespace prefix, the attribute is in no namespace. The expanded QName that results from evaluating *attribute-name* becomes the name of the constructed attribute node. The expanded QName of each attribute must be unique, or the expression results in a error.

Each attribute in a direct element constructor creates a new attribute node, with its own node identity. The parent of the new attribute node is the constructed element node. The new attribute node has a type annotation of xs:untypedAtomic.

attribute-value
A string of characters that specify a value for the attribute. The attribute value can contain enclosed expressions (expressions that are enclosed in curly braces) that are evaluated and replaced by their value when the element constructor is processed. Predefined entity references and character references are also valid and are replaced by the characters that they represent. The following table lists special characters that are valid within *attribute-value*, but must be represented by double characters or an entity reference.

Table 48. Representation of special characters in attribute values

Character	Representation required in attribute values
{	two open curly braces ({{)
}	two closed curly braces (}})
<	<
&	&
"	" or two double quotation marks (""), if the delimiters of the attribute value are double quotation marks
'	' or two single quotation marks (''), if the delimiters of the attribute value are single quotation marks

xmlns
The word that begins a namespace declaration attribute. When specified as a prefix in a QName, **xmlns** indicates that the value of *prefix-to-bind* will be bound to the URI that is specified by *URI-literal*. This namespace binding is added to the statically known namespaces for the constructor expression, and for all of the expressions that are nested inside of the expression, unless the binding is overridden by a nested namespace declaration attribute. For example, the namespace declaration attribute `xmlns:metric = "http://example.org/metric/units"` binds the prefix `metric` to the namespace `http://example.org/metric/units`.

When specified as the complete QName with no prefix, **xmlns** indicates that the default element namespace is set to the value of *URI-literal*. This default element namespace is in effect for this constructor expression and for all expressions that are nested inside of the constructor expression, unless the declaration is overridden by a nested namespace declaration attribute. For example, the namespace declaration attribute `xmlns = "http://example.org/animals"` sets the default element namespace to `http://example.org/animals`.

prefix-to-bind
The prefix that is to be bound to the URI that is specified for *URI-literal*. The value of *prefix-to-bind* cannot be `xml` or `xmlns`. Specification of either of these values results in an error.

URI-literal
A string literal (a sequence of zero or more characters that is enclosed in single quotation marks or double quotation marks) that represents a URI. The string literal value must be a valid URI. The value of *URI-literal* can be a zero-length string only when the namespace declaration attribute is used to set the default element namespace. Otherwise, specification of a zero-length string for *URI-literal* results in an error.

element-content
The content of the direct element constructor. The content consists of everything between the start tag and end tag of the constructor. The boundary-space declaration in the prolog controls the way in which boundary whitespace is handled in element constructors. The resulting content sequence is a concatenation of the content entities. Any resulting adjacent text characters, including text resulting

from enclosed expressions, are merged into a single text node. Any resulting attribute nodes must come before any other content in the resulting content sequence.

element-content can consist of any of the following content:

Text characters
Text characters create text nodes. Adjacent text nodes are merged into a single text node. Line endings within sequences of characters are normalized according to the rules for end-of-line handling that are specified for XML 1.0. The following table lists special characters that are valid within *element-content*, but must be represented by double characters or an entity reference.

Table 49. Representation of special characters in element content

Character	Representation required in element content
{	two open curly braces ({{)
}	two closed curly braces (}})
<	<
&	&

Nested direct constructors
Any direct constructors can be nested within direct element constructors.

CDataSections
CDataSections are specified using the following syntax: <![CDATA[*contents*]]>. *contents* is a series of characters. The characters that are specified for *contents*, including special characters such as < and &, are treated as literal characters rather than as delimiters. The sequence]]> terminates the CDataSection and is therefore not allowed within *contents*.

Character references and predefined entity references
During processing, predefined entity references and character references are expanded into their referenced strings.

Enclosed expressions
An enclosed expression is an XQuery expression that is enclosed in curly braces. For example, {5 + 7} is an enclosed expression. The value of an enclosed expression can be any sequence of nodes and atomic values. Enclosed expressions can be used within the content of a direct element constructor to compute the content and the attributes of the constructed node. For each node that is returned by an enclosed expression, a new copy is made of the node and all of its descendants, which retain their original type annotations. Any attribute nodes that are returned by *element-content* must be at the beginning of the resulting content sequence; these attribute nodes become attributes of the constructed element. Any element, content, or processing instruction nodes that are returned by *element-content* become children of the newly constructed node. Any atomic values that are returned by *element-content* are converted to strings and stored in text nodes, which become children of the constructed node. Adjacent text nodes are merged into a single text node.

Important: Because XMLQUERY, XMLEXISTS and XMLTABLE use single quotation marks to enclose an XQuery expression, single quotation marks (') within direct element constructors need to be replaced with two single quotation marks ('').

Examples

- The following direct element constructor creates a book element. The book element contains complex content that includes an attribute node, some nested element nodes, and some text nodes:

```
<book isbn="isbn-0060229357">
    <title>Harold and the Purple Crayon</title>
    <author>
        <first>Crockett</first>
        <last>Johnson</last>
```

```
    </author>
</book>
```

- The following example demonstrates the use of a CDATA section in a direct element constructor. CDATA sections are handled and stored as escaped text data. When the data is serialized, the contents of the CDATA sections display as escaped text. The CDATA sections are not preserved.

```
SELECT XMLQUERY('
   <TEST>
     {for $i in (1,2,3)
      return <a><![CDATA[<c>CDATA TEST!!!</c>]]></a> }
   </TEST>')
FROM SYSIBM.SYSDUMMY1;
```

  The SELECT statement returns results similar to these:

```
<?xml version="1.0" encoding="UTF8"?>
<TEST>
  <a>&lt;c&gt;CDATA TEST!!!&lt;/c&gt;</a>
  <a>&lt;c&gt;CDATA TEST!!!&lt;/c&gt;</a>
  <a>&lt;c&gt;CDATA TEST!!!&lt;/c&gt;</a>
</TEST>
```

- The following examples demonstrate how element content is processed in direct element constructors:
  - The following expression constructs an element node that has one child, a text node that contains the value "1":

```
<a>{1}</a>
```

  - The following expression constructs an element node that has one child, a text node that contains the value "1 2 3":

```
<a>{1, 2, 3}</a>
```

  - The following expression constructs an element node that has one child, a text node that contains the value "123":

```
<c>{1}{2}{3}</c>
```

  - The following expression constructs an element node that has one child, a text node that contains the value "1 2 3":

```
<b>{1, "2", "3"}</b>
```

  - The following expression constructs an element node that has one child, a text node that contains the value "I saw 8 cats."

```
<fact>I saw {5 + 3} cats.</fact>
```

Namespace declaration attributes

Namespace declaration attributes are specified in the start tag of a direct element constructor.

Namespace declaration attributes are used for two purposes:

- To bind a namespace prefix to a URI
- To set the default element namespace for the constructed element node and for its attributes and descendants

Syntactically, a namespace declaration attribute has the same form as an attribute in a direct element constructor: the attribute is specified by a name and a value. The attribute name is a constant qualified name (QName). The attribute value is a string literal that represents a valid URI.

A namespace declaration attribute does not cause an attribute node to be created.

Important: The name of each namespace declaration attribute in a direct element constructor must be unique. Otherwise, the expression results in an error.

How a namespace prefix is bound to a URI

If the QName begins with the prefix `xmlns` followed by a local name, the declaration is used to bind the namespace prefix (specified as the local name) to a URI (specified as the attribute value). For example, the namespace declaration attribute `xmlns:metric = "http://example.org/metric/units"` binds the prefix `metric` to the namespace `http://example.org/metric/units`.

When the namespace declaration attribute is processed, the prefix and URI are added to the statically known namespaces of the constructor expression, and the new binding overrides any existing binding of the given prefix. The prefix and URI are also added as a namespace binding to the in-scope namespaces of the constructed element.

For example, in the following element constructor, namespace declaration attributes are used to bind the namespace prefixes `metric` and `english`:

```
<box xmlns:metric = "http://example.org/metric/units"
xmlns:english = "http://example.org/english/units">
  <height> <metric:meters>3</metric:meters> </height>
  <width> <english:feet>6</english:feet> </width>
  <depth> <english:inches>18</english:inches> </depth>
</box>
```

How the default element namespace is set

If the QName is `xmlns` with no prefix, the declaration is used to set the default element namespace. For example, the namespace declaration attribute `xmlns = "http://example.org/animals"` sets the default element namespace to `http://example.org/animals`.

When the namespace declaration attribute is processed, the value of the attribute is interpreted as a namespace URI. This URI specifies the default element namespace of the constructor expression, and the new specification overrides any existing default. The URI is also added, with no prefix, to the in-scope namespaces of the constructed element, and the new specification overrides any existing namespace binding that has no prefix. If the namespace URI is a zero-length string, the default element namespace of the constructor expression is set to "none".

For example, in the following direct element constructor, a namespace declaration attribute sets the default element namespace to `http://example.org/animals`:

```
<cat xmlns = "http://example.org/animals">
  <breed>Persian</breed>
</cat>
```

Namespace bindings after elements are copied to constructors

When an existing element node is copied by an element constructor, the namespace bindings are resolved in the following way:

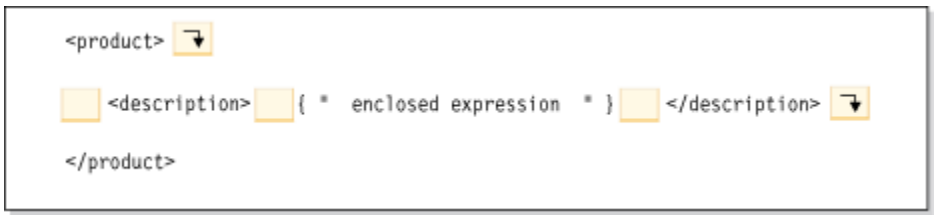
- The copied element retains all in-scope-namespaces of the original element. The default namespace is treated like any other namespace binding. The copied node has the same default namespace as the original node. If the original node has no default namespace, the copy has no default namespace.
- The copied node inherits in-scope namespaces from the constructed node. If there is a naming conflict, the namespace bindings that were copied from the original node take precedence.

Boundary whitespace in direct element constructors

Within a direct element constructor, *boundary whitespace* is a sequence of consecutive whitespace characters. The sequence is delimited at each end by the start or end of the content, by a direct constructor, or by an enclosed expression.

For example, boundary whitespace might be used in the content of the constructor to separate the end tag of a direct constructor from the start tag of a nested element.

The following diagram shows an example of a direct element constructor, with the boundary whitespace highlighted:



The boundary whitespace in this example includes the following characters:

- A newline character and four space characters that occur between the start tags of the `product` and `description` elements
- Four space characters that occur between the start tag of the `description` element and the enclosed expression
- Four space characters that occur between the enclosed expression and the end tag of the `description` element
- One newline character that appears after the end tag of the `description` element

Boundary whitespace does not include any of the following types of whitespace:

- Whitespace that is generated by an enclosed expression
- Characters that are generated by character references (for example, ` `) or by CDataSections
- Whitespace characters that are adjacent to a character reference or a CDataSection

The boundary-space policy controls whether boundary whitespace is preserved by element constructors. This policy is specified by a boundary-space declaration in the query prolog. If the boundary-space declaration specifies **`strip`**, boundary whitespace is discarded. If the boundary-space declaration specifies **`preserve`**, boundary whitespace is preserved. If no boundary-space declaration is specified, the default behavior is to strip boundary whitespace during element construction.

Examples

- In the following example, the constructed `cat` element node has two child element nodes that are named `breed` and `color`:

```
<cat>
  <breed>{$b}</breed>
  <color>{$c}</color>
</cat>
```

  Because the boundary-space policy is **`strip`** by default, the whitespace that surrounds the child elements is stripped away by the element constructor.

- In the following example, the boundary-space policy is **`strip`**. The result of the constructor is `<a>abc</a>`:

```
declare boundary-space strip;
<a> {"abc"} </a>
```

- In the following example, the boundary-space policy is **`preserve`**. The result of the constructor is `<a> abc </a>`:

```
declare boundary-space preserve;
<a> {"abc"} </a>
```

  Because the boundary-space policy is **`preserve`**, the spaces that appear before and after the enclosed expression is preserved by the element constructor.

- In the following example, the whitespace that surrounds `z` is not boundary whitespace. The whitespace is always preserved, and the result of the constructor is `<a> z abc</a>`:

```
<a> z {"abc"}</a>
```

- In the following example, the whitespace characters that are generated by the character reference and adjacent whitespace characters are preserved, regardless of the boundary-space policy. The result of the constructor is <a> abc:

```
<a>     &#x20;{"abc"}</a>
```

- In the following example, the whitespace in the enclosed expression is preserved, regardless of the boundary-space policy, because whitespace that is generated by an enclosed expression is never considered to be boundary whitespace. The result of the constructor is <a> :

```
<a>{" "}</a>
```

  The two spaces in the enclosed expression are preserved by the element constructor and appear between the start tag and the end tag in the result.

Related concepts

Whitespace in XQuery

Whitespace is allowed in most XQuery expressions to improve readability even if whitespace is not part of the syntax for the expression. Whitespace consists of space characters (U+0020), carriage returns (U +000D), line feeds (U+000A), and tabs (U+0009).

In-scope namespaces of a constructed element

A constructed element node has an in-scope namespaces property that consists of a set of namespace bindings. Each namespace binding associates a namespace prefix with a URI. The namespace bindings define the set of namespace prefixes that are available for interpreting qualified names (QNames) within the scope of an element.

Important: To understand this topic, you need to understand the difference between the following concepts:

Statically known namespaces
: *Statically known namespaces* is a property of an expression. This property denotes the set of namespace bindings that are used by XQuery to resolve namespace prefixes during the processing of the expression. These bindings are not part of the query result.

In-scope namespaces
: *In-scope namespaces* is a property of an element node. This property denotes the set of namespace bindings that are available to applications outside of XQuery when the element and its content are processed. These bindings are serialized as part of the query result so that they are available to outside applications.

The in-scope namespaces of a constructed element include all of the namespace bindings that are created in the following ways:

- Explicitly, through namespace declaration attributes

  A namespace binding is created for each namespace declaration attribute that is declared in the following constructors:

  – The current element constructor
  – An enclosing direct element constructor, unless the namespace declaration attribute is overridden by the current element constructor or an intermediate constructor

- Automatically, by the system

  A namespace binding is created in the following situations:

  – For every constructed element, to bind the prefix `xml` to the namespace URI `http://www.w3.org/XML/1998/namespace`
  – For each namespace that is used in the name of a constructed element or in the names of its attributes, unless the namespace binding already exists in the in-scope namespaces of the element. If the name of the node includes a prefix, the prefix is used in the namespace binding. If the name has no prefix, a binding is created for the empty prefix. If a conflict arises that would require two

different bindings of the same prefix, the prefix that is used in the node name is changed to an arbitrary prefix, and the namespace binding is created for the arbitrary prefix.

Important: A prefix that is used in a QName must resolve to a valid URI. Otherwise, a binding for that prefix cannot be added to the in-scope namespaces of the element. If the QName cannot be resolved, the expression results in an error.

Examples

The following query includes a prolog that contains namespace declarations and a body that contains a direct element constructor:

```
SELECT XMLQUERY(
 'declare namespace p="http://example.com/ns/p";
  declare namespace q="http://example.com/ns/q";
  declare namespace f="http://example.com/ns/f";
  <p:newElement q:b="B900" xmlns:r="http://example.com/ns/r"/>')
FROM SYSIBM.SYSDUMMY1
```

The namespace declarations in the prolog add the namespace bindings to the statically known namespaces of the expression. However, the namespace bindings are added to the in-scope namespaces of the constructed element only if the QNames in the constructor use these namespaces. Therefore, the in-scope namespaces of `p:newElement` consist of the following namespace bindings:

- `p = "http://example.com/ns/p"` - This namespace binding is added to the in-scope namespaces because the prefix p appears in the QName `p:newElement`.
- `q = "http://example.com/ns/q"` - This namespace binding is added to the in-scope namespaces because the prefix q appears in the attribute QName `q:b`.
- `r = "http://example.com/ns/r"` - This namespace binding is added to the in-scope namespaces because it is defined by a namespace declaration attribute.
- `xml = "http://www.w3.org/XML/1998/namespace"` - This namespace binding is added to the in-scope namespaces because it is defined for every constructed element node.

No binding for the namespace `f="http://example.com/ns/f"` is added to the in-scope namespaces. This is because the element constructor does not include element or attribute names that use the prefix `f`. Therefore, this namespace binding does not appear in the query result, even though it is present in the statically known namespaces and is available for use during processing of the query.

The query returns the following result:

```
<p:newElement xmlns:q="http://example.com/ns/q"
              xmlns:r="http://example.com/ns/r"
              xmlns:p="http://example.com/ns/p"
              q:b="B900"/>
```

The following query demonstrates that when a namespace binding is not used to generate a query result, the namespace binding is not added to the in-scope namespaces of a constructed element.

```
SELECT XMLQUERY(
  'declare namespace p="http://example.com/ns/p";
  <newdoc>
    { $d/p:element1/p:element2 }
  </newdoc>'
  PASSING XMLPARSE(DOCUMENT
  '<p2:element1 xmlns:p2="http://example.com/ns/p">
     <p2:element2>New element</p2:element2>
   </p2:element1>')
  as "d")
FROM SYSIBM.SYSDUMMY1
```

The namespace binding `p="http://example.com/ns/p"` is not added to the in-scope namespaces of `p:element2`, even though p appears in the query.

The query returns the following result:

```
<newdoc>
  <p2:element2 xmlns:p2="http://example.com/ns/p">
    New element
  </p2:element2>
</newdoc>
```

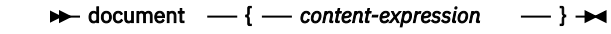
Document node constructors

All document node constructors are computed constructors. A document node constructor creates a document node for which the content of the node is computed based on an enclosed expression.

A document node constructor is useful when the result of a query is a complete document. The result of a document node constructor is a new document node that has its own node identity.

Important: No validation is performed on the constructed document node. The XQuery document node constructor does not enforce the XML 1.0 rules that govern the structure of an XML document. For example, a document node is not required to have exactly one child that is an element node.

Syntax



document
A keyword that indicates that the text that follows it constructs a document node.

content-expression
An expression that generates the content of the constructed document node. The value of *content-expression* can be any sequence of nodes and atomic values except for an attribute node. Attribute nodes in the content sequence result in an error. Document nodes in the content sequence are replaced by their children. For each node that is returned by *content-expression*, a new copy is made of the node and all of its descendants, which retain their original type annotations. Any atomic values that are returned by the content expression are converted to strings and stored in text nodes, which become children of the constructed document node. Adjacent text nodes are merged into a single text node.

Example

The following query has a document node constructor that includes a content expression that returns an XML document. The XML document contains a root element named `item-list`:

```
SELECT XMLQUERY(
  'declare namespace ipo="http://www.example.com/IPO";
  let $i := $po/ipo:purchaseOrder/items/item
  return
    document
    {
      <item-list>
       {$i}
      </item-list>
    }'
  PASSING XMLAGG(PORDER) as "po")
FROM PURCHASEORDER
```

Suppose that the PORDER column in the PURCHASEORDER table contains the following data:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
```

```
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

This query returns the following result:

```
<item-list>
  <item xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ipo="http://www.example.com/IPO" partNum="833-AA">
    <productName>Lapis necklace</productName>
    <quantity>1</quantity>
    <USPrice>99.95</USPrice>
    <ipo:comment>Want this for the holidays!</ipo:comment>
    <shipDate>2008-12-05</shipDate>
  </item>
  <item xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ipo="http://www.example.com/IPO" partNum="945-ZG">
    <productName>Sapphire Bracelet</productName>
    <quantity>2</quantity>
    <USPrice>178.99</USPrice>
    <shipDate>2009-01-03</shipDate>
  </item>
</item-list>
```

Processing instruction constructors

Processing instruction constructors create processing instruction nodes. Db2 for z/OS supports direct constructors for creating processing instruction nodes.

The constructed node has the following node properties:

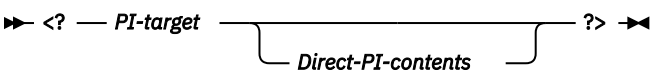
A target property
Identifies the application to which the processing instruction is directed.

A content property
Specifies the content of the processing instruction.

Direct processing instruction constructors

Direct processing instruction constructors use an XML-like notation to create processing instruction nodes.

Syntax



PI-target
An NCName that represents the name of the processing application to which the processing instruction is directed. The PI target of a processing instruction cannot consist of the characters "XML" in any combination of uppercase and lowercase.

Direct-PI-contents
A series of characters that specify the contents of the processing instruction. The contents of a processing instruction cannot contain the string ?>.

Example

The following constructor creates a processing instruction node:

```
<?format role="output" ?>
```

Comment constructors

Comment constructors create comment nodes. Db2 for z/OS supports direct constructors for creating comment nodes.

Direct comment constructors

Direct comment constructors use an XML-like notation to create comment nodes.

Syntax



direct-comment-contents
A series of characters that specify the contents of the comment. The contents of a comment cannot contain two consecutive hyphens or end with a hyphen.

Example

The following constructor creates a comment node:

```
<!-- This is an XML comment. -->
```

FLWOR expressions

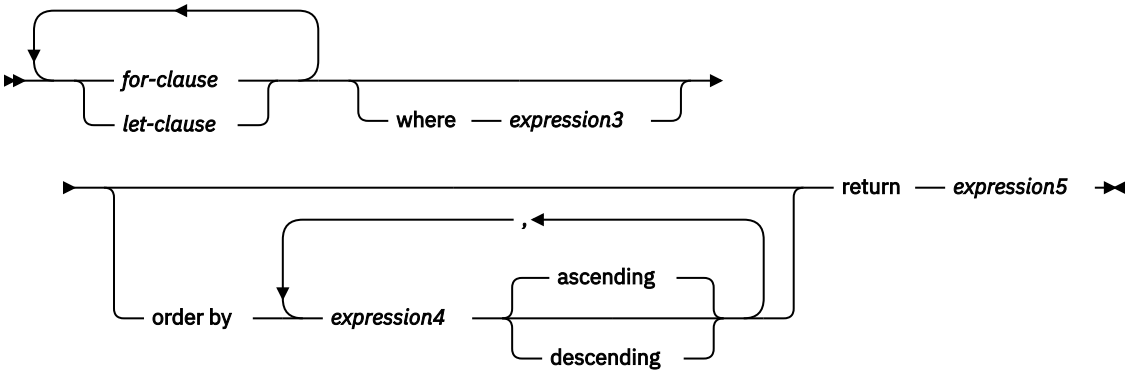
FLWOR expressions iterate over sequences and bind variables to intermediate results.

FLWOR expressions are useful for:

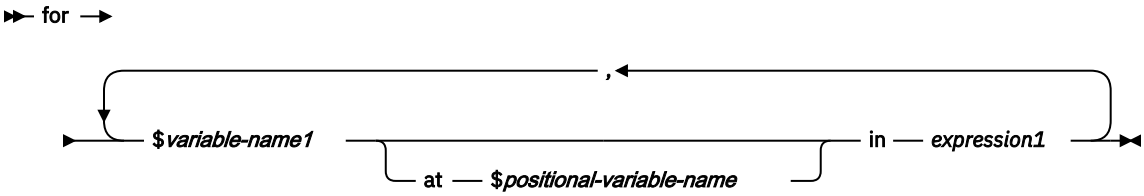
- Computing joins between two or more documents
- Restructuring data
- Sorting the results

Syntax of FLWOR expressions

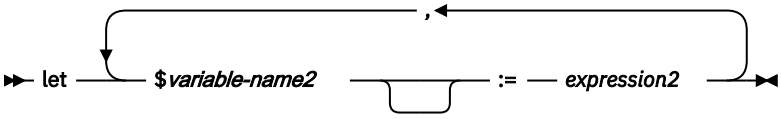
A FLWOR expression is composed of the following clauses, some of which are optional: **for**, **let**, **where**, **order by**, and **return**.



for-clause



let-clause



for
The keyword that begins a **for** clause. A **for** clause iterates over the result of *expression1* and binds *variable-name1* to each item that is returned by *expression1*.

let
The keyword that begins a **let** clause. A **let** clause binds *variable-name2* to the entire result of *expression2*.

variable-name1*, *variable-name2
The name of the variable to bind to the result of *expression1* or *expression2*.

positional-variable-name
The name of an optional variable that is bound to the position within the input stream of the item that is bound by each iteration of the **for** clause.

expression1*, *expression2*, *expression3*, *expression4*, *expression5
Any XQuery expression. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses.

where
The keyword that begins a where clause. A **where** clause filters the tuples of variable bindings that are generated by the **for** and **let** clauses.

order by
The keywords that begin an **order by** clause. An **order by** clause specifies the order in which values are processed by the return clause.

ascending
Specifies that ordering keys are processed in ascending order.

descending
Specifies that ordering keys are processed in descending order.

return
The keyword that begins a **return** clause. The expression in the **return** clause is evaluated once for each tuple of bound variables that is generated by the **for**, **let**, **where**, and **order by** clauses. The results of all of the evaluations of the **return** clause are concatenated into a single sequence, which is the result of the FLWOR expression.

for and let clauses
A **for** or **let** clause in a FLWOR expression binds one or more variables to values that are to be used in other clauses of the FLWOR expression.

for clauses
A **for** clause iterates through the result of an expression and binds a variable to each item in the sequence.

The simplest type of **for** clause contains one variable and an associated expression. In the following example, the **for** clause includes a variable called `$i` and an expression that constructs the sequence `(1, 2, 3)`:

```
SELECT XMLQUERY(
  'for $i in (1, 2, 3)
  return <output>{$i}</output>')
FROM SYSIBM.SYSDUMMY1
```

When the **for** clause is evaluated, three variable bindings are created (one binding for each item in the sequence):

```
$i = 1
$i = 2
$i = 3
```

The **return** clause in the example executes once for each binding. The SQL statement returns a column with the following row:

```
<output>1</output><output>2</output><output>3</output>
```

A **for** clause can contain multiple variables, each of which is bound to the result of an expression. In the following example, a **for** clause contains two variables, $a and $b, and expressions that construct the sequences 1 2 and 4 5:

```
SELECT XMLQUERY(
  'for $a in (1, 2), $b in (4, 5)
  return <output>{$a, $b}</output>'
  COLUMNS "x" XML PATH '.')
  FROM SYSIBM.SYSDUMMY1
```

When the **for** clause is evaluated, a tuple of variable bindings is created for each combination of values. This results in four tuples of variable bindings:

```
($a = 1, $b = 4)
($a = 2, $b = 4)
($a = 1, $b = 5)
($a = 2, $b = 5)
```

The **return** clause in the example executes once for each tuple of bindings. The SQL statement returns a column with the following rows:

```
<output>1 4</output>
<output>2 4</output>
<output>1 5</output>
<output>2 5</output>
```

When the binding expression evaluates to an empty sequence, no **for** binding is generated, and no iteration is performed. In the following example, the binding sequence evaluates to an empty sequence and no iteration is performed. The node sequence in the **return** clause is not returned.

```
SELECT XMLQUERY(
  'for $node in
  (<a test="b" />,
   <a test="c" />,
   <a test="d" />)[@test = "1"]
  return
  <test>
    Sample return response
  </test>')
  FROM SYSIBM.SYSDUMMY1
```

Positional variables in `for` clauses

Each variable that is bound in a **for** clause can have an associated positional variable that is bound at the same time. The name of the positional variable is preceded by the keyword **at**. When a variable iterates over the items in a sequence, the positional variable iterates over the integers that represent the positions of those items in the sequence, starting with 1. You can reference the positional variables in the same way that you reference any other variables.

In the following example, the **for** clause includes a variable called `$cat` and an expression that constructs the sequence `("Persian", "Calico", "Siamese")`. The clause also includes the positional variable `$i`, which is referenced in an attribute constructor to compute the value of the `order` attribute:

```
SELECT XMLQUERY(
  'for $cat at $i in
  ("Persian", "Calico","Siamese")
  return <cat order="{$i}">{$cat}</cat>')
  FROM SYSIBM.SYSDUMMY1
```

When the **for** clause is evaluated, three tuples of variable bindings are created, each of which includes a binding for the positional variable:

```
($i = 1, $cat = "Persian")
($i = 2, $cat = "Calico")
($i = 3, $cat = "Siamese")
```

The **return** clause in the example executes once for each tuple of bindings. The expression results in the following output:

```
<cat order="1">Persian</cat><cat order="2">Calico</cat><cat order="3">Siamese</cat>
```

Although each output element contains an order attribute, the actual order of the elements in the output stream is not guaranteed unless the FLWOR expression contains an **order by** clause such as `order by $i`. The positional variable represents the ordinal position of a value in the input sequence, not in the output sequence.

`let` clauses

A **let** clause binds a variable to the entire result of an expression. A **let** clause does not perform any iteration.

The simplest type of **let** clause contains one variable and an associated expression. In the following example, the **let** clause includes a variable called `$j` and an expression that constructs the sequence `(1, 2, 3)`.

```
SELECT XMLQUERY(
  'let $j := (1, 2, 3)
  return <output>{$j}</output>')
  FROM SYSIBM.SYSDUMMY1
```

When the **let** clause is evaluated, a single binding is created for the entire sequence that results from evaluating the expression:

```
$j = 1 2 3
```

The **return** clause in the example executes once. The SELECT statement results in the following output:

```
<output>1 2 3</output>
```

A **let** clause can contain multiple variables. However, unlike a **for** clause, a **let** clause binds each variable to the result of its associated expression, without iteration. In the following example, a **let** clause contains two variables, $a and $b, and expressions that construct the sequences 1 2 and 4 5:

```
SELECT XMLQUERY(
  'let $a := (1,2), $b := (4,5)
  return <output>{$a,$b}</output>')
  FROM SYSIBM.SYSDUMMY1
```

When the **let** clause is evaluated, one tuple of variable bindings is created:

```
($a = 1 2, $b = 4 5)
```

The **return** clause in the example executes once for the tuple. The expression results in the following output:

```
<output>1 2 4 5</output>
```

When the binding expression evaluates to an empty sequence, a **let** binding is created that contains the empty sequence.

for and let clauses in the same expression

When a FLWOR expression contains **for** and **let** clauses, the variable bindings for the **let** clauses are added to the variable bindings for the **for** clauses.

In the following example, the **for** clause includes a variable called $a and an expression that constructs the sequence `(1, 2, 3)`. The **let** clause includes a variable called $b and an expression that constructs the sequence `(4, 5, 6)`:

```
SELECT XMLQUERY(
  'for $a in (1, 2, 3)
  let $b := (4, 5, 6)
  return <output>{$a, $b}</output>')
  FROM SYSIBM.SYSDUMMY1
```

The **for** and **let** clauses in this example result in three tuples of bindings. The number of tuples is determined by the **for** clause.

```
($a = 1, $b = 4 5 6)
($a = 2, $b = 4 5 6)
($a = 3, $b = 4 5 6)
```

The **return** clause in the example executes once for each tuple of bindings. The SELECT statement results in the following output:

```
<output>1 4 5 6</output><output>2 4 5 6</output><output>3 4 5 6</output>
```

Variable scope in for and let clauses

A variable that is bound in a **for** or **let** clause is in scope for the sub-expressions that appear after the variable binding.

This means that a **for** or **let** clause can reference variables that are bound in earlier clauses or in earlier bindings in the same clause.

The following examples demonstrate variable scope in **for** or **let** clauses. They assume that the PORDER column in the PURCHASEORDER table contains the following data:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

The following example shows that a variable can be bound in a clause and referenced in a later clause. In the SELECT statement, the FLWOR expression has the following clauses:

- A **let** clause that binds the variable `$item`.
- A **for** clause that references `$item` and binds the variable $pn.
- A **let** clause that references both `$item` and $pn and binds the variables $n and $q.

```
SELECT XMLQUERY(
  'declare namespace ipo="http://www.example.com/IPO";
  let $item := $po/ipo:purchaseOrder/items/item
    for $pn in $item/@partNum
      let $n := $item[@partNum = $pn]/productName,
        $q := $item[@partNum = $pn]/fn:sum(quantity)
      return fn:concat("Quantity of ", $n, " = ", $q, ", ")'
  PASSING PORDER as "po")
  FROM PURCHASEORDER
```

The SELECT statement results in the following output:

```
Quantity of Lapis necklace = 1, Quantity of Sapphire Bracelet = 2,
```

The following example shows that two variables with the same name can be bound and referenced within the same scope. In the SELECT statement, the FLWOR expression has the following clauses:

- A **let** clause that binds a variable named `$iteration` to 0.
- A **for** clause that references a variable named $po that was bound by the PASSING clause, and binds a new $po variable.

- A **let** clause that references the $iteration variable from the first **let** clause, and binds another variable named $iteration.

```
SELECT XMLQUERY(
  'declare namespace ipo="http://www.example.com/IPO";
  let $iteration := 0
  return
    for $po in $po/ipo:purchaseOrder/items/item/USPrice
    return
      let $iteration := $iteration + 1
      return
        <test iteration="{$iteration}">{$po}</test>'
  PASSING PORDER as "po") FROM PURCHASEORDER
```

The SELECT statement results in the following output:

```
<test iteration="1">
  <USPrice xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:ipo="http://www.example.com/IPO">
    99.95
  </USPrice>
</test>
<test iteration="1">
  <USPrice xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:ipo="http://www.example.com/IPO">
    178.99
  </USPrice>
</test>
```

where clauses

A **where** clause in an FLWOR expression filters the tuples of variable bindings that are generated by the **for** and **let** clauses.

The **where** clause specifies a condition that is applied to each tuple of variable bindings. If the condition is true (that is, if the expression results in an effective Boolean value of true), the tuple is retained, and its bindings are used when the **return** clause executes. Otherwise, the tuple is discarded.

In the following example, the **for** clause binds the variables $x and $y to sequences of numeric values:

```
SELECT XMLQUERY(
  'for $x in (1.5, 2.6, 1.9), $y in (.5, 1.6, 1.7)
  where ((fn:floor($x) eq 1) and (fn:floor($y) eq 1))
  return <output>{$x, $y}</output>')
  FROM SYSIBM.SYSDUMMY1
```

When the **for** clause is evaluated, nine tuples of variable bindings are created:

```
($x = 1.5, $y = .5)
($x = 2.6, $y = .5)
($x = 1.9, $y = .5)
($x = 1.5, $y = 1.6)
($x = 2.6, $y = 1.6)
($x = 1.9, $y = 1.6)
($x = 1.5, $y = 1.7)
($x = 2.6, $y = 1.7)
($x = 1.9, $y = 1.7)
```

The **where** clause filters these tuples, and the following tuples are retained:

```
($x = 1.5, $y = 1.6)
($x = 1.9, $y = 1.6)
($x = 1.5, $y = 1.7)
($x = 1.9, $y = 1.7)
```

The **return** clause executes once for each of the retained tuples. The SELECT statement returns a single row with the follow contents:

```
<output>1.5 1.6</output><output>1.9 1.6</output><output>1.5 1.7</output>
<output>1.9 1.7</output>
```

Because the expression in this example does not include an **order by** clause, the order of the output elements is non-deterministic.

order by clauses

An **order by** clause in an FLWOR expression specifies the order in which values are to be processed by the **return** clause.

An **order by** clause contains one or more ordering specifications. Ordering specifications are used to reorder the tuples of variable bindings that are retained after being filtered by the **where** clause. The resulting order determines the order in which the **return** clause is evaluated.

Each ordering specification consists of an expression, which is evaluated to produce an ordering key, and an order modifier, which specifies the sort order (ascending or descending) for the ordering keys. The relative order of two tuples is determined by comparing the values of their ordering keys, working from left to right.

In the following example, an FLWOR expression includes an **order by** clause that sorts products in descending order based on their price:

```
SELECT XMLQUERY(
  'declare namespace ipo="http://www.example.com/IPO";
  for $i in $po/ipo:purchaseOrder/items/item
  order by xs:decimal($i/USPrice) descending
  return fn:concat($i/productName, ":US$", $i/USPrice)'
  PASSING PORDER as "po")
FROM PURCHASEORDER
```

Suppose that the PORDER column of the PURCHASEORDER table contains this data:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

During processing of the **order by** clause, the expression in the ordering specification is evaluated for each tuple that is generated by the **for** clause. For the first tuple, the value that is returned by the

expression `xs:decimal($i/USPrice)` is 99.95. The expression is then evaluated for the next tuple, and the expression returns the value 178.99. Because the ordering specification indicates that items are sorted in descending order, the product with the price 99.95 sorts before the product with the price 178.99. This sorting process continues until all tuples are reordered. The **return** clause then executes once for each tuple in the reordered tuple stream.

The query in the example returns the following result:

```
Sapphire Bracelet:US$178.99 Lapis necklace:US$99.95
```

In the example, the expression in the ordering specification constructs an xs:decimal value from the value of the USPrice element. This type conversion is necessary because the type annotation of the USPrice element in the XML schema is xs:untypedAtomic. Without this conversion, the result would use string ordering rather than numeric ordering.

Explicit type conversion is also required when the dynamic type of the ordering key value is xs:untypedAtomic because the rules for comparing ordering keys dictate that untyped atomic data is treated as a string.

Tip: You can use an **order by** clause in an FLWOR expression to specify value ordering in a query that would otherwise not require iteration. For example, the following path expression returns a list of customerinfo elements with a customer ID (Cid) that is greater than 1000:

```
$ci/customerinfo[@Cid > "1000"]
```

To return these items in ascending order by the name of the customer, however, you need to specify an FLWOR expression that includes an **order by** clause:

```
SELECT XMLQUERY(
  'declare default element namespace "http://posample.org";
  for $custinfo in $ci/customerinfo
  where ($custinfo/@Cid > 1000)
  order by $custinfo/name ascending
  return $custinfo'
  PASSING XMLAGG(INFO) as "ci")
FROM CUSTOMER
```

The ordering key does not need to be part of the output. The following query produces a list of product names, in descending order by price, but does not include the price in the output:

```
SELECT XMLQUERY(
  'declare namespace ipo="http://www.example.com/IPO";
  for $i in $po/ipo:purchaseOrder/items/item
  order by xs:decimal($i/USPrice) descending
  return $i/productName'
  PASSING PORDER as "po")
FROM PURCHASEORDER
```

Rules for comparing ordering specifications

The process of evaluating and comparing ordering specifications is based on the following rules:

- The expression in the ordering specification is evaluated and the result is converted to an atomic value. The result of the conversion must be a single atomic value or an empty sequence; otherwise an error is returned. The result of evaluating an ordering specification is called an ordering key.
- If the type of an ordering key is xs:untypedAtomic, that key is cast to the type xs:string.
- If the values that are generated by an ordering specification are not all of the same type, those values (keys) are converted to a common type by subtype substitution or type promotion. Keys are compared by converting them to the least common type that supports the **gt** operator. If the ordering keys that are generated by a given ordering specification do not have a common type that supports the **gt** operator, an error results.
- The values of the ordering keys are used to determine the order in which tuples of bound variables are passed to the return clause for execution. The ordering of tuples is determined by comparing their ordering keys, from left to right, by using the following rules:

– If the sort order is ascending, tuples with ordering keys that are greater than other tuples sort after those tuples.
– If the sort order is descending, tuples with ordering keys that are greater than other tuples sort before those tuples.

The greater-than relationship for ordering keys is defined as follows:

– An empty sequence is greater than all other values.
– NaN is greater than all other values except the empty sequence.
– A value is greater than another value if, when the value is compared to another value, the **gt** operator returns true.
– Neither of the special floating-point values positive zero or negative zero is greater than the other because `+0.0 gt -0.0` and `-0.0 gt +0.0` are both false.

`return` clauses

A **return** clause generates the result of the FLWOR expression.

The **return** clause is evaluated once for each tuple of variable bindings that is generated by the other clauses of the FLWOR expression. The order in which tuples of bound variables are processed by the **return** clause is non-deterministic unless the FLWOR expression contains an **order by** clause.

Tip: In **return** clauses, use parentheses to enclose expressions that contain top-level comma operators. Because FLWOR expressions have a higher precedence than the comma operator, expressions that contain top-level comma operators can result in errors or unexpected results if parentheses are not used.

FLWOR examples

You can use FLWOR expressions in complete queries to perform joins, grouping, and aggregation.

FLWOR expression that joins XML data

The following query uses an XQuery FLWOR expression to express a join.

Suppose that the PORDER column of the PURCHASEORDER table contains this data:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
```

```
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

Also suppose that a table named STATUS has an XML column named STATUS, which has the following data:

```
<status>
  <statusItem>
    <name>Robert Smith</name>
    <status>Premier</status>
    <comment>Orders a lot of jewelry</comment>
    <comment>Has friends in the Silicon Valley</comment>
  </statusItem>
  <statusItem>
    <name>Jane Carmody</name>
    <status>Unreliable</status>
    <comment>Has unpaid bills</comment>
  </statusItem>
</status>
```

The following example shows how to use an XQuery FLWOR expression to find those purchase orders that were made by customers with Premier status.

```
SELECT XMLQUERY(
  'declare namespace ipo="http://www.example.com/IPO";
  for $i in $po/ipo:purchaseOrder
  return
    <premierOrders> {
      for $j in $status/status/statusItem
      where $j/name=$i/billTo/name and $j/status="Premier"
      return
      $i }
    </premierOrders>'
  PASSING T1.PORDER as "po", T2.STATUS as "status")
  FROM PURCHASEORDER T1, STATUS T2;
```

The query returns the following result:

```
<premierOrders>
  <ipo:purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ipo="http://www.example.com/IPO" orderDate="1999-12-01">
    <shipTo exportCode="1" xsi:type="ipo:UKAddress">
      <name>Helen Zoe</name>
      <street>55 Eden Street</street>
      <city>San Jose</city>
      <state>CA</state>
      <postcode>CB1 1JR</postcode>
    </shipTo>
    <shipTo exportCode="1" xsi:type="ipo:UKAddress">
      <name>Joe Lee</name>
      <street>66 University Avenue</street>
      <city>Palo Alto</city>
      <state>CA</state>
      <postcode>CB1 1JR</postcode>
    </shipTo>
    <billTo xsi:type="ipo:USAddress">
      <name>Robert Smith</name>
      <street>8 Oak Avenue</street>
      <city>Old Town</city>
      <state>PA</state>
      <zip>95819</zip>
    </billTo>
    <items>
      <item partNum="833-AA">
        <productName>Lapis necklace</productName>
        <quantity>1</quantity>
        <USPrice>99.95</USPrice>
        <ipo:comment>Want this for the holidays!</ipo:comment>
        <shipDate>2008-12-05</shipDate>
      </item>
      <item partNum="945-ZG">
        <productName>Sapphire Bracelet</productName>
```

```
        <quantity>2</quantity>
        <USPrice>178.99</USPrice>
        <shipDate>2009-01-03</shipDate>
      </item>
    </items>
  </ipo:purchaseOrder>
</premierOrders>
```

FLWOR expression that uses conditional logic

The following example shows how to use XQuery to build a report of revenue for items in the PORDER column of the PURCHASEORDER table that have already shipped.

```
SELECT XMLQUERY(
    'let $j := (
       for $i in $po//item
       return if (xs:date($i/shipDate) <= $currentDate)
          then xs:decimal($i/USPrice)
          else 0.0 )
    return fn:sum($j)'
  PASSING T1.PORDER as "po", CURRENT DATE as "currentDate")
FROM PURCHASEORDER T1;
```

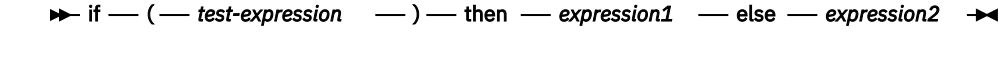
Suppose that the current date is 2009-01-01. The query returns the following result:

```
99.95
```

Conditional expressions

Conditional expressions use the keywords **if**, **then**, and **else** to evaluate one of two expressions based on whether the value of a test expression is true or false.

Syntax



if
The keyword that directly precedes the test expression.

test-expression
An XQuery expression that determines which part of the conditional expression to evaluate.

then
If the effective Boolean value of *test-expression* is true, then the expression that follows this keyword is evaluated. The expression is not evaluated or checked for errors if the effective Boolean value of the test expression is false.

else
If the effective Boolean value of *test-expression* is false, then the expression that follows this keyword is evaluated. The expression is not evaluated or checked for errors if the effective Boolean value of the test expression is true.

expression1 and expression2
Any XQuery expression. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses.

If either the **then** or **else** condition branch contains an updating expression, then the conditional expression is an updating expression. An updating expression must be within the **modify** clause of a transform expression.

For an updating conditional expression, each branch must contain either an updating expression or an empty sequence. Based on the value of the test expression, either the **then** or **else** clause is selected and evaluated. The result of the conditional updating expression is a list of updates returned by the selected branch. The containing transform expression performs the updates after merging them with updates returned by other updating expressions within the **modify** clause of the transform expression.

Restriction: *test-expression, expression1, and expression2* cannot contain an FLWOR expression.

Example

Suppose that the PORDER column in the PURCHASEORDER table contains the following data:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

In the following example, the query constructs a list of `item` elements. The value of the `shipping` element for an item is specified conditionally, based on whether the value of the USPrice element is less than 100. In this example, the test expression constructs an xs:decimal value from the value of the USPrice element. The xs:decimal function is used to force a decimal comparison.

```
SELECT XMLQUERY(
  'declare namespace ipo="http://www.example.com/IPO";
  for $i in $po/ipo:purchaseOrder/items/item
  return
  <item>
   {$i/productName}
   <shipping>
     { if (xs:decimal($i/USPrice) lt 100) then 5 else 10 }
   </shipping>
  </item>'
  PASSING PORDER as "po")
  FROM PURCHASEORDER
```

The query returns the following result:

```
<item>
  <productName
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ipo="http://www.example.com/IPO">
    Lapis necklace
  </productName>
  <shipping>5</shipping>
</item>
```

```
<item>
  <productName
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ipo="http://www.example.com/IPO">
    Sapphire Bracelet
  </productName>
  <shipping>10</shipping>
</item>
```

Basic updating expressions

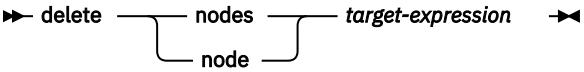
Using the basic XQuery updating expressions, you can create complex updating expressions to update existing XML data.

Basic updating expressions are valid only in the *xquery-update-constant* argument of the SQL XMLMODIFY function.

Delete expression

A delete expression deletes zero or more nodes from a node sequence.

Syntax



delete nodes or delete node
The keywords that begin a delete expression. **`delete nodes`** or **`delete node`** is valid, regardless of the number of nodes that are to be deleted.

target-expression
An XQuery expression that is not an updating expression. The result of *target-expression* must be a sequence of zero or more nodes.

The result of the delete expression is a list of nodes that are to be deleted. Any node that matches *target-expression* is marked for deletion. If no nodes match *target-expression*, no nodes are deleted. The deleted nodes are detached from their parent nodes. The nodes and the nodes' children are no longer part of the node sequence. An error is returned if a node's parent property is empty.

If insertion of nodes results in adjacent text nodes with the same parent, the adjacent text nodes are merged into a single text node. The string value of the resulting text node is the concatenation of the string values of the adjacent text nodes, with no spaces added between string values.

Example

Suppose that a purchaseOrder document looks like this:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
```

```
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

The following SQL UPDATE statement uses a basic updating expression to delete the item whose productName value is "Lapis necklace" from the purchaseOrder document.

```
UPDATE purchaseOrders
 SET PO = XMLMODIFY(
 'declare namespace ipo="http://www.example.com/IPO";
 delete nodes /ipo:purchaseOrder/items/item[productName="Lapis necklace"]')
```

The result of the statement is:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

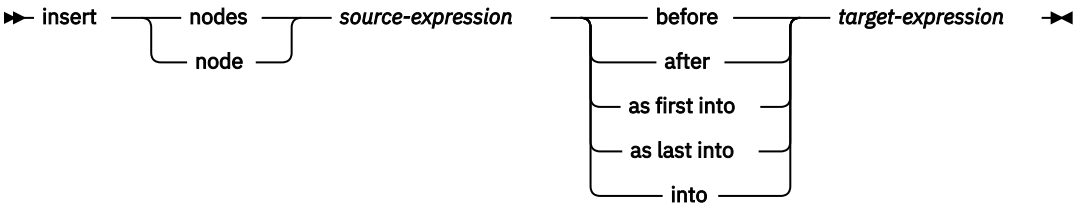
Related reference

XMLMODIFY (Db2 SQL)

Insert expression

An insert expression inserts copies of one or more nodes into a designated position in a node sequence.

Syntax



insert nodes or insert node
The keywords that begin an insert expression. **`insert nodes`** or **`insert node`** is valid, regardless of the number of nodes that are to be inserted.

source-expression
An XQuery expression that is not an updating expression. The result of the *source-expression* is a sequence of zero or more nodes that are to be inserted. Those nodes are called the *insertion sequence*. If the insertion sequence contains a document node, the document node is replaced in the insertion sequence by its children. If the insertion sequence contains an adjacent sequence of one or more atomic values, the atomic values are replaced in the insertion sequence with a new text node that contains the result of casting each atomic value to a string, and inserting a single space character between adjacent atomic values.

If the insertion sequence contains attribute nodes that appear first in the sequence, the attributes are added to the *target-expression* node or to its parent, depending on the keyword that is specified. If the insertion sequence contains an attribute node that follows a node that is not an attribute node, Db2 returns an error.

before
Keyword that specifies that the *source-expression* nodes become the preceding siblings of the *target-expression* node.

If multiple nodes are inserted before *target-expression*, the nodes remain adjacent and their order is preserved. If the insertion sequence contains attribute nodes, the attribute nodes become attributes of the parent of the target node.

after
Keyword that specifies that the *source-expression* nodes become the following siblings of the *target-expression* node.

If multiple nodes are inserted after *target-expression*, the nodes remain adjacent and their order is preserved. If the insertion sequence contains attribute nodes, the attribute nodes become attributes of the parent of the target node.

as first into
Keywords that specify that the *source-expression* nodes become the first children of the *target-expression* node.

The *source-expression* nodes are inserted as the first children of the *target-expression* node. If multiple nodes are inserted as the first children of the *target-expression* node, the nodes remain adjacent and their order is preserved. If the insertion sequence contains attribute nodes, the attribute nodes become attributes of the target node.

as last into
Keywords that specify that the *source-expression* nodes become the last children of the *target-expression* node.

The *source-expression* nodes are inserted as the last children of the *target-expression* node. If multiple nodes are inserted as the last children of the *target-expression* node, the nodes remain adjacent and their order is preserved. If the insertion sequence contains attribute nodes, the attribute nodes become attributes of the target node.

into
Has the same behavior as **`as last into`**.

target-expression

An XQuery expression that is not an updating expression. If the result of *target-expression* is an empty sequence, an error is returned. The following rules apply to *target-expression*:

- If the **before** or **after** keyword is specified:
  - The result of *target-expression* must be a single element, text, processing instruction, or comment node whose parent property is not empty.
  - If the insertion sequence contains an attribute node, the parent of the *target-expression* node must be an element node.
  - The parent property of the *target-expression* node cannot be empty.
  - If an attribute node in the insertion sequence has a qualified name (QName) with an implied namespace binding, the namespaces property of the parent node of the *target-expression* node is modified to include a namespace binding for any attribute namespace prefixes that do not already have bindings. If the implied namespace binding conflicts with a namespace binding in the namespaces property of the parent node of the *target-expression* node, an error is returned.

  If the parent of the *target-expression* node is a document node and **before** or **after** is specified, the insertion sequence cannot contain an attribute node.
- If the **into**, **as first into**, or **as last into** keywords are specified:
  - The result of *target-expression* must be a single element, text, processing instruction, or comment node whose parent property is not empty.
  - If the insertion sequence contains an attribute node, the *target-expression* node cannot be a document node.
  - The parent property of the *target-expression* node cannot be empty.
  - If an attribute node in the insertion sequence has a qualified name (QName) with an implied namespace binding, the namespaces property of the *target-expression* node is modified to include a namespace binding for any attribute namespace prefixes that do not already have bindings. If the implied namespace binding conflicts with a namespace binding in the namespaces property of the *target-expression* node, an error is returned.

If insertion of nodes results in adjacent text nodes with the same parent, the adjacent text nodes are merged into a single text node. The string value of the resulting text node is the concatenation of the string values of the adjacent text nodes, with no spaces added between string values.

If the result of the insert violates any constraint of the XPath 2. 0 and XQuery 1.0 data model, an error is returned.

Example

Suppose that a purchaseOrder document looks like this:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
```

```
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

The following SQL UPDATE statement uses a basic updating expression to insert a new `item` node as the first node under the `items` node in the purchaseOrder document.

```
UPDATE T1
  SET X1 = XMLMODIFY(
  'declare namespace ipo="http://www.example.com/IPO";
   insert nodes $item as first
   into /ipo:purchaseOrder/items',
   XMLPARSE(DOCUMENT
   '<item partNum="747-BB">
    <productName>Ruby Ring</productName>
    <quantity>1</quantity>
    <USPrice>75.50</USPrice>
    <shipDate>2007-05-13</shipDate>
    </item>') as "item")
```

The result of the statement is:

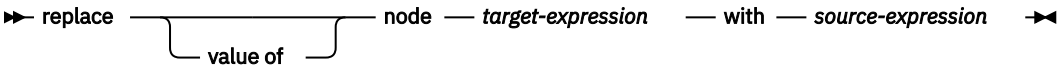
```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="747-BB">
     <productName>Ruby Ring</productName>
     <quantify>1</quantity>
     <USPrice>75.50</USPrice>
     <shipDate>2007-05-13</shipDate>
    </item>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
```

```
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

Related reference
XMLMODIFY (Db2 SQL)

Replace expression
A replace expression replaces an existing node with a new sequence of zero or more nodes, or replaces a node's value while preserving the node's identity.

Syntax



replace
The keyword that begins a replace expression.

value of
The keywords that specify replacing the value of the *target-expression* node that is to be replaced.

node
The keyword that begins the target expression.

target-expression
An XQuery expression that is not an updating expression. The result of *target-expression* must be a single node that is not a document node. If the result of *target-expression* is an empty sequence, an error is returned.

If the **value of** keywords are not specified, the result of *target-expression* must be a single node whose parent property is not empty.

with
The keyword that begins the source expression.

source-expression
An XQuery expression that is not an updating expression.

If the **value of** keywords are specified, the result of *source-expression* is a single text node or an empty sequence. During processing, atomization is applied to *source-expression*, to convert it to a sequence of atomic values. If the result of atomization is an empty sequence, the result of *source-expression* is an empty sequence. Otherwise, each atomic value in the atomized sequence is cast to a string. All of the strings are concatenated, with a single space character between each pair of strings.

If the **value of** keywords are not specified, the result of *source-expression* must be a sequence of nodes. If the *source-expression* sequence contains a document node, the document node is replaced by its children. If *source-expression* contains an adjacent sequence of one or more atomic values, a new text node is constructed containing the result of casting each atomic value to a string, with a single space character inserted between adjacent values. The *source-expression* sequence must consist of the following node types:

- If the *target-expression* node is an attribute node, the replacement sequence must consist of zero or more attribute nodes.
- If the *target-expression* node is an element, text, comment, or processing instruction node, the replacement sequence must consist of some combination of zero or more element, text, comment, or processing instruction nodes.

The following updates are generated when the **value of** keywords are specified:

- If the *target-expression* node is an element node, the existing children of the *target-expression* node are replaced by the text node returned by the *source-expression*. If the *source-expression* returns an empty

sequence, the children property of the *target-expression* node becomes empty. If the *target-expression* node contains attribute nodes, they are not affected.

- If the *target-expression* node is not an element node, the string value of the *target-expression* node is replaced by the string value of the text node that is returned by the *source-expression*. If the *source-expression* does not return a text node, the string value of the *target-expression* node is replaced by a zero-length string.

  If the *target-expression* node is a comment node, and if the string value of the text node that is returned by *source-expression* contains two adjacent hyphens or ends with a hyphen, an error is returned.

  If the *target-expression* node is a processing instruction node, and if the string value of the text node that is returned by *source-expression* contains the substring "?>", an error is returned.

The following updates are generated when the **value of** keywords are not specified:

- *source-expression* nodes replace the *target-expression* node. The parent node of the *target-expression* node becomes the parent of each of the *source-expression* nodes. The *source-expression* nodes occupy the position in the node hierarchy that is occupied by the *target-expression* node.
- The *target-expression* node, and all of its attributes and descendants are detached from the node sequence.
- If *target-expression* has an attribute node, and an attribute node in the replacement sequence has a qualified name (QName) with an implied namespace binding, the `namespaces` property of the parent node of the *target-expression* node is modified to include a namespace binding for any attribute namespace prefixes that do not already have bindings. If the implied namespace binding conflicts with a namespace binding in the `namespaces` property of the parent node of the *target-expression* node, an error is returned.

If replacement of nodes results in adjacent text nodes with the same parent, the adjacent text nodes are merged into a single text node. The string value of the resulting text node is the concatenation of the string values of the adjacent text nodes, with no spaces added between string values.

If the result of the replace violates any constraint of the XPath 2. 0 and XQuery 1.0 data model, an error is returned.

Example

Suppose that a purchaseOrder document looks like this:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
```

```
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

The following SQL UPDATE statement uses a basic updating expression to replace the value of the `street` node in the `billTo` node of the purchaseOrder document with a new value.

```
UPDATE PURCHASEORDER
 SET PORDER = XMLMODIFY(
 'declare namespace ipo="http://www.example.com/IPO";
 replace value of node /ipo:purchaseOrder/billTo/street
 with "505 First Street"')
```

The result of the statement is:

```
<ipo:purchaseOrder
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ipo="http://www.example.com/IPO"
   orderDate="2008-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>55 Eden Street</street>
    <city>San Jose</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Joe Lee</name>
    <street>66 University Avenue</street>
    <city>Palo Alto</city>
    <state>CA</state>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>505 First Street</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>2008-12-05</shipDate>
    </item>
    <item partNum="945-ZG">
      <productName>Sapphire Bracelet</productName>
      <quantity>2</quantity>
      <USPrice>178.99</USPrice>
      <shipDate>2009-01-03</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

Related reference

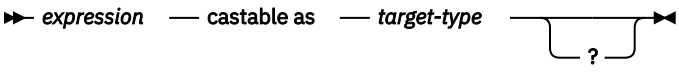
XMLMODIFY (Db2 SQL)

Castable expressions

Castable expressions test whether a value can be cast to a specific data type. If the value can be cast to the data type, the castable expression returns true. Otherwise, the expression returns false.

Castable expressions can be used as predicates to avoid cast errors at evaluation time. They can also be used to select an appropriate type for processing a value.

Syntax



expression
An XQuery expression that returns a single atomic value or an empty sequence.

target-type
The type used to test if the value of *expression* can be cast. *target-type* must be an atomic type that is one of the predefined XML schema types. The data types xs:anyAtomicType and xs:anySimpleType are not valid types for *target-type*.

?
Indicates that an empty sequence is considered to be a valid instance of the target type. If *expression* evaluates to an empty sequence and ? is not specified, the castable expression returns `false`.

Returned value

If *expression* can be cast to *target-type*, the castable expression returns `true`. Otherwise, the expression returns `false`.

If the result of *expression* is an empty sequence, and the question mark indicator follows *target-type*, the castable expression returns `true`. In the following example, the question mark indicator follows the target type xs:integer.

```
$prod/revision castable as xs:integer?
```

An error is returned in the following cases:

- The result of *expression* is a sequence of more than one atomic value.
- *target-type* is not an atomic data type that is defined for the in-scope XML schema types or is a data type that cannot be used in a castable expression.

Examples

The following example uses a castable expression as a predicate to avoid errors at evaluation time. The example avoids a dynamic error if `@OrderDate` is not a valid date.

```
let $i := if ( $po/@OrderDate castable as xs:date)
              then xs:date($po/@OrderDate) gt xs:date("2009-01-01")
              else 0
return  $po/orderID[$i]
```

The predicate is true and returns the `orderID` only if the date attribute is a valid date greater than January 1, 2009. Otherwise, the predicate is false and returns an empty sequence.

The following example uses a castable expression to select an appropriate type for processing of a given value. The example casts a postal code as either an integer or a string.

```
if ($postalcode castable as xs:integer)
   then xs:integer($postalcode)
   else xs:string($postalcode)
```

The following example uses a castable expression in the FLWOR **`let`** clause to test the value of `$prod/mfgdate` and bind a value to `$currdate`. The castable expression supports processing an empty sequence using the question mark indicator.

```
let $currdate := if ($prod/mfgdate castable as xs:date?)
   then xs:date($prod/mfgdate)
   else xs:date("2000-01-01")
```

If the value of `$prod/mfgdate` can be cast as xs:date, it is cast to the data type and is bound to `$currdate`. If `$prod/mfgdate` is an empty sequence, an empty sequence is bound to `$currdate`. If

$prod/mfgdate cannot be cast as xs:date, a value of 2000-01-01 of type xs:date is bound to $currdate.

In the following example, the castable expression in the XMLEXISTS predicate tests the value of /prod/category before performing a comparison to avoid errors at evaluation time. In the XML column FEATURES.INFO, the documents contain the element /prod/category. The value is either a numeric code or string code.

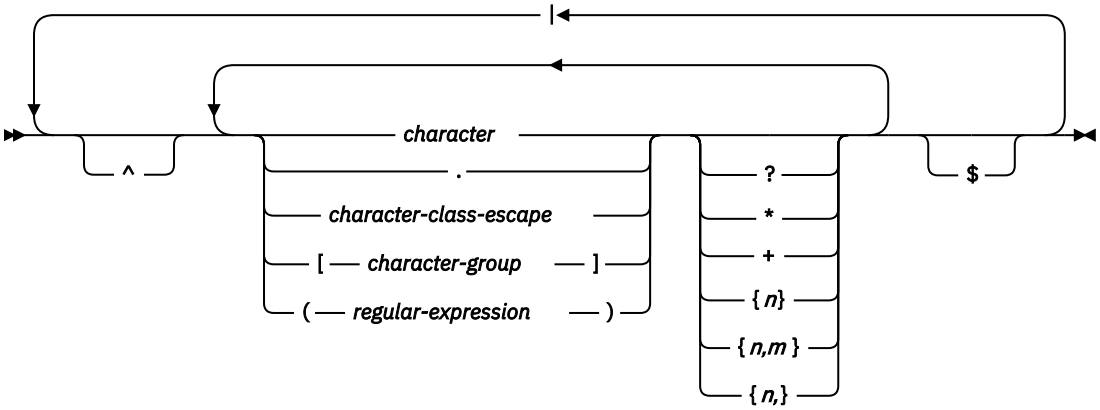
```
SELECT F.PRODID FROM F FEATURES
  WHERE XMLEXISTS('$test/prod/category[ (( . castable as xs:double) and . > 100 ) or
  (( . castable as xs:string) and . > "A100" )]'
  PASSING F.INFO as "test")
```

The returned values are product IDs where the category codes are either greater than the number 100 or greater than the string "A100".

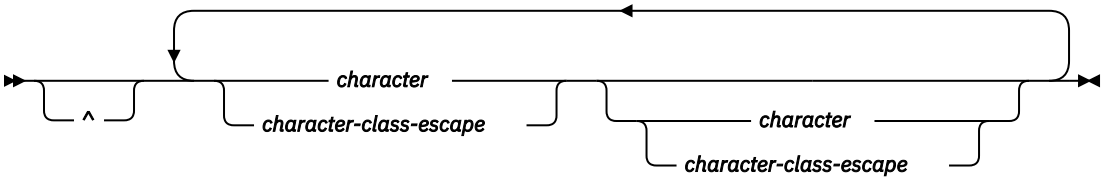
Regular expressions

A regular expression is a sequence of characters that act as a pattern for matching and manipulating strings. Regular expressions are used in the fn:matches, fn:replace, and fn:tokenize functions.

Syntax



character-group



character

In a regular expression, *character* is a normal XML character that is not a metacharacter.

Metacharacters

Metacharacters are control characters in regular expressions. The regular expression metacharacters that are currently supported are:

backslash (\)

Begins a character class escape. A character class escape indicates that the metacharacter that follows is to be used as a character, instead of a metacharacter.

period (.)

Matches any single character except a newline character (\n).

carat (^)

If the carat character appears outside of a character class, the characters that follow the carat match the start of the input string or, for multi-line input strings, the start of a line. An input string

is considered to be a multi-line input string if the function that uses the input string includes the `m` flag.

If the carat character appears as the first character within a character class, the carat acts as a not-sign. A match occurs if none of the characters in the character group appear in the string that is being compared to the regular expression.

dollar sign ($)
Matches the end of the input string or, for multi-line input strings, the end of a line. An input string is considered to be a multi-line input string if the function that uses the input string includes the `m` flag.

question mark (?)
Matches the preceding character or character group in the regular expression zero or one time.

asterisk (*)
Matches the preceding character or character group in the regular expression zero or more times.

plus sign (+)
Matches the preceding character or character group in the regular expression one or more times.

{*n*}
Matches the preceding character or character group in the regular expression exactly *n* times. *n* must be a positive integer.

{*n*,*m*}
Matches the preceding character or character group in the regular expression at least *n* times, but not more than *m* times. *n* must be a positive integer, and *m* must be a positive integer that is greater than or equal to *n*.

{*n*,}
Matches the preceding character or character group in the regular expression at least *n* times. *n* must be a positive integer.

opening bracket ([) and closing bracket (])
The opening and closing brackets and the enclosed character group define a character class. For example, the character class [aeiou] matches any single vowel. Character classes also support character ranges. For example:

- [a-z] means any lowercase letter.
- [a-p] means any lowercase letter from a through p.
- [0-9] means any single digit.

opening parenthesis (() and closing parenthesis ())
An opening and closing parenthesis denote a grouping of some characters within a regular expression. You can then apply an operator, such as a repetition operator, to the entire group.

character-class-escape
A character class escape specifies that you want certain special characters to be treated as characters, instead of performing some function. A character class escape consists of a backslash (\), followed by a single metacharacter, newline character, return character, or tab character. The following table lists the character class escapes.

Table 50. Single-character character class escapes

Character escape	Character represented	Description
`\n`	#x0A	Newline
`\r`	#x0D	Return
`\t`	#x09	Tab
`\\`	\	Backslash
`\\|`	|	Pipe

Table 50. Single-character character class escapes (continued)

Character escape	Character represented	Description
\.	.	Period
\?	?	Question mark
*	*	Asterisk
\+	+	Plus sign
\(	(	Opening parenthesis
\)	)	Closing parenthesis
\{	{	Opening curly brace
\}	}	Closing curly brace
\$	$	Dollar sign
\-	-	Dash
\[	[	Opening bracket
\]	]	Closing bracket
\^	^	Caret

character-group
A character group is the set of characters in a character class. The character class is used for matching. It can consist characters, character ranges, character class escapes, and an optional opening carat. If the carat is included, it indicates the complement of the set of characters that are defined by the rest of character group.

Examples

The following examples demonstrate how each of the metacharacters affects a regular expression.

- "hello[0-9]world" matches "hello3world", but not "hello world".
- "^hello" matches this text:

```
hello world
```

  However, "^hello" does not match this text:

```
world hello
```

- "hello$" matches this text:

```
world hello
```

  However, "hello$" does not match this text:

```
hello world
```

- "(ca)|(bd)" matches "arcade" or "abdicate".
- "^((ca)|(bd))" does not match "arcade" or "abdicate".
- "w?s" matches "ws" or "s".
- "w.*s" matches "was" or "waters".
- "be+t" matches "beet" or "bet".
- "be{1,3}t" matches "bet", "beet", or "beeet".

- "\[n\]" matches "[n]".

Related reference

fn:matches function
The fn:matches function determines whether a string matches a given pattern.

fn:replace function
The fn:replace function compares each set of characters within a string to a given pattern. fn:replace replaces the characters that match the pattern with another set of characters.

fn:tokenize function
The fn:tokenize function breaks a string into a sequence of substrings.

Chapter 11. Descriptions of XQuery functions

The XQuery functions are a subset of the XPath 2.0 and XQuery 1.0 functions and operators.

These topics provide detailed reference information for the XQuery functions that are supported by Db2 for z/OS.

Restriction: An argument to an XQuery function cannot contain an FLWOR expression.

fn:abs function

The fn:abs function returns the absolute value of a numeric value.

Syntax



numeric-value
An atomic value or an empty sequence.

If *numeric-value* is an atomic value, it has one of the following types:

- xs:double
- xs:decimal
- xs:integer
- A type that is derived from any of the previously listed types
- xs:untypedAtomic

If *numeric-value* has the xs:untypedAtomic data type, it is converted to an xs:double value.

Returned value

If *numeric-value* is not the empty sequence, the returned value is the absolute value of *numeric-value*.

If *numeric-value* is the empty sequence, fn:abs returns the empty sequence.

The data type of the returned value depends on the data type of *numeric-value*:

- If *numeric-value* is xs:double, xs:decimal or xs:integer, the value that is returned has the same type as *numeric-value*.
- If *numeric-value* has a data type that is derived from xs:double, xs:decimal or xs:integer, the value that is returned has the direct parent data type of *numeric-value*.
- If *numeric-value* has the xs:untypedAtomic data type, the value that is returned has the xs:double data type.

Example

The following function returns the absolute value of –10.5.

```
fn:abs(-10.5)
```

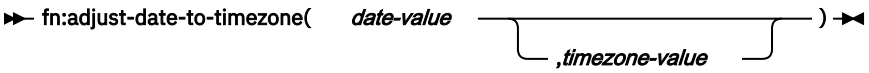
The returned value is 10.5.

© Copyright IBM Corp. 2007, 2020

fn:adjust-date-to-timezone function

The `fn:adjust-date-to-timezone` function adjusts an xs:date value to a specific time zone, or removes the time zone component from the value.

Syntax



date-value
The date value that is to be adjusted.

date-value is of type xs:date, or is an empty sequence.

timezone-value
A duration that represents the time zone to which *date-value* is to be adjusted.

timezone-value can be an empty sequence or a single value of type xs:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

Returned value

The returned value is either a value of type xs:date or an empty sequence depending on the parameters that are specified. If *date-value* is not an empty sequence, the returned value is of type xs:date. The following table describes the possible returned values:

Table 51. Types of input values and returned value for `fn:adjust-date-to-timezone`

date-value	***timezone-value***	**Returned value**
date-value that contains a time zone component	An explicit value, or no value specified (duration of PT0H)	The *date-value* adjusted for the time zone represented by *timezone-value*.
date-value that contains a time zone component	An empty sequence	The *date-value* with no time zone component.
date-value that does not contain a time zone component	An explicit value, or no value specified (duration of PT0H)	The *date-value* with a time zone component. The time zone component is the time zone represented by *timezone-value*. The date component is not adjusted for the time zone.
date-value that does not contain a time zone component	An empty sequence	The *date-value*.
An empty sequence	An explicit value, empty sequence, or no value specified	An empty sequence.

When adjusting *date-value* to a different time zone, *date-value* is treated as a dateTime value with time component 00:00:00. The returned value contains the time zone component represented by *timezone-value*. The following function calculates the adjusted date value:

```
xs:date(fn:adjust-dateTime-to-timezone(xs:dateTime(date-value),timezone-value))
```

Examples

In the following examples, the variable `$tz` is a duration of -10 hours, defined as `xs:dayTimeDuration("-PT10H")`.

The following function adjusts the date value for May 7, 2009 in the UTC+1 time zone. The function specifies a *timezone-value* of -PT10H.

```
fn:adjust-date-to-timezone(xs:date("2009-05-07+01:00"), $tz)
```

The returned date value is 2009-05-06-10:00. The date is adjusted to the UTC-10 time zone.

The following function adds a time zone component to the date value for March 7, 2009 without a time zone component. The function specifies a *timezone-value* of -PT10H.

```
fn:adjust-date-to-timezone(xs:date("2009-03-07"), $tz)
```

The returned value is 2009-03-07-10:00. The time zone component is added to the date value.

The following function adjusts the date value for February 9, 2009 in the UTC-7 time zone. Without a *timezone-value* specified, the function uses the default *timezone-value* PT0H.

```
fn:adjust-date-to-timezone(xs:date("2009-02-09-07:00"))
```

The returned date is 2009-02-09Z, the date is adjusted to UTC.

The following function removes the time zone component from the date value for May 7, 2009 in the UTC-7 time zone. The *timezone-value* is an empty sequence.

```
fn:adjust-date-to-timezone(xs:date("2009-05-07-07:00"), ())
```

The returned value is 2009-05-07.

fn:adjust-dateTime-to-timezone function

The `fn:adjust-dateTime-to-timezone` function adjusts an xs:dateTime value to a specific time zone, or removes the time zone component from the value.

Syntax



dateTime-value
The dateTime value that is to be adjusted.

dateTime-value is of type xs:dateTime, or is an empty sequence.

timezone-value
A duration that represents the time zone to which *dateTime-value* is to be adjusted.

timezone-value can be an empty sequence or a single value of type xs:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

Returned value

The returned value is either a value of type xs:dateTime or is an empty sequence depending on the types of input values. If *dateTime-value* is not an empty sequence, the returned value is of type xs:dateTime. The following table describes the possible returned values:

Table 52. Types of input values and returned value for `fn:adjust-dateTime-to-timezone`

dateTime-value	***timezone-value***	**Returned value**
dateTime-value that contains a time zone component	An explicit value, or no value specified (duration of PT0H)	The *dateTime-value* adjusted to the time zone represented by *timezone-value*. The returned value contains the time zone component represented by *timezone-value*.
dateTime-value that contains atime zone component	An empty sequence	The *dateTime-value* with no time zone component.
dateTime-value that does not contain a time zone component	An explicit value, or no value specified (duration of PT0H)	The *dateTime-value* with a time zone component. The time zone component is the time zone represented by *timezone-value*. The date and time components are not adjusted to the time zone.
dateTime-value that does not contain a time zone component	An empty sequence	The *dateTime-value*.
An empty sequence	An explicit value, empty sequence, or no value specified	An empty sequence.

Examples

In the following examples, the variable `$tz` is a duration of -10 hours, defined as `xs:dayTimeDuration("-PT10H")`.

The following function adjusts the dateTime value of March 7, 2009 at 10 a.m. in the UTC-7 time zone to the time zone specified by *time zone-value* of -PT10H.

```
fn:adjust-dateTime-to-timezone(xs:dateTime("2009-03-07T10:00:00-07:00"), $tz)
```

The returned dateTime value is 2009-03-07T07:00:00-10:00.

The following function adjusts the dateTime value for March 7, 2009 at 10 am. The *dateTime-value* does not have a time zone component, and the function specifies a *timezone-value* of -PT10H.

```
fn:adjust-dateTime-to-timezone(xs:dateTime("2009-03-07T10:00:00"), $tz)
```

The returned dateTime is 2009-03-07T10:00:00-10:00.

In the following function adjusts the dateTime value for June 4, 2009 at 10 a.m. in the UTC-7 time zone. Without a *timezone-value* specified, the function uses the default time zone value of PT0H.

```
fn:adjust-dateTime-to-timezone(xs:dateTime("2009-06-04T10:00:00-07:00"))
```

The returned dateTime value is 2009-06-04T17:00:00Z, which is the dateTime value adjusted to UTC.

The following function removes the time zone component from the dateTime value for March 7, 2009 at 10 a.m. in the UTC-7 time zone. The *timezone-value* value is the empty sequence.

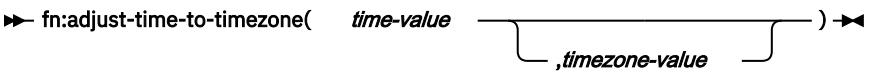
```
fn:adjust-dateTime-to-timezone(xs:dateTime("2009-03-07T10:00:00-07:00"), ())
```

The returned dateTime value is 2009-03-07T10:00:00.

fn:adjust-time-to-timezone function

The `fn:adjust-time-to-timezone` function adjusts an xs:time value to a specific time zone, or removes the time zone component from the value.

Syntax



time-value
The time value that is to be adjusted.

time-value is of type xs:time, or is an empty sequence.

timezone-value
A duration that represents the time zone to which *time-value* is to be adjusted.

timezone-value can be an empty sequence or a single value of type xs:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

Returned value

The returned value is either a value of type xs:time or an empty sequence depending on the parameters that are specified. If *time-value* is not an empty sequence, the returned value is of type xs:time. The following table describes the possible returned values:

Table 53. Types of input values and returned value for `fn:adjust-time-to-timezone`

date-value	***timezone-value***	**Returned value**
time-value that contains a time zone component	An explicit value, or no value specified (duration of PT0H)	The *time-value* adjusted for the time zone represented by *timezone-value*. The returned value contains the time zone component represented by *timezone-value*. If the time zone adjustment crosses over midnight, the change in date is ignored.
time-value that contains a time zone component	An empty sequence	The *time-value* with no time zone component.
time-value that does not contain a time zone component	An explicit value, or no value specified (duration of PT0H)	The *time-value* with a time zone component. The time zone component is the time zone represented by *timezone-value*. The time component is not adjusted for the time zone.
time-value that does not contain a time zone component	An empty sequence	The *time-value*.
An empty sequence	An explicit value, empty sequence, or no value specified	An empty sequence.

Examples

In the following examples, the variable `$tz` is a duration of -10 hours, defined as `xs:dayTimeDuration("-PT10H")`.

The following function adjusts the time value for 10:00 a.m. in the UTC-7 time zone, and the function specifies a *timezone-value* of -PT10H.

```
fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"), $tz)
```

The returned value is 7:00:00-10:00. The time is adjusted to the time zone represented by the duration -PT10H.

The following function adjusts the time value for 1:00 p.m. The time value does not have a time zone component.

```
fn:adjust-time-to-timezone(xs:time("13:00:00"), $tz)
```

The returned value is 13:00:00-10:00. The time contains a time zone component represented by the duration -PT10H.

The following function adjusts the time value for 10:00 a.m. in the UTC-7 time zone. The function does not specify a *timezone-value* and uses the default value of PT0H.

```
fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"))
```

The returned value is 17:00:00Z, the time adjusted to UTC.

The following function removes the time zone component from the time value 8:00 am in the UTC-7 time zone. The *timezone-value* is the empty sequence.

```
fn:adjust-time-to-timezone(xs:time("08:00:00-07:00"), ())
```

The returned value is 8:00:00.

The following example compares two times. The time zone adjustment crosses over midnight and causes a date change. However, `fn:adjust-time-to-timezone` ignores date changes.

```
fn:adjust-time-to-timezone(xs:time("01:00:00+14:00"), $tz)
   = xs:time("01:00:00-10:00")
```

The returned value is true.

fn:avg function

The fn:avg function returns the average of the values in a sequence.

Syntax



sequence-expression

A sequence that contains items of any of the following atomic types, or an empty sequence:

- xs:double
- xs:decimal
- xs:integer
- xs:untypedAtomic
- xs:dayTimeDuration
- xs:yearMonthDuration
- A type that is derived from any of the previously listed types

Input items of type xs:untypedAtomic are cast to xs:double.

Returned value

If *sequence-expression* is not the empty sequence, the returned value is the average of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* is the empty sequence, the empty sequence is returned.

If *sequence-expression* has two or more of the following data types, an error is returned:

- xs:dayTimeDuration
- xs:yearMonthDuration
- numeric data types

Example

The following function returns the average of the sequence (5, 1.0E2, 40.5):

```
fn:avg((5, 1.0E2, 40.5))
```

The values are promoted to the xs:double data type. The function returns the xs:double value 4.85E1, which is serialized as "48.5".

fn:boolean function

The fn:boolean function returns the effective boolean value of a sequence.

Syntax



sequence-expression
Any sequence that contains items of any type, or the empty sequence.

Returned value

The returned value is an xs:boolean value that depends on the value of *sequence-expression*:

- If *sequence-expression* is the empty sequence, false is returned.
- If *sequence-expression* is not the empty sequence, and *sequence-expression* contains one value:
  - If the value is the xs:boolean value false, false is returned.
  - If the value is a string of length 0, and the type is xs:string or xs:untypedAtomic, false is returned.
  - If the value is 0, and the type is a numeric type, false is returned.
  - If the value is NaN, and the type is xs:double, false is returned.
  - Otherwise, true is returned.
- If *sequence-expression* is not the empty sequence, and *sequence-expression* contains more than one value, true is returned.

Examples

The following functions return true, because they are both strings with a length greater than zero

- `fn:boolean("true")`
- `fn:boolean("false")`

The following function returns false, because the value of the *sequence-expression* is 0 and the type is numeric: `fn:boolean(0)`

The following examples show how the fn:boolean function can be used in the larger context of Db2. The XMLQUERY statements are used to query XML data and take an XQuery expression as the first argument. In both cases, the fn:boolean function is passed data from the PASSING clause. In the first example, the argument of the fn:boolean function is a sequence of three zeros. In the second example, the argument of the fn:boolean function is three "false" values. Both functions evaluate to true, because the *sequence-expression* contains more than one value.

```
XMLQUERY('fn:boolean($x)' PASSING XMLCONCAT(XMLQUERY('0'),
 XMLQUERY('0'),XMLQUERY('0')))
```

```
XMLQUERY('fn:boolean(fn:data(//b))'
 PASSING XMLPARSE(DOCUMENT '<a><b>false</b><b>false></b><b>false</b></a>'))
```

fn:compare function

The fn:compare function compares two strings.

Syntax



string-1* and *string-2

The xs:string values that are to be compared. Db2 compares the numeric Unicode UTF-8 code value of each character.

Returned value

If *string-1* and *string-2* are not the empty sequence, one of the following xs:integer values is returned:

-1

If *string-1* is less than *string-2*.

0

If *string-1* is equal to *string-2*.

1

If *string-1* is greater than *string-2*.

Two strings are compared by comparing the corresponding bytes of each string. If the strings do not have the same length, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks so that it has the same length as the other string.

string-1 and *string-2* are equal if they both have length 0 or if all corresponding bytes are equal.

If *string-1* and *string-2* are not equal, their relationship (that is, which has the greater value) is determined by the comparison of the first pair of unequal bytes from the left end of the strings. This comparison is made according to the collation.

If *string-1* is longer than *string-2*, and all bytes of *string-2* are equal to the leading bytes of *string-1*, *string-1* is greater than *string-2*.

If *string-1* or *string-2* is the empty sequence, the empty sequence is returned.

Example

The following function compares 'ABC' to 'ABD' using the default collation.

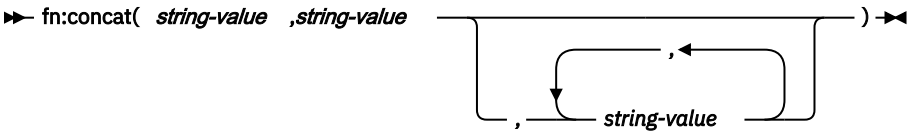
```
fn:compare('ABC', 'ABD')
```

'ABC' is less than 'ABD'. The returned value is -1.

fn:concat function

The fn:concat function concatenates two or more strings into a single string.

Syntax



string-value
An xs:string value or the empty sequence.

Returned value

If all *string-value* arguments are the empty sequence, the returned value is the empty sequence. Otherwise, the returned value is an xs:string value that is the concatenation of all *string-value* arguments that are not the empty sequence.

Example

The following function concatenates the strings 'ABC', 'ABD', the empty sequence, and 'ABE',

```
fn:concat('ABC', 'ABD', (), 'ABE')
```

The returned value is 'ABCABDABE'.

fn:contains function

The fn:contains function determines whether a string contains a given substring.

Syntax



string
The string to search for *substring*.

string has the xs:string data type, or is the empty sequence. If *string* is the empty sequence, *string* is set to a string of length 0.

substring
The substring to search for in *string*.

substring has the xs:string data type, or is the empty sequence.

Returned value

The returned value depends on the values of *string* and *substring*:

- If *string* and *substring* are not the empty sequence, the returned value is true if *substring* occurs anywhere within *string*. If *substring* does not occur within *string*, the returned value is false.
- If *string* is the empty sequence, the returned value is true if *substring* is the empty sequence or a string of length 0.
- If *substring* is the empty sequence or a string of length 0, the returned value is true.

Example

The following function determines whether the string 'Test literal' contains the string 'lite'.

```
fn:contains('Test literal','lite')
```

The returned value is true.

fn:count function

The fn:count function returns the number of values in a sequence.

Syntax



sequence-expression
A sequence that contains items of any atomic type, or an empty sequence.

Returned value

If *sequence-expression* is not the empty sequence, an xs:integer value that is the number of values in *sequence-expression* is returned. If *sequence-expression* is the empty sequence, 0 is returned.

Example

The following function returns 1:

```
fn:count(5)
```

The following function returns the number of employees with a department ID of K55:

```
fn:count(//company/emp[dept/@id="K55"])
```

fn:current-date function

The `fn:current-date` function returns the current date in the local time zone.

Syntax



Returned value

The returned value is an xs:date value that is the current date. The time zone component of the returned value is the local time zone.

Example

The following function returns the current date.

```
fn:current-date()
```

If this function were invoked on December 2, 2009, in Pacific Standard Time, the returned value would be 2009-12-02-08:00.

fn:current-dateTime function

The `fn:current-dateTime` function returns the current date and time in the local time zone.

Syntax



Returned value

The returned value is an xs:dateTime value that is the current date and time. The time zone component of the returned value is the local time zone. The maximum precision for fractions of seconds is 12.

Example

The following function returns the current date and time.

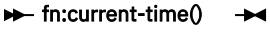
```
fn:current-dateTime()
```

If this function were invoked on December 2, 2009 at 6:25 in Toronto (time zone -PT5H), the returned value would be 2009-12-02T06:25:30.3847249023-05:00.

fn:current-time function

The `fn:current-time` function returns the current time in the local time zone.

Syntax



Returned value

The returned value is an xs:time value that is the current time. The time zone component of the returned value is the local time zone. The precision for fractions of seconds is 12.

Example

The following function returns the current time.

```
fn:current-time()
```

If this function were invoked at 6:31 Pacific Standard Time (-08:00), the returned value would be 06:31:35.519003948231-08:00.

fn:data function

The fn:data function converts a sequence of items to a sequence of atomic values.

Syntax



sequence
Any sequence, including the empty sequence.

Returned values

The returned value is a sequence of items of type xs:anyAtomicType. For each item in the sequence:

- If the item is an atomic value, the returned value is that value.
- If the item is a node, the returned value is the typed value of the node.

Example

The following function returns the typed values of all qualifying name nodes. Qualifying name nodes are all name nodes that are children of a billTo node in the document.

```
fn:data(//billTo/name)
```

fn:dateTime function

The `fn:dateTime` function constructs an xs:dateTime value from an xs:date value and an xs:time value.

Syntax



date-value
An xs:date value.

time-value
An xs:time value.

Returned value

The returned value is an xs:dateTime value with a date component that is equal to *date-value* and a time component that is equal to *time-value*. The time zone of the result is computed as follows:

- If neither argument has a time zone, the result has no time zone.
- If exactly one of the arguments has a time zone, or if both arguments have the same time zone, the result has this time zone.
- If the two arguments have different time zones, an error is returned.

Example

The following function returns an xs:dateTime value from an xs:date value and an xs:time value.

```
fn:dateTime((xs:date("2009-04-16")), (xs:time("12:30:59")))
```

The returned value is the xs:dateTime value 2009-04-16T12:30:59.

fn:day-from-date function

The `fn:day-from-date` function returns the day component of an xs:date value that is in its localized form.

Syntax



date-value
The date value from which the day component is to be extracted.

date-value is of type xs:date, or is an empty sequence.

Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer, and the value is between 1 and 31, inclusive. The value is the day component of *date-value*.

If *date-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the day component of the date value for June 1, 2009.

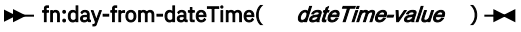
```
fn:day-from-date(xs:date("2009-06-01"))
```

The returned value is 1.

fn:day-from-dateTime function

The `fn:day-from-dateTime` function returns the day component of an xs:dateTime value that is in its localized form.

Syntax



dateTime-value

The dateTime value from which the day component is to be extracted.

dateTime-value is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 1 and 31, inclusive. The value is the day component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the day component of the dateTime value for January 31, 2009 at 8:00 p.m. in the UTC+4 time zone.

```
fn:day-from-dateTime(xs:dateTime("2009-01-31T20:00:00+04:00"))
```

The returned value is 31.

fn:days-from-duration function

The `fn:days-from-duration` function returns the days component of a duration.

Syntax



duration-value

The duration value from which the days component is to be extracted.

duration-value is an empty sequence, or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:dayTimeDuration or is of type xs:duration, the returned value is of type xs:integer, and is the days component of *duration-value* cast as xs:dayTimeDuration. The returned value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:yearMonthDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The days component of *duration-value* cast as xs:dayTimeDuration is the integer number of days computed as `(S idiv 86400)`. The value *S* is the total number of seconds of *duration-value* cast as xs:dayTimeDuration to remove the years and months components.

Examples

This function returns the days component of the duration -10 days and 0 hours.

```
fn:days-from-duration(xs:dayTimeDuration("-P10DT00H"))
```

The returned value is -10.

This function returns the days component of the duration 3 days and 55 hours.

```
fn:days-from-duration(xs:dayTimeDuration("P3DT55H"))
```

The returned value is 5. When calculating the total number of days in the duration, 55 hours is converted to 2 days and 7 hours. The duration is equal to P5D7H which has a days component of 5 days.

fn:distinct-values function

The fn:distinct-values function returns the distinct values in a sequence.

Syntax



sequence-expression
A sequence of atomic values, or the empty sequence. The items in the sequence can have any of the following types:

- Numeric
- String
- Date or time types

Returned value

If *sequence-expression* is not the empty sequence, the returned value is a sequence that contains xs:string values that are the distinct values in *sequence-expression*. Two items are distinct if they are not equal to each other. XQuery uses the following rules to obtain a sequence of distinct values:

- If two values cannot be compared, those values are considered to be distinct.
- Values of type xs:untypedAtomic are compared using the rules for xs:string types.
- The order in which the sequence of values is returned might not be the same as the input order.
- The first value of a set of values that compare equal is returned.
- If *sequence-expression* is the empty sequence, the empty sequence is returned.
- For xs:double values, positive zero is equal to negative zero.
- If *sequence-expression* contains multiple NaN values, a single NaN value is returned.

Example

The following example returns the distinct values of node b:

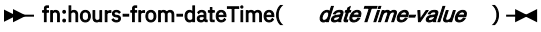
```
SELECT XMLSERIALIZE(
 XMLQUERY ('declare default element namespace
 "http://posample.org";
 fn:distinct-values($d/x/b)'  PASSING XMLPARSE(DOCUMENT
 '<x xmlns="http://posample.org">
 <b>1</b><b>a</b><b>1.0</b><b>A</b><b>1</b></x>')
 AS "d")
 AS CLOB(1K) EXCLUDING XMLDECLARATION)
 FROM SYSIBM.SYSDUMMY1
```

The returned value is ("1", "a", "1.0", "A").

fn:hours-from-dateTime function

The `fn:hours-from-dateTime` function returns the hours component of an xs:dateTime value that is in its localized form.

Syntax



dateTime-value

The dateTime value from which the hours component is to be extracted.

dateTime-value is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 0 and 23, inclusive. The value is the hours component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the hours component of the dateTime value for January 31, 2009 at 2:00 p.m. in the UTC-8 time zone.

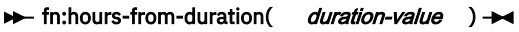
```
fn:hours-from-dateTime(xs:dateTime("2009-01-31T14:00:00-08:00"))
```

The returned value is 14.

fn:hours-from-duration function

The `fn:hours-from-duration` function returns the hours component of a duration value.

Syntax



duration-value

The duration value from which the hours component is to be extracted.

duration-value is an empty sequence or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:dayTimeDuration or is of type xs:duration, the returned value is of type xs:integer, and is a value between -23 and 23, inclusive. The value is the hours component of *duration-value* cast as xs:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:yearMonthDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The hours component of *duration-value* cast as xs:dayTimeDuration is the integer number of hours computed as `((`*S*` mod 86400) idiv 3600)`. The value *S* is the total number of seconds of *duration-value* cast as xs:dayTimeDuration to remove the days and months component. The value 86400 is the number of seconds in a day, and 3600 is the number of seconds in an hour.

Example

The following function returns the hours component of the duration 126 hours.

```
fn:hours-from-duration(xs:dayTimeDuration("PT126H"))
```

The returned value is 6. When calculating the total number of hours in the duration, 126 hours is converted to 5 days and 6 hours. The duration is equal to P5DT6H which has an hours component of 6 hours.

fn:hours-from-time function

The `fn:hours-from-time` function returns the hours component of an xs:time value that is in its localized form.

Syntax



time-value
The time value from which the hours component is to be extracted.

time-value is of type xs:time, or is an empty sequence.

Returned value

If *time-value* is not an empty sequence, the returned value is of type xs:integer, and the value is between 0 and 23, inclusive. The value is the hours component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the hours component of the time value for 9:30 a.m. in the UTC-8 time zone.

```
fn:hours-from-time(xs:time("09:30:00-08:00"))
```

The returned value is 9.

fn:implicit-timezone function

The `fn:implicit-timezone` function returns the time zone that is used when a date, time, or dateTime value that does not have a time zone is used in a comparison or arithmetic operation.

The implicit time zone is the value of PT0S.

Syntax



Returned value

The returned implicit time zone value has type xs:dayTimeDuration.

Example

The following function returns xs:dayTimeDuration("PT0S"):

```
fn:implicit-timezone()
```

fn:minutes-from-dateTime function

The `fn:minutes-from-dateTime` function returns the minutes component of an xs:dateTime value that is in its localized form.

Syntax



dateTime-value
The dateTime value from which the minutes component is to be extracted.

dateTime-value is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 0 and 59, inclusive. The value is the minutes component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the minutes component from the dateTime value for January 23, 2009 at 9:42 a.m. in the UTC-8 time zone.

```
fn:minutes-from-dateTime(xs:dateTime("2009-01-23T09:42:00-08:00"))
```

The returned value is 42.

fn:minutes-from-duration function

The `fn:minutes-from-duration` function returns the minutes component of a duration.

Syntax



duration-value
The duration value from which the minutes component is to be extracted.

duration-value is an empty sequence, or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:dayTimeDuration or is of type xs:duration, the returned value is of type xs:integer and is a value between -59 and 59, inclusive. The value is the minutes component of *duration-value* cast as xs:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:yearMonthDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The minutes component of *duration-value* cast as xs:dayTimeDuration is the integer number of minutes computed as `((S mod 3600) idiv 60)`. The value *S* is the total number of seconds of *duration-value* cast as xs:dayTimeDuration to remove the years and months components.

Example

The following function returns the minutes component of the duration 2 days, 16 hours, and 93 minutes.

```
fn:minutes-from-duration(xs:dayTimeDuration("P2DT16H93M"))
```

The returned value is 33. When calculating the total number of minutes in the duration, 93 minutes is converted to 1 hour and 33 minutes. The duration is equal to P2DT17H33M which has a minutes component of 33 minutes.

fn:minutes-from-time function

The `fn:minutes-from-time` function returns the minutes component of an xs:time value that is in its localized form.

Syntax

time-value
The time value from which the minutes component is to be extracted.

time-value is of type xs:time, or is an empty sequence.

Returned value

If *time-value* is of type xs:time, the returned value is of type xs:integer, and the value is between 0 and 59, inclusive. The value is the minutes component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the minutes component of the time value for 8:59 a.m. in the UTC-8 time zone.

```
fn:minutes-from-time(xs:time("08:59:00-08:00"))
```

The returned value is 59.

fn:month-from-date function

The `fn:month-from-date` function returns the month component of a xs:date value that is in its localized form.

Syntax



date-value
The date value from which the month component is to be extracted.

date-value is of type xs:date, or is an empty sequence.

Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer, and the value is between 1 and 12, inclusive. The value is the month component of *date-value*.

If *date-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the month component of the date value for December 1, 2009.

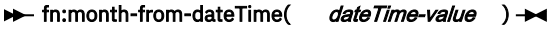
```
fn:month-from-date(xs:date("2009-12-01"))
```

The returned value is 12.

fn:month-from-dateTime function

The `fn:month-from-dateTime` function returns the month component of an xs:dateTime value that is in its localized form.

Syntax



dateTime-value
The dateTime value from which the month component is to be extracted.

dateTime-value is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 1 and 12, inclusive. The value is the month component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the month component of the dateTime value for October 31, 2009 at 8:15 a.m. in the UTC-8 time zone.

```
fn:month-from-dateTime(xs:dateTime("2009-10-31T08:15:00-08:00"))
```

The returned value is 10.

fn:months-from-duration function

The `fn:months-from-duration` function returns the months component of a duration value.

Syntax



duration-value
The duration value from which the months component is to be extracted.

duration-value is an empty sequence, or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:duration or is of type xs:yearMonthDuration, the returned value is of type xs:integer, and is a value is between -11 and 11, inclusive. The value is the months component of *duration-value* cast as xs:yearMonthDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:dayTimeDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The months component of *duration-value* cast as xs:yearMonthDuration is the integer number of months remaining from the total number of months of *duration-value* divided by 12.

Examples

The following function returns the months component of the duration 20 years and 5 months.

```
fn:months-from-duration(xs:duration("P20Y5M"))
```

The returned value is 5.

The following function returns the months component of the yearMonthDuration -9 years and -13 months.

```
fn:months-from-duration(xs:yearMonthDuration("-P9Y13M"))
```

The returned value is -1. When calculating the total number of months in the duration, -13 months is converted to -1 year and -1 month. The duration is equal to -P10Y1M which has a month component of -1 month.

The following function returns the months component of the duration 14 years, 11 months, 40 days, and 13 hours.

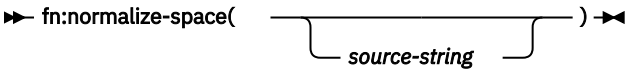
```
xquery fn:months-from-duration(xs:duration("P14Y11M40DT13H"))
```

The returned value is 11.

fn:normalize-space function

The fn:normalize-space function strips leading and trailing whitespace characters from a string and replaces multiple consecutive whitespace characters in the string with a single blank character.

Syntax



source-string

A string in which whitespace is to be normalized.

source-string is an xs:string value or the empty sequence.

If *source-string* is not specified, the argument of fn:normalize-space is the current context item, which is converted to an xs:string value by using the fn:string function.

Returned value

The returned value is the xs:string value that results when the following operations are performed on *source-string*:

- Leading and trailing whitespace characters are removed.
- Each internal sequence of one or more adjacent whitespace characters is replaced by a single space (U +0020) character.

Whitespace characters are the space character, (U+0020), carriage return, (U+000D), line feed, (U +000A), and tab (U+0009).

If *source-string* is the empty sequence, a string of length 0 is returned.

Example

The following function removes extra whitespace characters from the string "a b c d ".

```
fn:normalize-space(" a b c d " )
```

The returned value is "a b c d".

fn:last function

The fn:last function returns the number of values in the sequence of items that is currently being processed.

Syntax



Returned value

If the sequence that is currently being processed is not the empty sequence, the returned value is an xs:integer value that is the number of values in the sequence. If the sequence that is currently being processed is the empty sequence, the returned value is the empty sequence.

In the following cases, an error is returned:

- fn:last is separated from its context item by "/" or "//".

  For example, the following expressions are not supported:

  ```
  /a/b/c/fn:last
  /a/b/[c/fn:last=3]
  ```

- The context node has a descendant axis or descendant-or-self axis.

  For example, the following expression is not supported:

  ```
  /a/b/descendant::c[fn:last()=1]
  ```

- The context node is a filter expression, and the filter expression has a step with a descendant axis or descendant-or-self axis, or a nested filter expression.

  For example, the following expression is not supported:

  ```
  /a/(b/descendant::c)[fn:last()=1]
  ```

Example

In the sample CUSTOMER table, the customer document for customer 1003 looks like this:

```
<customerinfo xmlns="http://posample.org" Cid="1003">
  <name>Robert Shoemaker</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X-7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
  <phone type="home">416-555-2937</phone>
  <phone type="cell">905-555-8743</phone>
  <phone type="cottage">613-555-3278</phone>
</customerinfo>
```

The following query retrieves the last phone number in the document. The query calls the fn:last function to determine the number of phone number items, and then uses the fn:last result to point to the last phone number.

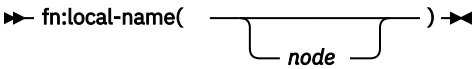
```
SELECT
 XMLQUERY('declare default element namespace "http://posample.org";
 $X/customerinfo/phone[fn:last()]'
 PASSING INFO AS "X") FROM CUSTOMER WHERE CID=1003
```

The returned value is <phone type="cottage">613-555-3278</phone>.

fn:local-name function

The fn:local-name function returns the local name property of a node.

Syntax



node
The node for which the local name is to be retrieved. If *node* is not specified, fn:local-name is evaluated for the current context node.

Returned value

The returned value is an xs:string value. The value depends on whether *node* is specified, and the value of *node*:

- If *node* is not specified, the local name of the context node is returned.
- If *node* meets any of the following conditions, a string of length 0 is returned:
  - *node* is the empty sequence.
  - *node* is a document node, a comment, or a text node. These nodes have no name.
- In the following cases, an error is returned:
  - The context node is undefined.
  - The context item is not a node.
  - *node* is a sequence of more than one node.
- Otherwise, an xs:string value is returned that contains the local name part of the expanded name for *node*.

Examples

The following example returns the local name for node b.

```
SELECT XMLQUERY (
 'declare default element namespace "http://posample.org";
 fn:local-name($d/x/b)'
 PASSING XMLPARSE(DOCUMENT
 '<x xmlns="http://posample.org"><b><c></c></b></x>')
 AS "d")
 FROM SYSIBM.SYSDUMMY1
```

The returned value is "b".

The following example demonstrates that fn:localname() with no argument returns the context node.

In the sample CUSTOMER table, the customer document for customer 1001 looks like this:

```
<customerinfo xmlns="http://posample.org" Cid="1001">
  <name>Kathy Smith</name>
  <addr country="Canada">
```

```
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
</customerinfo>
```

The following example returns the local name for the context node.

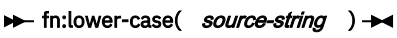
```
SELECT
XMLSERIALIZE(
XMLQUERY('declare default element namespace "http://posample.org";
 $X/customerinfo/phone/fn:local-name()'
 PASSING INFO AS "X")
 AS CLOB(1K))
 FROM CUSTOMER WHERE CID=1001
```

The returned value is "phone".

fn:lower-case function

The fn:lower-case function converts a string to lowercase.

Syntax



source-string
The string that is to be converted to lowercase.

source-string is of type xs:string, or is the empty sequence.

Returned value

If *source-string* is not the empty sequence, the returned value is an xs:string value that is *source-string*, with each character converted to its lowercase correspondent. Every character that does not have a lowercase correspondent is included in the returned value in its original form.

If *source-string* is the empty sequence, the returned value is a string of length zero.

Example

The following function converts the string "Wireless Router TB2561" to lowercase:

```
fn:lower-case("Wireless Router TB2561")
```

The returned value is "wireless router tb2561".

fn:matches function

The fn:matches function determines whether a string matches a given pattern.

Syntax



source-string
A string that is compared to a pattern.

source-string is a literal string, or an XQuery expression that resolves to an xs:string value or the empty sequence.

pattern
A regular expression that is compared to *source-string*. A regular expression is a set of characters, pattern-matching characters, and operators that define a string or group of strings in a search pattern.

pattern is string literal.

flags
A string literal that can contain any of the following values that control matching of *pattern* to *source-string*:

s
Indicates that the dot (.) matches any character.

If the s flag is not specified, the dot (.) matches any character except the new line character (#x0A).

m
Indicates that the caret (^) matches the start of any line (the position after a new line character), and the dollar sign ($) matches the end of any line (the position before a new line character).

If the m flag is not specified, the caret (^) matches the start of the entire string, and the dollar sign ($) matches the end of the entire string.

i
Indicates that matching is case-insensitive for the letters "a" through "z" and "A" through "Z".

If the i flag is not specified, case-sensitive matching is done.

x
Indicates that whitespace characters (#x09, #x0A, #x0D, and #x20) within *pattern* are ignored, unless they are within a character class. Whitespace characters in a character class are never ignored.

If the x flag is not specified, whitespace characters are used for matching.

Returned value

If *source-string* is not the empty sequence, the returned value is an xs:boolean value that is true if *source-string* matches *pattern*. The returned value is false if *source-string* does not match *pattern*.

The rules for matching are:

- If *pattern* does not contain the string-starting or line-starting character caret (^), or the string-ending or line-ending character dollar sign ($), *source-string* matches *pattern* if any substring of *source-string* matches *pattern*.
- If *pattern* contains the string-starting or line-starting character caret (^), *source-string* matches *pattern* only if *source-string* matches *pattern* from the beginning of *source-string* or the beginning of a line in *source-string*.
- If *pattern* contains the string-ending or line-ending character dollar sign ($), *source-string* matches *pattern* only if *source-string* matches *pattern* at the end of *source-string* or at the end of a line of *source-string*.
- The m flag determines:
  - Whether a match occurs from the beginning of the string or the beginning of a line
  - Whether a match occurs from the end of the string or the end of a line.

If *source-string* is the empty sequence, *source-string* is considered to be a string of length 0, and *source-string* matches *pattern* if *pattern* matches a string of length 0.

Examples

Example of matching a pattern to any substring within a string: The following function determines whether the strings "ac" or "bd" appear anywhere within the string "abbcacadbdcd".

```
fn:matches("abbcacadbdcd","(ac)|(bd)")
```

The returned value is true.

Example of matching a pattern to an entire string: The following function determines whether the strings "ac" or "bd" match the string "bd". The caret (^) character and the dollar sign ($) character indicate that the match must start at the beginning of the source string and end at the end of the source string.

```
fn:matches("bd","^(ac)|(bd)$")
```

The returned value is true.

Related reference

Regular expressions
A regular expression is a sequence of characters that act as a pattern for matching and manipulating strings. Regular expressions are used in the fn:matches, fn:replace, and fn:tokenize functions.

fn:max function

The fn:max function returns the maximum of the values in a sequence.

Syntax



sequence-expression
The empty sequence, or a sequence in which all of the items are one of the following types:

- Numeric
- String
- xs:date
- xs:dateTime
- xs:time
- xs:dayTimeDuration
- xs:yearMonthDuration

Input items of type xs:untypedAtomic are cast to xs:double. Numeric input items are converted to the least common type that can be compared by a combination of type promotion and subtype substitution.

Returned value

If *sequence-expression* is not the empty sequence, the returned value is a value of type xdt:anyAtomicType that is the maximum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the common data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* contains one item, that item is returned. If *sequence-expression* is the empty sequence, the empty sequence is returned. If the sequence includes the value NaN, NaN is returned.

Example

The following query returns the maximum of the sequence (500, 1.0E2, 40.5).

```
SELECT XMLSERIALIZE(
 XMLQUERY ('declare default element namespace "http://
posample.org";
 fn:max($d/x/b)'  PASSING XMLPARSE(DOCUMENT
 '<x xmlns="http://posample.org">
 <b>500</b><b>1.0E2</b><b>40.5</b></x>')
 AS "d")
 AS CLOB(1K) EXCLUDING XMLDECLARATION)
 FROM SYSIBM.SYSDUMMY1
```

The values are promoted to the xs:double data type. The function returns the xs:double value 5.0E2, which is serialized as 500.

fn:min function

The fn:min function returns the minimum of the values in a sequence.

Syntax



sequence-expression

The empty sequence, or a sequence in which all of the items are one of the following types:

- Numeric
- String
- xs:date
- xs:dateTime
- xs:time
- xs:dayTimeDuration
- xs:yearMonthDuration

Input items of type xs:untypedAtomic are cast to xs:double. Numeric input items are converted to the least common type that can be compared by a combination of type promotion and subtype substitution.

Returned value

If *sequence-expression* is not the empty sequence, the returned value is a value of type xs:anyAtomicType that is the minimum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the common data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* contains one item, that item is returned. If *sequence-expression* is the empty sequence, the empty sequence is returned. If the sequence includes the value NaN, NaN is returned.

Example

The following query returns the minimum of the sequence (500, 1.0E2, 40.5).

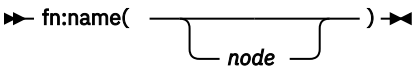
```
SELECT XMLSERIALIZE(
 XMLQUERY ('declare default element namespace "http://
posample.org";
 fn:min($d/x/b)'  PASSING XMLPARSE(DOCUMENT
 '<x xmlns="http://posample.org" xmlns="http://posample.org">
 <b>500</b><b>1.0E2</b><b>40.5</b></x>')
 AS "d")
 AS CLOB(1K) EXCLUDING XMLDECLARATION)
 FROM SYSIBM.SYSDUMMY1
```

The values are promoted to the xs:double data type. The function returns the xs:double value 4.05E1, which is serialized as 40.5.

fn:name function

The fn:name function returns the prefix and local name parts of a node name.

Syntax



node
The qualified name of a node for which the name is to be retrieved. If *node* is not specified, fn:name is evaluated for the current context node.

Returned value

The returned value is an xs:string value. The value depends on the value of *node*:

- If *node* meets any of the following conditions, a string of length 0 is returned:
  - *node* is the empty sequence.
  - *node* is a document node, a comment, or a text node. These nodes have no name.
- In the following cases, an error is returned:
  - The context node is undefined.
  - The context item is not a node.
  - *node* is a sequence of more than one node.
- Otherwise, an xs:string value is returned that contains the prefix (if present) and local name for *node*.

Example

The following example returns the qualified name for node b.

```
SELECT XMLSERIALIZE(
 XMLQUERY (  'declare namespace ns1="http://posample.org";
 fn:name($d/x/ns1:b)'
 PASSING XMLPARSE(DOCUMENT
 '<x xmlns:n="http://posample.org">
  <n:b><n:c></n:c></n:b></x>')
 AS "d")
 AS CLOB(1K) EXCLUDING XMLDECLARATION)
 FROM SYSIBM.SYSDUMMY1
```

The returned value is "n:b".

The following example demonstrates that fn:name() with no argument returns the context node.

In the sample CUSTOMER table, the customer document for customer 1001 looks like this:

```
<customerinfo xmlns="http://posample.org" Cid="1001">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
</customerinfo>
```

The following example returns the qualified name for the context node.

```
SELECT
XMLSERIALIZE(
XMLQUERY('declare default element namespace "http://posample.org";
 $X/customerinfo/phone/fn:name()'
 PASSING INFO AS "X")
 AS CLOB(1K))
 FROM CUSTOMER WHERE CID=1001
```

The returned value is "phone".

fn:not function

The fn:not function returns false if the effective boolean value of an item is true. fn:not returns true if the effective boolean value of an item is false.

Syntax



sequence-expression
Any sequence that contains items of any type, or the empty sequence.

Returned value

The returned value is an xs:boolean value. If the effective boolean value of *sequence-expression* is false, this function returns true. If the effective boolean value of *sequence-expression* is true, this function returns false.

Example

The following function returns true:

```
fn:not("a"="b")
```

The following function returns false:

```
fn:not("false")
```

fn:position function

The fn:position function returns the position of the context item in the sequence that is currently being processed.

The position function is typically used in a predicate. However it can also be used to produce the position of each occurrence of its context item.

Syntax



Returned value

The returned value is an xs:integer value that indicates the position of the context item in the sequence that is currently being processed. The first item in the sequence has position 1. If the context item is undefined, an error is returned. The position function returns a deterministic result only if the sequence that contains the context item has a deterministic order.

Examples

The following query returns the second element in the sequence of <c> elements in the document <x xmlns="http://posample.org"><c>x</c><c>y</c><c>z</c></x>.

```
SELECT XMLSERIALIZE(
 XMLQUERY ('declare default element namespace "http://posample.org";
 $d/x/b/c[fn:position()=2]'
 PASSING XMLPARSE(DOCUMENT
 '<x xmlns="http://posample.org">
 <b><c>x</c><c>y</c><c>z</c></b></x>')
 AS "d")
 AS CLOB(1K) EXCLUDING XMLDECLARATION)
 FROM SYSIBM.SYSDUMMY1
```

The returned value is "y".

The following query returns the position of each occurrence of <a><c>.

```
SELECT XMLSERIALIZE(
XMLQUERY('/a/b/c/fn:position()'
   PASSING XMLPARSE(DOCUMENT
   '<a><b><c>c1</c></b><b><c>c2</c><c>c3</c></b></a>'))
   AS CLOB(1K))
   FROM SYSIBM.SYSDUMMY1
```

The returned values is "1 2 3".

fn:replace function

The fn:replace function compares each set of characters within a string to a given pattern. fn:replace replaces the characters that match the pattern with another set of characters.

Syntax



source-string
A string that contains characters that are to be replaced.

source-string is a literal string, or an XQuery expression that resolves to an xs:string value or the empty sequence.

pattern
A regular expression that is compared to *source-string*. A regular expression is a set of characters, pattern-matching characters, and operators that define a string or group of strings in a search pattern.

pattern is string literal.

replacement-string
A string that contains characters that replace characters that match *pattern* in *source-string*.

replacement-string is an xs:string value.

flags
A string literal that can contain any of the following values that control matching of *pattern* to *source-string*:

s
Indicates that the dot (.) matches any character.

If the s flag is not specified, the dot (.) matches any character except the new line character (#x0A).

m
Indicates that the caret (^) matches the start of any line (the position after a new line character), and the dollar sign ($) matches the end of any line (the position before a new line character).

If the m flag is not specified, the caret (^) matches the start of the entire string, and the dollar sign ($) matches the end of the entire string.

i

Indicates that matching is case-insensitive for the letters "a" through "z" and "A" through "Z".

If the i flag is not specified, case-sensitive matching is done.

x

Indicates that whitespace characters (#x09, #x0A, #x0D, and #x20) within *pattern* are ignored, unless they are within a character class. Whitespace characters in a character class are never ignored.

If the x flag is not specified, whitespace characters are used for matching.

Returned value

If *source-string* is not the empty sequence, the returned value is an xs:string value that results when the following operations are performed on a copy of *source-string*:

- *source-string* is searched for characters that match *pattern*.
  - If two overlapping substrings of *source-string* match *pattern*, only the substring whose first character comes first in *source-string* is considered to match *pattern*.
  - If *pattern* contains two or more alternative sets of characters, and the alternative sets of characters match characters that start at the same position in *source-string*, the first set of characters in *pattern* that matches characters in *source-string* is considered to match *pattern*.
- Each set of characters in *source-string* that matches *pattern* is replaced with *replacement-string*.

If *pattern* is not found in *source-string*, *source-string* is returned.

If *pattern* matches a string of length zero, an error is returned.

If *source-string* is the empty sequence, a string of length 0 is returned.

Example

The following function replaces all instances of "a" followed by any single character or "b" followed by any single character with "*@".

```
fn:replace("abbcacadbdcd","(a(.))|(b(.))","*@")
```

The returned value is "*@*@*@*@*@cd".

Related reference

Regular expressions
A regular expression is a sequence of characters that act as a pattern for matching and manipulating strings. Regular expressions are used in the fn:matches, fn:replace, and fn:tokenize functions.

fn:round function

The fn:round function returns the integer that is closest to the specified numeric value.

Syntax



numeric-value

An atomic value or an empty sequence.

If *numeric-value* is an atomic value, it has one of the following types:

- xs:double
- xs:decimal

- xs:integer
- A type that is derived from any of the previously listed types

Returned value

If *numeric-value* is not the empty sequence, the returned value is the integer that is closest to *numeric-value*. The data type of the returned value depends on the data type of *numeric-value*:

- If *numeric-value* is xs:double, xs:decimal or xs:integer, the value that is returned has the same type as *numeric-value*.
- If *numeric-value* has a data type that is derived from xs:double, xs:decimal or xs:integer, the value that is returned has the direct parent data type of *numeric-value*.

If *numeric-value* is the empty sequence, the returned value is the empty sequence.

Examples

Example with a positive argument: The following function returns the rounded value of 0.5:

```
fn:round(0.5)
```

The returned value is 1.

Example with a negative argument: The following function returns the rounded value of (-1.5):

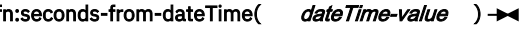
```
fn:round(-1.5)
```

The returned value is -1.

fn:seconds-from-datetime function

The `fn:seconds-from-dateTime` function returns the seconds component of an xs:dateTime value that is in its localized form.

Syntax



dateTime-value
The dateTime value from which the seconds component is to be extracted.

dateTime-value is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:decimal, and the value is greater than or equal to 0 and less than 60. The value is the seconds and fractional seconds component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Examples

The following function returns the seconds component of dateTime value for February 8, 2009 at 2:16:23 p.m. in the UTC-8 time zone.

```
fn:seconds-from-dateTime(xs:dateTime("2009-02-08T14:16:23-08:00"))
```

The returned value is 23.

The following function returns the seconds component of dateTime value for June 23, 2009 at 9:16:20.43 a.m. in the UTC time zone.

```
fn:seconds-from-dateTime(xs:dateTime("2009-06-23T09:16:20.43Z"))
```

The returned value is 20.43.

fn:seconds-from-duration function

The `fn:seconds-from-duration`function returns the seconds component of a duration.

Syntax



duration-value
The duration value from which the seconds component is to be extracted.

duration-value is an empty sequence, or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:dayTimeDuration, or is of type xs:duration, the returned value is of type xs:decimal, and is a value greater than -60 and less than 60. The value is the seconds and fractional seconds component of *duration-value* cast as xs:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:yearMonthDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The seconds and fractional seconds component of *duration-value* cast as xs:dayTimeDuration is computed as `(S mod 60)`. The value *S* is the total number of seconds and fractional seconds of *duration-value* cast as xs:dayTimeDuration to remove the years and months components.

Example

The following function returns the seconds component of the duration 150.5 seconds.

```
fn:seconds-from-duration(xs:dayTimeDuration("PT150.5S"))
```

The returned value is 30.5. When calculating the total number of seconds in the duration, 150.5 seconds is converted to 2 minutes and 30.5 seconds. The duration is equal to PT2M30.5S which has a seconds component of 30.5 seconds.

fn:seconds-from-time function

The `fn:seconds-from-time` function returns the seconds component of an xs:time value that is in its localized form.

Syntax



time-value
The time value from which the seconds component is to be extracted.

time-value is of type xs:time, or is an empty sequence.

Returned value

If *time-value* is of type xs:time, the returned value is of type xs:decimal, and the value is greater than or equal to zero and less than 60. The value is the seconds and fractional seconds component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the seconds component of the time value for 08:59:59 a.m. in the UTC-8 time zone.

```
fn:seconds-from-time(xs:time("08:59:59-08:00"))
```

The returned value is 59.

fn:starts-with function

The fn:starts-with function determines whether a string begins with a given substring. The substring is matched using the default collation.

Syntax



string
The string in which to search for *substring*.

string has the xs:string data type, or is the empty sequence. If *string* is the empty sequence, *string* is set to a string of length 0.

substring
The substring to search for.

substring has the xs:string data type, or is the empty sequence.

Returned value

The returned value is the xs:boolean value `true` if either of the following conditions are satisfied:

- *substring* occurs at the beginning of *string*.
- *substring* is an empty sequence or a string of length zero.

Otherwise, the returned value is `false`.

Example

The following function determines whether the string 'Test literal' begins with the string 'lite'.

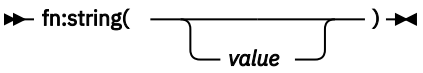
```
fn:starts-with('Test literal','lite')
```

The returned value is `false`.

fn:string function

The fn:string function returns the string representation of a value.

Syntax



value
The value that is to be represented as a string.

value is a node or an atomic value, or is the empty sequence.

If *value* is not specified, fn:string is evaluated for the current context item. If the current context item is undefined, an error is returned.

Returned value

If *value* is not the empty sequence, an xs:string value is returned:

- If *value* is a node, the returned value is the string value property of the *value* node.
- If *value* is an atomic value, the returned value is the result of casting *value* to the xs:string type.

If *value* is the empty sequence, the result is a string of length 0.

Example

The following function returns the string representation of 123:

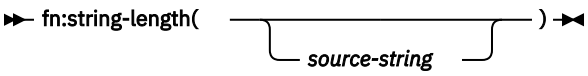
```
fn:string(xs:integer(123))
```

The returned value is '123'.

fn:string-length function

The fn:string-length function returns the length of a string.

Syntax



source-string
The string for which the length is to be returned.

source-string has the xs:string data type, or is an empty sequence.

Returned value

If *source-string* is not the empty sequence, the returned value is an xs:integer value that is the number of characters in *source-string*.

If *source-string* is the empty sequence, the returned value is the xs:integer value 0.

If *source-string* is not specified, the argument of fn:string-length defaults to the string value of the context item. If the context item is undefined, an error is raised.

Example

The following function returns the length of the string "Test literal".

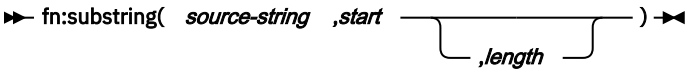
```
fn:string-length("Test literal")
```

The returned value is 12.

fn:substring function

The fn:substring function returns a substring of a string.

Syntax



source-string
The string from which the substring is retrieved.

source-string has the xs:string data type, or is an empty sequence.

start
The starting position in *source-string* of the substring. The first position of *source-string* is 1. If *start*<=0, *start* is set to 1.

start has the xs:double data type.

length
The length of the substring. The default for *length* is the length of *source-string*. If *start*+*length*-1 is greater than the length of *source-string*, *length* is set to (length of *source-string*)-*start*+1.

length has the xs:double data type.

Returned value

If *source-string* is not the empty sequence, the returned value is an xs:string value that is the substring of *source-string* that starts at position *start* and is *length* bytes. If *source-string* is the empty sequence, the result is a string of length 0.

Example

The following function returns seven characters starting at the sixth character of the string 'Test literal'.

```
fn:substring('Test literal',6,7)
```

The returned value is 'literal'.

fn:sum function

The fn:sum function returns the sum of the values in a sequence.

Syntax

sequence-expression
A sequence that contains items of any of the following atomic types, or an empty sequence:

- xs:double
- xs:decimal
- xs:integer
- A type that is derived from any of the previously listed types
- xs:dayTimeDuration
- xs:yearMonthDuration
- xs:untypedAtomic

All values in the sequence must be of the same type or a derived type of the type, or must be promotable to a single common type. An xs:untypedAtomic value is promoted to the xs:double data type. A derived type is promoted to its direct parent data type.

Returned value

If *sequence-expression* is not the empty sequence, the returned value is the sum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* is the empty sequence, fn:sum returns 0.0E0.

Example

The following function returns the sum of the sequence (5, 1.0E2, 4.5):

```
fn:sum(5, 1.0E2, 40.5)
```

The values are promoted to the xs:double data type. The returned value is 1.455E2.

fn:timezone-from-date function

The `fn:timezone-from-date` function returns the time zone component of an xs:date value.

Syntax



date-value
The date value from which the time zone component is to be extracted.

date-value is of type xs:date, or is an empty sequence.

Returned value

If *date-value* is of type xs:date and has an explicit time zone component, the returned value is of type xs:dayTimeDuration, and the value is between -PT14H hours and PT14H, inclusive. The value is the deviation of the *date-value* time zone component from the UTC time zone.

If *date-value* does not have an explicit time zone component or is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the time zone component of the date value for March 13, 2009 in the UTC-8 time zone.

```
fn:timezone-from-date(xs:date("2009-03-13-08:00"))
```

The returned value is -PT8H.

fn:timezone-from-dateTime function

The `fn:timezone-from-dateTime` function returns the time zone component of an xs:dateTime value.

Syntax



dateTime-value
The dateTime value from which the time zone component is to be extracted.

dateTime-value is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime and has an explicit time zone component, the returned value is of type xs:dayTimeDuration, and the value is between -PT14H and PT14H, inclusive. The value is the deviation of the *dateTime-value* time zone component from the UTC time zone.

If *dateTime-value* does not have an explicit time zone component, or is an empty sequence, the returned value is an empty sequence.

Examples

The following function returns the time zone component of the dateTime value for October 30, 2009 at 7:30 a.m. in the UTC-8 time zone.

```
fn:timezone-from-dateTime(xs:dateTime("2009-10-30T07:30:00-08:00"))
```

The returned value is -PT8H.

The following function returns the time zone component of the dateTime value for January 1, 2009 at 2:30 p.m. in the UTC+10:30 time zone.

```
fn:timezone-from-dateTime(xs:dateTime("2009-01-01T14:30:00+10:30"))
```

The returned value is PT10H30M.

fn:timezone-from-time function

The `fn:timezone-from-time` function returns the time zone component of an xs:time value.

Syntax



time-value
The time value from which the time zone component is to be extracted.

time-value is of type xs:time, or is an empty sequence.

Returned value

If *time-value* is of type xs:time and has an explicit time zone component, the returned value is of type xs:dayTimeDuration, and the value is between -PT14H and PT14H, inclusive. The value is the deviation of the *time-value* time zone component from UTC time zone.

If *time-value* does not have an explicit time zone component, or is an empty sequence, the returned value is an empty sequence.

Examples

The following function returns the time zone component of the time value for 12 noon in the UTC-5 time zone.

```
fn:timezone-from-time(xs:time("12:00:00-05:00"))
```

The returned value is -PT5H.

In the following function, the time value for 1:00 p.m. does not have a time zone component.

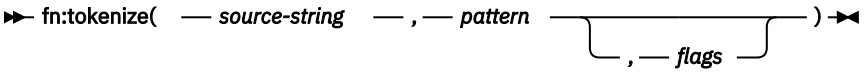
```
fn:timezone-from-time(xs:time("13:00:00"))
```

The returned value is the empty sequence.

fn:tokenize function

The fn:tokenize function breaks a string into a sequence of substrings.

Syntax



source-string
A string that is to be broken into a sequence of substrings.

source-string is a literal string, or an XQuery expression that resolves to an xs:string value or the empty sequence.

pattern
The delimiter between substrings in *source-string*.

pattern is a string literal that contains a regular expression. A regular expression is a set of characters, pattern-matching characters, and operators that define a string or group of strings in a search pattern.

flags
A string literal that can contain any of the following values that control matching of *pattern* to *source-string*:

s
Indicates that the dot (.) matches any character.

If the s flag is not specified, the dot (.) matches any character except the new line character (#x0A).

m
Indicates that the caret (^) matches the start of any line (the position after a new line character), and the dollar sign ($) matches the end of any line (the position before a new line character).

If the m flag is not specified, the caret (^) matches the start of the entire string, and the dollar sign ($) matches the end of the entire string.

i
Indicates that matching is case-insensitive for the letters "a" through "z" and "A" through "Z".

If the i flag is not specified, case-sensitive matching is done.

x
Indicates that whitespace characters (#x09, #x0A, #x0D, and #x20) within *pattern* are ignored, unless they are within a character class. Whitespace characters in a character class are never ignored.

If the x flag is not specified, whitespace characters are used for matching.

Returned value

If *source-string* is not the empty sequence or a zero-length string, the returned value is a sequence of xs:string values that results when the following operations are performed on *source-string*:

- *source-string* is searched for characters that match *pattern*.

- If *pattern* contains two or more alternative sets of characters, and the alternative sets of characters match characters that start at the same position in *source-string*, the first set of characters in *pattern* that matches characters in *source-string* is considered to match *pattern*.
- Each set of characters that does not match *pattern* becomes an item in the result sequence.
- If *pattern* matches characters at the beginning of *source-string*, the first item in the returned sequence is a string of length 0.
- If two successive matches for *pattern* are found within *source-string*, a string of length 0 is added to the sequence.
- If *pattern* matches characters at the end of *source-string*, the last item in the returned sequence is a string of length 0.

If *pattern* is not found in *source-string*, *source-string* is returned.

If *pattern* matches a string of length zero, an error is returned.

If *source-string* is the empty sequence, or is a zero-length string, the result is the empty sequence.

Example

The following function creates a sequence from the string "?A?B?C?D?" by removing the question mark (?) characters and creating a sequence from the remaining characters.

```
fn:tokenize("?A?B?C?D?","\?")
```

The returned value is the sequence ("", "A", "B", "C", "D", "").

Related reference

Regular expressions
A regular expression is a sequence of characters that act as a pattern for matching and manipulating strings. Regular expressions are used in the fn:matches, fn:replace, and fn:tokenize functions.

fn:translate function

The fn:translate function replaces selected characters in a string with replacement characters.

Syntax



source-string
The string in which characters are to be converted.

source-string has the xs:string data type, or is the empty sequence.

original-string
A string that contains the characters that can be converted.

original-string has the xs:string data type.

replacement-string
A string that contains the characters that replace the characters in *original-string*.

replacement-string has the xs:string data type.

If the length of *replacement-string* is greater than the length of *original-string*, the additional characters in *replacement-string* are ignored.

Returned value

If *source-string* is not the empty sequence, the returned value is the xs:string value that results when the following operations are performed:

- For each character in *source-string* that appears in *original-string*, replace the character in *source-string* with the character in *replacement-string* that appears at the same position as the character in *original-string*.

  If the length of *original-string* is greater than the length of *replacement-string*, delete each character in *source-string* that appears in *original-string*, but the character position in *original-string* does not have a corresponding position in *replacement-string*.

  If a character appears more than once in *original-string*, the position of the first occurrence of the character in *original-string* determines the character in *replacement-string* that is used.
- For each character in *source-string* that does not appear in *original-string*, leave the character as it is.

If *source-string* is the empty sequence, a string of length 0 is returned.

Example

The following function replaces the character a with the character A and deletes any - characters from the string "—aaa—".

```
fn:translate("---aaa---","a-","A")
```

The returned value is "AAA".

fn:upper-case function

The fn:upper-case function converts a string to uppercase.

Syntax



source-string
The string that is to be converted to uppercase.

source-string has the xs:string data type, or is an empty sequence.

Returned value

If *source-string* is not an empty sequence, the returned value is the xs:string value *source-string*, with each character converted to its uppercase correspondent. Every character that does not have an uppercase correspondent is included in the returned value in its original form.

If *source-string* is the empty sequence, the returned value is a string of length zero.

Examples

The following function converts the string 'Test literal 1' to uppercase.

```
fn:upper-case("Test literal 1")
```

The returned value is "TEST LITERAL 1".

The argument of the following function resolves to "ii".

```
fn:upper-case("&#x131;i")
```

The returned value is "II".

fn:year-from-date function

The `fn:year-from-date` function returns the year component of an xs:date value that is in its localized form.

Syntax



date-value
The date value from which the year component is to be extracted.

date-value is of type xs:date, or is an empty sequence.

Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer, The value is the year component of the *date-value*, a non-negative value.

If *date-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the year component of the date value for October 29, 2009.

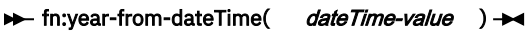
```
fn:year-from-date(xs:date("2009-10-29"))
```

The returned value is 2009.

fn:year-from-datetime function

The `fn:year-from-dateTime` function returns the year component of an xs:dateTime value that is in its localized form.

Syntax



dateTime-value
The dateTime value from which the year component is to be extracted.

dateTime-value is of type xs:dateTime, or is an empty sequence.

Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer. The value is the year component of the *dateTime-value*, a non-negative value.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

Example

The following function returns the year component of the dateTime value for October 29, 2009 at 8:00 a.m. in the UTC-8 time zone.

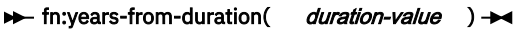
```
fn:year-from-dateTime(xs:dateTime("2009-10-29T08:00:00-08:00"))
```

The returned value is 2009.

fn:years-from-duration function

The `fn:years-from-duration` function returns the years component of a duration.

Syntax



duration-value
The duration value from which the years component is to be extracted.

duration-value is an empty sequence, or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:yearMonthDuration or is of type xs:duration, the returned value is of type xs:integer. The value is the years component of *duration-value* cast as xs:yearMonthDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:dayTimeDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The years component of *duration-value* cast as xs:yearMonthDuration is the integer number of years determined by the total number of months of *duration-value* cast as xs:yearMonthDuration divided by 12.

Examples

The following function returns the years component of the duration -4 years, -11 months, and -320 days.

```
fn:years-from-duration(xs:duration("-P4Y11M320D"))
```

The returned value is -4.

The following function returns the years component of the duration 9 years and 13 months.

```
fn:years-from-duration(xs:yearMonthDuration("P9Y13M"))
```

The returned value is 10. When calculating the total number of years in the duration, 13 months is converted to 1 year and 1 month. The duration is equal to P10Y1M which has a years component of 10 years.

Information resources for Db2 11 for z/OS and related products

Information about Db2 11 for z/OS and products that you might use in conjunction with Db2 11 is available online in IBM Knowledge Center.

You can find the complete set of product documentation for Db2 11 for z/OS in IBM Knowledge Center.

You can also download other PDF format manuals for Db2 11 for z/OS from IBM Knowledge Center in PDF format manuals for Db2 11 for z/OS (Db2 Home).

© Copyright IBM Corp. 2007, 2020

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing Legal and Intellectual Property Law IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

© Copyright IBM Corp. 2007, 2020

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as shown below:

© (*your company name*) (*year*).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. (*enter the year or years*).

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

General-use Programming Interface and Associated Guidance Information

This information is intended to help you write applications that access XML data on Db2 11 for z/OS servers. This information primarily documents General-use Programming Interface and Associated Guidance Information provided by Db2 11 for z/OS. However, this information also documents Product-sensitive Programming Interface and Associated Guidance Information.

General-use Programming Interface and Associated Guidance Information

General-use Programming Interfaces allow the customer to write programs that obtain the services of Db2 11 for z/OS.

Product-sensitive Programming Interface and Associated Guidance Information

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:

PSPI Product-sensitive Programming Interface and Associated Guidance Information... PSPI

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered marks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at: http://www.ibm.com/legal/copytrade.shtml.

Linux® is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions:

Applicability: These terms and conditions are in addition to any terms of use for the IBM website.

Personal use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights: Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Glossary

The glossary is available in IBM Knowledge Center.

See the Glossary topic for definitions of Db2 for z/OS terms.

© Copyright IBM Corp. 2007, 2020

Index

A

abbreviated syntax
  path expression 194
abs function 243
accessibility
  keyboard x
  shortcut keys x
adding nodes 231
adjust-date-to-timezone function 244
adjust-datetime-to-timezone function 245
adjust-time-to-timezone function 247
application development
  XML data 107
archiving
  XML data 122
arithmetic expressions 197
atomic values 3
attribute axis 190
attribute nodes 8
attributes
  namespace declaration 210
avg function 248
axis
  attribute 190
  child 190
  descendant 190
  descendant-or-self 190
  parent 190
  self 190
axis step 189

B

binary XML format
  inserting XML data 33
  serializing XML data 77
  utilities 95
  XML data type 14
binding IBM Data Server Driver for JDBC and SQLJ packages
  XML schema repository setup 25
boolean function 249
boundary whitespace
  declaration 180
  direct element constructors 211
boundary-space declarations 180
byte order mark (BOM)
  XML data 123

C

case sensitivity
  XQuery 159
castable expressions 237
casting
  SQL types to XML types 57
casting *(continued)*
  XML schema types to SQL types 57
  XML schema types to SQL types, examples 57
casts between XML schema data types
  list 175
CCSID-to-encoding-name mappings
  textual XML data 148
CHECK DATA utility
  XML support 95
CHECK INDEX utility
  XML support 95
child axis 190
comment node 11
comments
  constructing, XQuery 217
  direct constructors 217
  XPath 160
compare function 250
comparison
  general 202
comparison expressions
  nodes 204
  values 200
  XQuery 200
concat function 251
conditional expressions 228
constructors
  built-in types 163
  direct comment 217
  direct element 207
  direct processing instruction 216
  document node 215
  enclosed expressions 206
  in-scope namespaces 213
  namespace declaration attributes 210
  processing instruction 216
  XML 206
contains function 251
context item expression 187
copy-namespace declarations 180
count function 252
current-date function 252
current-dateTime function 252
current-time function 253

D

data function 253
data model generation
  XQuery 11
data types
  promotion 183
  substitution 184
  testing cast of a value (DB2 XQuery) 237
  XML indexes 84
  xs:dayTimeDuration 171
  xs:decimal 166

data types *(continued)*
xs:duration 172
xs:yearMonthDuration 174
dateTime function 254
day-from-date function 254
day-from-dateTime function 255
days-from-duration function 255
dayTimeDuration data type
description 171
normalized form 171
DB2 XQuery functions
abs 243
adjust-date-to-timezone 244
adjust-time-to-timezone 247
avg 248
boolean 249
compare 250
concat 251
contains 251
count 252
current-date 252
current-dateTime 252
current-time 253
data 253
dateTime 254
day-from-date 254
day-from-dateTime 255
days-from-duration 255
hours-from-dateTime 257
hours-from-duration 257
hours-from-time 258
implicit-timezone function 258
minutes-from-dateTime 259
minutes-from-duration 259
minutes-from-time 260
month-from-date 260
month-from-dateTime 261
months-from-duration 261
normalize-space 262
not 270
round 272
seconds-from-dateTime 273
seconds-from-duration 274
seconds-from-time 274
string 275
string-length 276
substring 277
sum 277
timezone-from-date 278
timezone-from-dateTime 278
timezone-from-time 279
year-from-date 283
year-from-dateTime 283
years-from-duration 284
Db2-supplied stored procedures
XML schema 104
declarations
boundary-space 180
copy-namespaces 180
default namespace 182
namespace 181
default column value
XMLTABLE 68
default namespace declaration
default namespace declaration *(continued)*
XQuery 182
defining stored procedures
XML schema repository 25
delete expression 230
deleting nodes 230
descendant axis 190
descendant-or-self axis 190
differences in XML data
before and after storage and retrieval 79
direct constructors
comment 217
description 206
element 207
processing instruction 216
whitespace in element 211
disability x
distinct-values function 256
document node constructors
description 206
document nodes
constructing 215
description 5
documents, XML
deleting 38
down-level clients
retrieving XML data 72
DSN_XMLVALIDATE
choosing XML schema 54
moving from user-defined function to built-in function 53
XML schema validation 53
DTDs
XML parsing 43
duration data type
description 172
normalized form 172

E

element nodes 6
elements
direct constructors 207
in-scope namespaces 213
embedded SQL applications
host variables, XML data 108
XML data 108
enclosed expressions
constructors 206
encoding considerations
XML in Java 125
encoding scenarios
internally encoded XML 126
XML 126
encoding-name-to-CCSID mappings
textual XML data 135
entity references
predefined 185
evaluating expressions 183
examples
XML column updates 118
XML index 87–92
XSR_ADDSCHEMADOC 104
XSR_COMPLETE 104

examples *(continued)*
XSR_REGISTER 104
XSR_REMOVE 104
EXEC SQL utility
XML support 95
explicit XML parsing
XMLPARSE 42
expressions
arithmetic 197
atomization 183
castable 237
conditional 228
constructing sequences 196
constructors, description 206
constructors, namespace declaration attributes 210
context item 187
delete 230
direct comment constructors 217
direct element constructors 207
direct processing instruction constructors 216
document node constructors 215
enclosed in constructors 206
filter 196
FLWOR, examples 226
FLWOR, for and let clauses together 221
FLWOR, for and let clauses, overview 219
FLWOR, for and let clauses, variable scope 221
FLWOR, for clauses 219
FLWOR, let clauses 220
FLWOR, order by clauses 224
FLWOR, overview 217
FLWOR, return clauses 226
FLWOR, syntax 218
FLWOR, where clauses 223
in-scope namespaces constructors 213
insert 231
logical 205
node comparisons 204
parenthesized 186
path 188
processing 183
processing instruction constructors 216
replace 235
sequence 196
subtype substitution 184
type promotion 183
value comparisons 200
external encoding
XML data 123
external encoding input scenarios
XML data 128

F

filter expression 196
FLWOR expressions
example 226
for and let clauses, same expression 221
for and let clauses, variable scope 221
for clauses 219
let clauses 220
order by clauses 224
overview 217
return clauses 226
FLWOR expressions *(continued)*
syntax 218
where clauses 223
for clause
XQuery 219
function call
XQuery 187
functions
adjust-date-to-timezone 244
adjust-time-to-timezone 247
current-date 252
current-dateTime 252
current-time 253
dateTime 254
day-from-date 254
day-from-dateTime 255
days-from-duration 255
distinct-values 256
hours-from-dateTime 257
hours-from-duration 257
hours-from-time 258
implicit-timezone 258
last 263
local-name 264
lower-case 265
matches 265
max 267
min 268
minutes-from-dateTime 259
minutes-from-duration 259
minutes-from-time 260
month-from-date 260
month-from-dateTime 261
months-from-duration 261
name 269
position 270
replace 271
seconds-from-dateTime 273
seconds-from-duration 274
seconds-from-time 274
starts-with 275
timezone-from-date 278
timezone-from-dateTime 278
timezone-from-time 279
tokenize 280
translate 281
upper-case 282
year-from-date 283
year-from-dateTime 283
years-from-duration 284

G

general comparisons
XQuery 202
general-use programming information, described 288

H

host variables
XML in assembler 109
XML in C language 109, 110
XML in COBOL 110, 111

host variables *(continued)*
XML in embedded SQL applications 108
XML in PL/I 112, 113
hours-from-dateTime function 257
hours-from-duration function 257
hours-from-time function 258

I

if-then-else expressions
description 228
ignorable whitespace
removing, XML schema validation 45
implicit serialization
encoding scenarios, XML data 130
implicit-timezone function 258
implicitly created objects
XML columns 29
in-scope namespaces
constructed elements 213
index
access methods
XML indexes 86
indexing
XML data 81
inline COPY
XML support 95
inline statistics
XML support 95
input of XML data 124
insert expression 231
inserting nodes 231
internal encoding
background information 123
description 123
XML data input 126

K

kind test 191

L

last function 263
let clause
description 220
links
non-IBM Web sites 289
LISTDEF utility
XML support 95
literal
XQuery 184
LOAD utility
XML support 95
local-name function 264
lock avoidance
XML versions 39
logical expression 205
lower-case function 265

M

matches function 265
max function 267
min function 268
minutes-from-dateTime function 259
minutes-from-duration function 259
minutes-from-time function 260
month-from-date function 260
month-from-dateTime function 261
months-from-duration function 261
moving from user-defined function to built-in function
DSN_XMLVALIDATE 53

N

name function 269
name test 191
namespace declarations
attributes 210
XML index 83
XQuery 181
namespaces
binding a prefix 210
default element 210
description 158
in-scope 213
setting default 210
native SQL routines
XML parameters 70
node test 191
node values
replacing 235
nodes
adding 231
attribute 8
comment 11
comparing 204
constructing comment 217
constructing document 215
constructing processing instruction 216
deleting 230
direct comment constructors 217
document 5
element 6
overview 3
processing instruction 10
replacing 235
text 9
normalize-space function 262
normalized form
dayTimeDuration data type 171
duration data type 172
yearMonthDuration data type 174
not function 270
numeric data types
range 168
numeric literal 184

O

order by clause 224
order of processing
order by clauses 224
ordinality column
XMLTABLE 69

output of XML data 125

P

parent axis 190
parenthesized expression
  XQuery 186
parsing
  XML 42
partial XML revalidation
  XML type modifier 51
path expression
  abbreviated syntax 194
  definition 188
pattern expression
  XML index 82
position function 270
predefined entity reference
  XQuery 185
predicate
  XQuery 194
processing instruction node
  constructing 216
  description 10
product-sensitive programming information, described 288
programming interface information, described 288
prolog
  boundary-space declarations 180
  XQuery 179
prologs
  copy-namespace declarations 180
PSPI symbols 288
publishing functions
  special character handling 76
  XML 72
pureXML
  data model 2
  software prerequisites 17
  tutorial 15

Q

qualified names (QNames)
  XQuery 158
queries
  XML data 60
querying XML
  with XMLTABLE 65
  XMLTABLE overview 65
QUIESCE utility
  XML support 95

R

REBUILD INDEX utility
  XML support 95
RECOVER INDEX utility
  XML support 95
RECOVER TABLESPACE utility
  XML support 95
regular expression
  description 239
  syntax 239
removing nodes
  XQuery 230
REORG INDEX utility
  XML support 95
REORG TABLESPACE utility
  XML support 95
REPAIR utility
  XML support 95
replace expression 235
replace function 271
replacing node values
  XQuery 235
replacing nodes
  XQuery 235
REPORT utility
  XML support 95
return clause 226
round function 272
routines
  XML variables 70
RUNSTATS utility
  XML support 95

S

seconds-from-dateTime function 273
seconds-from-duration function 274
seconds-from-time function 274
self axis 190
sequence expression 196
sequences
  atomization 183
  constructing 196
  description 2
serialization, XML
  explicit 77
  implicit 77
shortcut keys
  keyboard x
software prerequisites
  pureXML 17
special characters
  publishing functions 76
SQL/XML functions
  XMLTABLE overview 65
  XMLTABLE usage 65
starts-with function 275
statically known namespaces 213
step
  axis 189
string function 275
string literals
  XQuery 184
string-length function 276
substring function 277
subtype substitution 184
sum function 277
syntax
  FLWOR expressions 218
  XQuery expression 179
syntax diagram
  how to read xi

T

table spaces
  XML data 27
tables
  adding XML columns 27
  creating with XML columns 27
text node 9
textual XML data
  CCSID-to-encoding-name mappings 148
  encoding-name-to-CCSID mappings 135
  serializing 77
time zone
  implicit 258
timezone-from-date function 278
timezone-from-dateTime function 278
timezone-from-time function 279
tokenize function 280
translate function 281
trigger
  XML column 41
tutorial
  pureXML 15
type promotion
  XQuery 183
types
  generic 164
  numeric, range 168
  overview 163
  untyped data 165
  xs:anyAtomicType 165
  xs:anySimpleType 164
  xs:anyType 164
  xs:boolean 168
  xs:date 169
  xs:dateTime 170
  xs:double 167
  xs:integer 167
  xs:string 166
  xs:time 173
  xs:untyped 165
  xs:untypedAtomic 165

U

UNIQUE keyword
  XML index 85
UNLOAD utility
  XML support 95
untyped data 165
upper-case function 282
URI
  binding a namespace prefix 210

V

validation
  XML 45
value comparisons 200
variable references
  XQuery 186
variables
  in scope, for and let clauses 221
variables *(continued)*
  positional, for clauses 219

W

where clause
  XQuery 223
whitespace
  boundary 180
  in direct element constructors 211
  removal, XML schema validation 45
  XMLPARSE 42
  XQuery 160
WLM environment setup
  C language, XML schema repository 19
  Java language, XML schema repository 21

X

XML
  application development 107
  casting variables to XML types 64
  column updates 118
  data retrieval 60, 120
  data types 118
  filtering with XMLEXISTS 63
  inserting XMLTABLE results 66
  overview of Db2 for z/OS support 1
  parameters in native SQL routines 70
  parsing 42
  querying with XMLQUERY 61
  querying with XMLTABLE 65
  querying with XMLTABLE, example 67
  validation 45
  variables in routines 70
XML character reference 185
XML columns
  altering definitions 27
  defining 27
  implicitly created objects 29
  retrieving entire XML document 60
XML data
  archiving 122
  before and after storage and retrieval 79
  embedded SQL applications 108
  encoding considerations for input 124
  encoding considerations for Java 125
  encoding considerations for output 125
  encoding scenarios for input, internally encoded 126
  encoding scenarios, explicitly serializing 132
  encoding scenarios, implicitly serializing 130
  external encoding 123
  external encoding input scenarios 128
  indexing 81
  inserting in a table 33
  internal encoding 123
  Java applications 107, 125
  retrieving from tables, embedded SQL applications 115
  storage structure 29
  updating entire document 35
  updating in a table 35
  updating part of a document 36
  updating, embedded SQL applications 113

XML data *(continued)*
working with 27
XML data retrieval
earlier clients 72
entire XML document 60
using XMLTABLE 65
XMLTABLE advantages 65
XML data type 14
XML declaration
XML data 123
XML document
versions 39
XML documents
determining whether validated 56
XML encoding considerations
input 124
output 125
XML index
data types 84
data types, XML schema 84
examples 87–92
namespace declaration 83
pattern expression 82
UNIQUE keyword 85
XML model
comparison to relational model 12
XML publishing functions
description 72
handling special characters 76
XML revalidation
XML type modifier 51
XML schema
data types, casts 175
index data types 84
XML schema registration
stored procedure examples 104
XML schema repository
binding stored procedures 25
setting up 18
support in Db2 for z/OS 103
testing 26
XML schema repository stored procedures
defining 25
WLM environment, C language 19
WLM environment, Java language 21
XML schema to SQL
casting 57
XML schema validation
choosing XML schema, DSN_XMLVALIDATE 54
DSN_XMLVALIDATE 53
XML type modifier 46
XML type modifier, choosing schema 48
XML type modifier
adding during ALTER TABLE 27
adding during CREATE TABLE 27
choosing XML schema 48
partial XML revalidation 51
XML schema validation 46
XML versions 39
XML virtual storage
limitation of usage 32
XMLCAST
casting to from XML to SQL 57
XMLEXISTS
XMLEXISTS *(continued)*
casting variables to XML types 64
filtering XML data 63
XMLPARSE
DTD handling 43
whitespace handling 42
XMLQUERY
casting variables to XML types 64
description 61
returning empty sequences 62
returning non-empty sequences 61
XMLSERIALIZE
earlier clients 72
XMLTABLE function
advantages 65
example 66–69
overview 65
XMLVALA
limitation of XML virtual storage usage for a user 32
XMLVALS
limitation of XML virtual storage usage for a subsystem 32
XMLXSROBJECTID
checking for XML validation 56
XPath
numeric data types 166
when to use 157
XQuery
date and time types 169
expressions 183
function reference 243
overview 155
primary expressions 184
updating expressions 230
variable reference 186
when to use 157
XQuery expression
example 179
general format 179
XQuery expression, structure 179
XQuery function call 187
XQuery functions
adjust-datetime-to-timezone 245
distinct-values 256
last 263
local-name 264
lower-case 265
matches 265
max 267
min 268
name 269
position 270
replace 271
starts-with 275
timezone-from-dateTime 278
tokenize 280
translate 281
upper-case 282
XQuery predicate 194
XQuery prolog 179
XQuery type system
overview 163
xs:anyAtomicType 165
xs:anySimpleType 164

xs:anyType 164
xs:boolean 168
xs:date 169
xs:dateTime 170
xs:decimal 166
xs:double 167
xs:integer 167
xs:string 166
xs:time 173
xs:untyped 165
xs:untypedAtomic 165
XSLTRANSFORM 80
XSR_ADDSCHEMADOC
    stored procedure, add to XML schema 104
XSR_COMPLETE
    stored procedure, complete schema registration 104
XSR_REGISTER
    stored procedure, XML schema registration 104
XSR_REMOVE
    stored procedure, remove XML schema 104

Y

year-from-date function 283
year-from-dateTime function 283
yearMonthDuration data type 174
years-from-duration function 284



Product Number: 5615-DB2
5697-P43

SC19-4064-06

