

DB2®

IBM

DB2 Version 9
for Linux, UNIX, and Windows

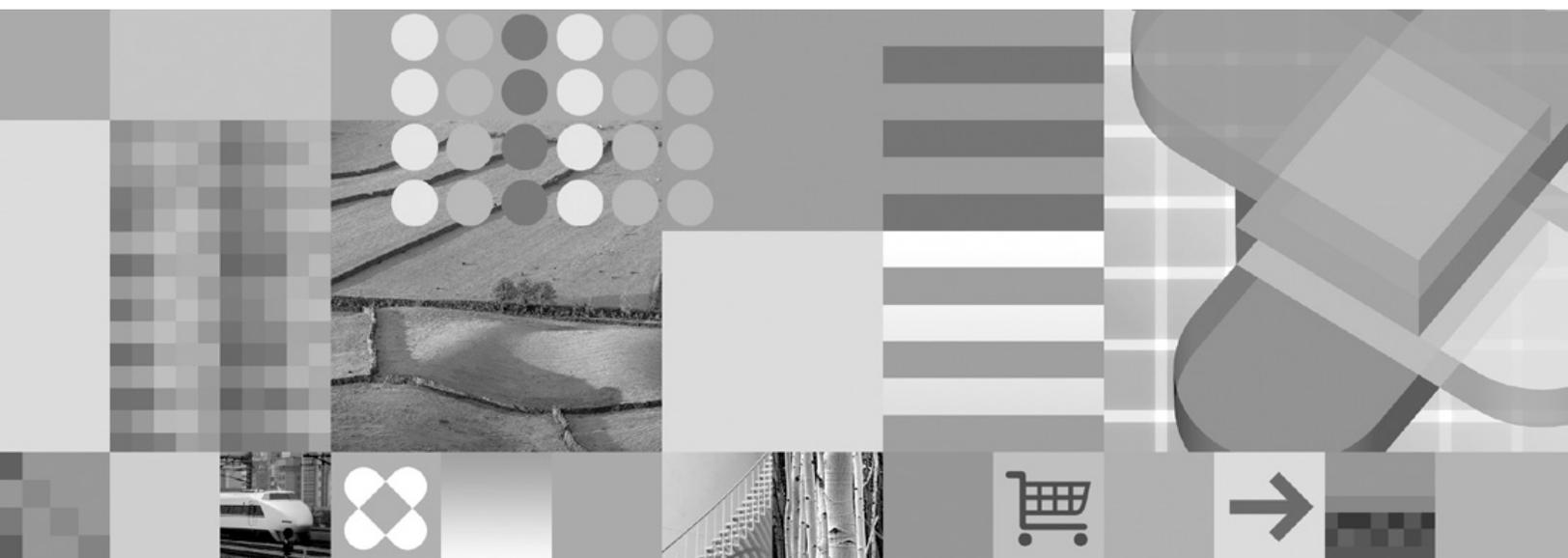


XML Guide

DB2

IBM

DB2 Version 9
for Linux, UNIX, and Windows



XML Guide

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

Edition Notice

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

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

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

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

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

© Copyright International Business Machines Corporation 2006. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. Introduction 1

Native XML data store overview	1
XML input and output overview	3
Comparison of the XML model and the relational model	7
XQuery and XPath data model	9
Sequences and items.	9
Atomic values	10
Node hierarchies	10
Node properties	12
Node kinds	13
Document nodes	13
Element nodes	13
Attribute nodes	14
Text nodes.	14
Processing instruction nodes.	14
Comment nodes	15
Document order of nodes.	15
Node identity	15
Typed values and string values of nodes	16
XML data type	17
Tutorial for native XML data store.	18
XML information roadmap	28

Chapter 2. XML storage 29

XML storage object overview	29
Guidelines for storage requirements for XML documents.	29

Chapter 3. Working with XML data 31

Creation of tables with XML columns.	31
Addition of XML columns to existing tables	32
Insertion into XML columns.	32
Updates of XML columns.	34
Deletion of XML data from tables	36
XML support in triggers	37
XML parsing	37
XML validation	41
Querying and retrieving XML	45
XML data querying.	45
Introduction to XQuery	45
Retrieving DB2 data with XQuery functions	46
Introduction to querying XML data with SQL	48
Comparison of XQuery to SQL	48
Comparison of methods for querying XML data	49
XMLQUERY scalar function	51
XMLQUERY overview.	51
Non-empty sequences returned by XMLQUERY	52
Empty sequences returned by XMLQUERY.	53
Casting of XMLQUERY results to non-XML types	55
Casting between data types	56
XMLTABLE table function	61
XMLTABLE overview	61

XMLTABLE example: Inserting values returned from XMLTABLE	63
XMLTABLE example: Returning one row for each occurrence of an item	65
XMLEXISTS predicate	66
XMLEXISTS predicate when querying XML data	66
XMLEXISTS predicate usage.	67
Constant and parameter marker passing to XMLEXISTS and XMLQUERY	69
Data retrieval with XQuery	70
Guidelines for matching indexes with queries	72
Guidelines for matching indexes with queries overview	72
Restrictiveness of index definitions	74
Considerations when specifying text()	75
Data type of literals	76
Join predicate conversion.	77
Indeterminate query evaluation.	79
Full-text search in XML documents	79
Retrieval of data in XML columns to earlier DB2 clients	80
Publishing.	81
Publishing XML values with SQL/XML	81
Special character handling in SQL/XML publishing functions	85
XML serialization	86
Differences in an XML document after storage and retrieval	88
Data types for archiving XML documents	89
Tools that support XML	90

Chapter 4. XML data indexing 93

Overview	93
XML data indexing overview	93
Index XML pattern expressions.	94
XML namespace declarations	96
Data types associated with index XML pattern expressions	97
Indexing elements with complex schema types	99
Data type conversion for indexes over XML data	102
XML schemas and index key generation	104
UNIQUE keyword semantics	105
Database objects associated with XML data indexing	106
Logical and physical indexes over XML data	106
Other database objects associated with XML columns	107
CREATE INDEX	108
Sample queries against indexes over XML data	124
Restrictions on indexes over XML data	126
Recreation of indexes over XML data	127
Troubleshooting	127
Common XML indexing issues	127

Preference of database managed table spaces for native XML data store performance 131

Chapter 5. XML schema repository (XSR) 133

XML schema, DTD, and external entity management using the XML schema repository (XSR)	133
XSR objects	133
XSR object registration	134
Registering XSR objects through stored procedures	135
Registering XSR objects through the command line processor	136
Java support for XML schema registration and removal	137
Altering registered XSR objects	139
Examples of extracting XML schema information	139
Listing XML schemas registered with the XSR	139
Retrieving all components of an XML schema registered with the XSR	140
Retrieving the XML schema of an XML document	140
Stored procedures	141
XSR_REGISTER procedure	141
XSR_ADDSCHEMADOC procedure	142
XSR_COMPLETE procedure	143
XSR_DTD procedure	144
XSR_EXTENTENTITY procedure	145
Commands	147
REGISTER XMLSCHEMA	147
ADD XMLSCHEMA DOCUMENT	149
COMPLETE XMLSCHEMA	150
REGISTER XSROBJECT	151

Chapter 6. Data movement utility support 155

XML data movement overview	155
Important considerations for XML data movement	156
XQuery data model	156
LOB and XML file behavior with regard to import and export	157
XML data specifier	158
Exporting XML data	159
Importing XML data	162

Chapter 7. Application programming and routines support 165

Application programming language support for XML	165
CLI	167
XML data handling in CLI applications - Overview	167
XML column inserts and updates in CLI applications	168
XML data retrieval in CLI applications	169
Changing of default XML type handling in CLI applications	170
Embedded SQL	171
Declaring XML host variables in embedded SQL applications	171

Example: Referencing XML host variables in embedded SQL applications	173
Executing XQuery expressions in embedded SQL applications	174
Recommendations for developing embedded SQL applications with XML and XQuery	176
Identifying XML values in an SQLDA	177
Java	177
JDBC	177
XML data in JDBC applications	177
XML column updates in JDBC applications	178
XML data retrieval in JDBC applications	180
Invocation of routines with XML parameters in Java applications	182
SQLJ	183
XML data in SQLJ applications	183
XML column updates in SQLJ applications	184
XML data retrieval in SQLJ applications	186
PHP	187
Introduction to PHP application development for DB2	187
Executing XQuery expressions in PHP (ibm_db2)	188
Routines	189
SQL procedures	189
XML and XQuery support in SQL procedures	189
Cursors for XQuery expressions in SQL procedures	191
Effect of commits and rollbacks on XML parameter and variable values in SQL procedures	193
External routines	194
XML data type support in external routines	194
Specification of a driver for Java routines	195
Example: XML and XQuery support in Java (JDBC) procedure	196
Example: XML and XQuery support in C# .NET CLR procedure	200
Example: XML and XQuery support in C procedure	204
Performance of routines	208

Chapter 8. XML encoding 217

XML data encoding	217
Background information on XML internal encoding	217
Considerations	219
Encoding considerations for input of XML data to a database	219
Encoding considerations for retrieval of XML data from a database	219
Encoding considerations for passing XML data in routine parameters.	220
Encoding considerations for XML data in JDBC, SQLJ, and .NET applications	221
Scenarios	222
Encoding scenarios for input of internally encoded XML data to a database	222
Encoding scenarios for input of externally encoded XML data to a database	224
Encoding scenarios for retrieval of XML data with implicit serialization	226

Encoding scenarios for retrieval of XML data with explicit XMLSERIALIZE	228	rowSets in annotated XML schema decomposition	294
Chapter 9. Restrictions on native XML data store	233	Examples of mappings in annotated XML schema decomposition	296
Chapter 10. Annotated XML schema decomposition	235	Decomposition annotation example: Mapping to an XML column	297
Annotated XML schema decomposition	235	Decomposition annotation example: A value mapped to a single table that yields a single row	298
Annotated XML schema decomposition	235	Decomposition annotation example: A value mapped to a single table that yields multiple rows	300
Advantage of annotated XML schema decomposition	236	Decomposition annotation example: A value mapped to multiple tables	302
Decomposing XML documents with annotated XML schemas	236	Decomposition annotation example: Grouping multiple values mapped to a single table	303
Registering and enabling XML schemas for decomposition	237	Decomposition annotation example: Multiple values from different contexts mapped to a single table	305
Disabling of annotated XML schema decomposition	239	Decomposition XML schema to SQL type compatibility	307
xdbDecompXML stored procedures for annotated schema decomposition.	240	XML schema to SQL types compatibility for annotated schema decomposition.	307
DECOMPOSE XML DOCUMENT	242	XML schema to SQL type compatibility for decomposition - Character types	308
Annotations	244	XML schema to SQL type compatibility for decomposition - Datetime types	310
XML decomposition annotations - Overview	244	XML schema to SQL type compatibility for decomposition - Graphic types	311
XML decomposition annotations - Specification and scope	244	XML schema to SQL type compatibility for decomposition - LOB and for bit data types	313
XML decomposition annotations - Summary	246	XML schema to SQL type compatibility for decomposition - Numeric types	315
db2-xdb:defaultSQLSchema decomposition annotation	247	Annotated XML schema decomposition - Limits and restrictions.	317
db2-xdb:rowSet decomposition annotation.	249	Troubleshooting considerations for annotated XML schema decomposition	319
db2-xdb:table decomposition annotation	254	Schema for XML decomposition annotations	322
db2-xdb:column decomposition annotation	257	Chapter 11. Sample applications	325
db2-xdb:locationPath decomposition annotation	259	Native XML Samples	325
db2-xdb:expression decomposition annotation	263	Native XML - Administration samples	326
db2-xdb:condition decomposition annotation	266	Native XML - Application Development samples	328
db2-xdb:contentHandling decomposition annotation	269	Appendix A. Encoding mappings	333
db2-xdb:normalization decomposition annotation	274	Mappings of encoding names to effective CCSIDs for stored XML data	333
db2-xdb:truncate decomposition annotation	278	Mappings of CCSIDs to encoding names for serialized XML output data.	344
db2-xdb:rowSetMapping decomposition annotation	280	Appendix B. DB2 Database technical information	349
Keywords for annotated XML schema decomposition	284	Overview of the DB2 technical information	349
Decomposition results	285	Documentation feedback	349
How decomposition results are formed in annotated XML schema decomposition	285	DB2 technical library in hardcopy or PDF format	350
Effect of validation on XML decomposition results.	286	Ordering printed DB2 books	352
Treatment of CDATA sections in annotated XML schema decomposition	287	Displaying SQL state help from the command line processor.	353
NULL values and empty strings in annotated XML schema decomposition	287	Accessing different versions of the DB2 Information Center	354
Recommendations	289		
Annotations of derived complex types for annotated XML schema decomposition	289		
Checklist for annotated XML schema decomposition	291		
XML schema structuring recommendations for decomposition	292		
Mapping examples	294		

Displaying topics in your preferred language in the DB2 Information Center	354
Updating the DB2 Information Center installed on your computer or intranet server	355
DB2 tutorials	357
DB2 troubleshooting information	357
Terms and Conditions	358

Appendix C. Notices	359
Trademarks	361
Index	363
Contacting IBM	369

Chapter 1. Introduction

Native XML data store overview

In contrast to XML Extender, the native XML data store enables well-formed XML documents to be stored in their hierarchical form within columns of a table. XML columns are defined with the XML data type. By storing XML data in XML columns, the data is kept in its native hierarchical form, rather than stored as text or mapped to a different data model.

Because the native XML data store is fully integrated into the DB2[®] database system, the stored XML data can be accessed and managed by leveraging DB2 functionality.

The storage of XML data in its native hierarchical form enables efficient search and retrieval of XML. XQuery, SQL, or a combination of both can be used to query XML data. SQL functions that return XML data or take XML arguments (referred to as SQL/XML functions) also enable XML data to be constructed or published from values retrieved from the database.

Querying:

XQuery

XQuery is a generalized language for querying XML data. The DB2 database system allows XQuery to be invoked directly or from within SQL. Because the XML data is stored in DB2 tables and views, functions are provided that extract the XML data from specified tables and views by naming the table or view directly, or by specifying an SQL query. XQuery supports various expressions for processing XML data and for constructing new XML objects such as elements and attributes. The programming interface to XQuery provides facilities similar to those of SQL to execute queries and retrieve results.

SQL statements and SQL/XML functions

Many SQL statements support the new 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. The set of SQL/XML functions, expressions, and specifications supported by the DB2 database system has been enhanced to take full advantage of the new XML data type.

XQuery can be invoked from within an SQL query. In this case, the SQL query can pass data to XQuery in the form of bound variables.

Application development:

Programming language support

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

- C or C++ (embedded SQL or DB2 CLI)

- COBOL
- Java™ (JDBC or SQLJ)
- C# and Visual Basic (DB2 .NET Data Provider)
- PHP

SQL and external procedures

XML data can be passed to SQL procedures and external procedures by including parameters of data type XML in CREATE PROCEDURE parameter signatures. Existing procedure features support the implementation of procedural logic flow around SQL statements that produce or make use of XML values as well as the temporary storage of XML data values in variables.

Administration:

XML schema repository (XSR)

The XML schema repository (XSR) is a repository for all XML artifacts required to validate and process XML instance documents stored in XML columns. It stores XML schemas, DTDs, and external entities referenced in XML documents.

Import and export utilities

The import and export utilities have been updated to support the native XML data type. These utilities treat XML data like LOB data: both types of data are stored outside the actual table. Application development support for importing and exporting XML data is also provided by updated db2Import and db2Export APIs. These updated utilities permit data movement of XML documents stored in XML columns that is similar to the data movement support for relational data.

Explain and Visual Explain

The Explain facility and the Visual Explain GUI tool have been updated to support SQL enhancements for querying XML data and to support XQuery expressions. These updates to the Explain facility and to the Visual Explain GUI tool allow you to see quickly how DB2 evaluates query statements against XML data.

Performance:

Indexes over XML data

Indexing support is available for data stored in XML columns. The use of indexes over XML data can improve the efficiency of queries issued against XML documents. Similar to a relational index, an index over XML data indexes a column. They differ, however, in that a relational index indexes an entire column, while an index over XML data indexes part of a column. You indicate which parts of an XML column are indexed by specifying an XML pattern, which is a limited XPath expression.

Optimizer

The optimizer has been updated to support the evaluation of SQL, XQuery, and SQL/XML functions that embed XQuery, against XML and relational data. The optimizer exploits statistics gathered over XML data, as well as data from indexes over XML data, to produce efficient query execution plans.

Tooling:

Support for the XML data type is available in tools including the Control Center, command line processor, Developer Workbench, and DB2 Development Add-In for Microsoft® Visual Studio .NET.

Annotated XML schema decomposition:

While the native XML data store enables you to store and access XML data as XML, in its hierarchical form, there can be cases where accessing XML data as relational data is required. Annotated XML schema decomposition is a new feature that decomposes documents based on annotations specified in an XML schema.

Related concepts:

- “XQuery” in *IBM DB2 XQuery Reference*
- “Annotated XML schema decomposition” on page 235
- “Visual Explain overview” in *Administration Guide: Implementation*
- “XML data movement overview” on page 155
- “Comparison of the XML model and the relational model” on page 7
- “Application programming language support for XML” on page 165
- “Tutorial for native XML data store” on page 18
- “Introduction to querying XML data with SQL” on page 48
- “Tools that support XML” on page 90
- “XML data type” on page 17
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133
- “Explain facility” in *Performance Guide*
- “XML and XQuery support in SQL procedures” on page 189
- “XML data type support in external routines” on page 194

Related tasks:

- “Migrating from XML Extender” in *Migration Guide*

Related reference:

- Chapter 9, “Restrictions on native XML data store,” on page 233
- “Native XML data store roadmap” in *DB2 Information Center*
- “Supported functions and administrative SQL routines and views” in *SQL Reference, Volume 1*

XML input and output overview

The DB2 database system, which manages both relational and XML data, offers various methods for the input and output of XML.

Background:

XML documents are stored in columns defined with the XML data type. Each row of an XML column stores a single well-formed XML document. The stored document is kept in its hierarchical form, preserving the XML data model. The document is not stored as text or mapped to a different data model.

XML columns can be defined in tables that contain columns of other types, which hold relational data, and multiple XML columns can be defined for a single table.

Input:

Figure 1 shows the various ways that XML data can be put into the database system.

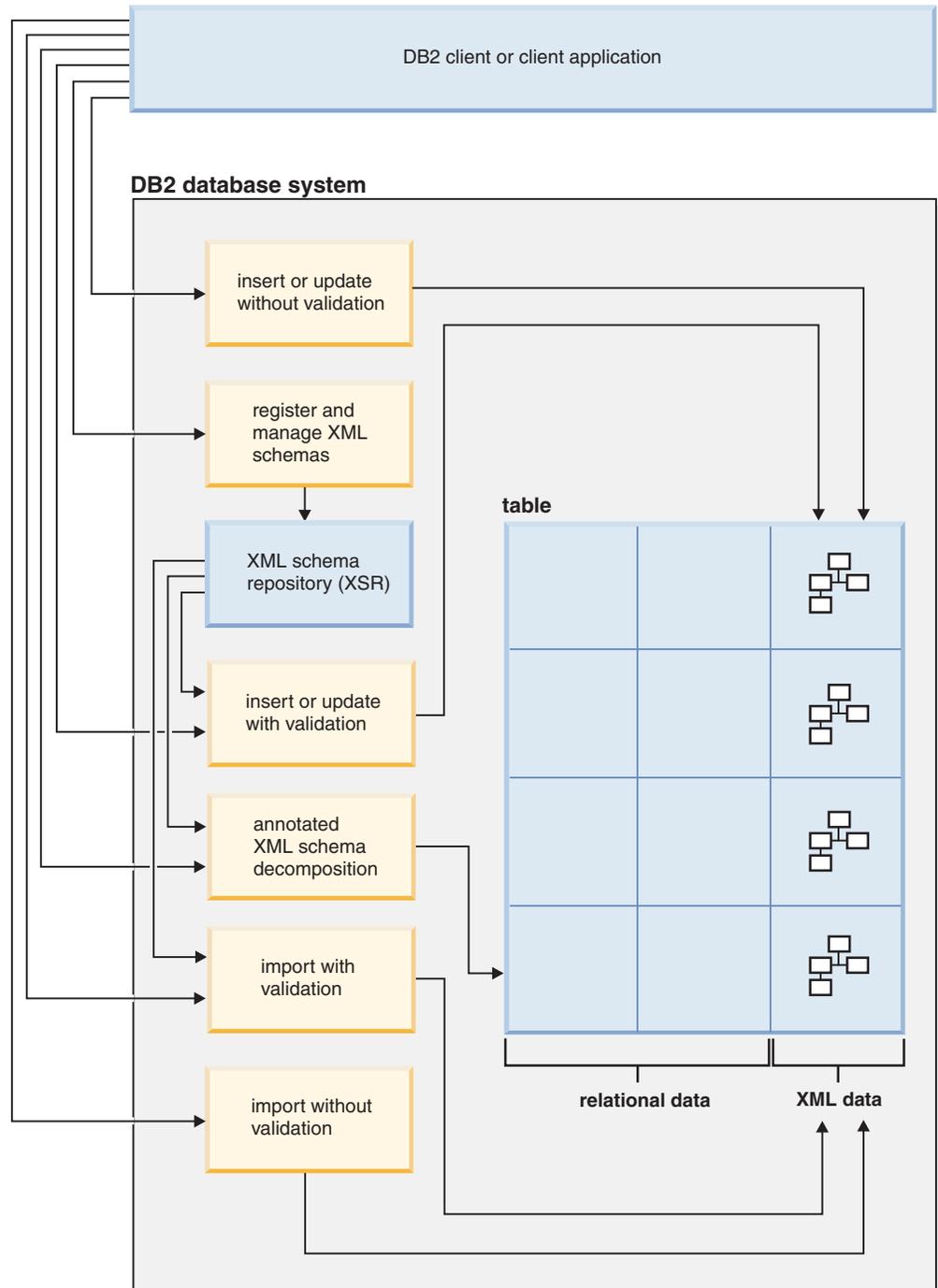


Figure 1. Methods for input of XML data

Insert or update

XML documents are inserted into XML columns using the INSERT SQL statement, and documents are updated using the UPDATE SQL statement. Validation of the XML documents during an insert or update operation is

optional. If validation is performed, the XML schema must first be registered with the XML schema repository (XSR).

Annotated XML schema decomposition

Data from XML documents can be decomposed or stored into relational and XML columns using the annotated XML schema decomposition feature. This feature stores data into columns according to annotations that are added to XML schema documents. These annotations map the data in XML documents to columns of tables.

XML schema documents referenced by the decomposition feature are stored in the XML schema repository (XSR).

Import

XML documents can be imported into XML columns using the import utility. Validation of the XML documents being imported is optional. If validation is performed, the XML schema against which the documents are validated, must first be registered with the XML schema repository (XSR).

XML schema repository (XSR)

The XML schema repository (XSR) stores XML schemas that are used for the validation or decomposition of XML documents. The XML schemas are registered with the XSR using stored procedures or commands.

Output:

Figure 2 on page 6 shows the various ways that XML data can be retrieved from the database system.

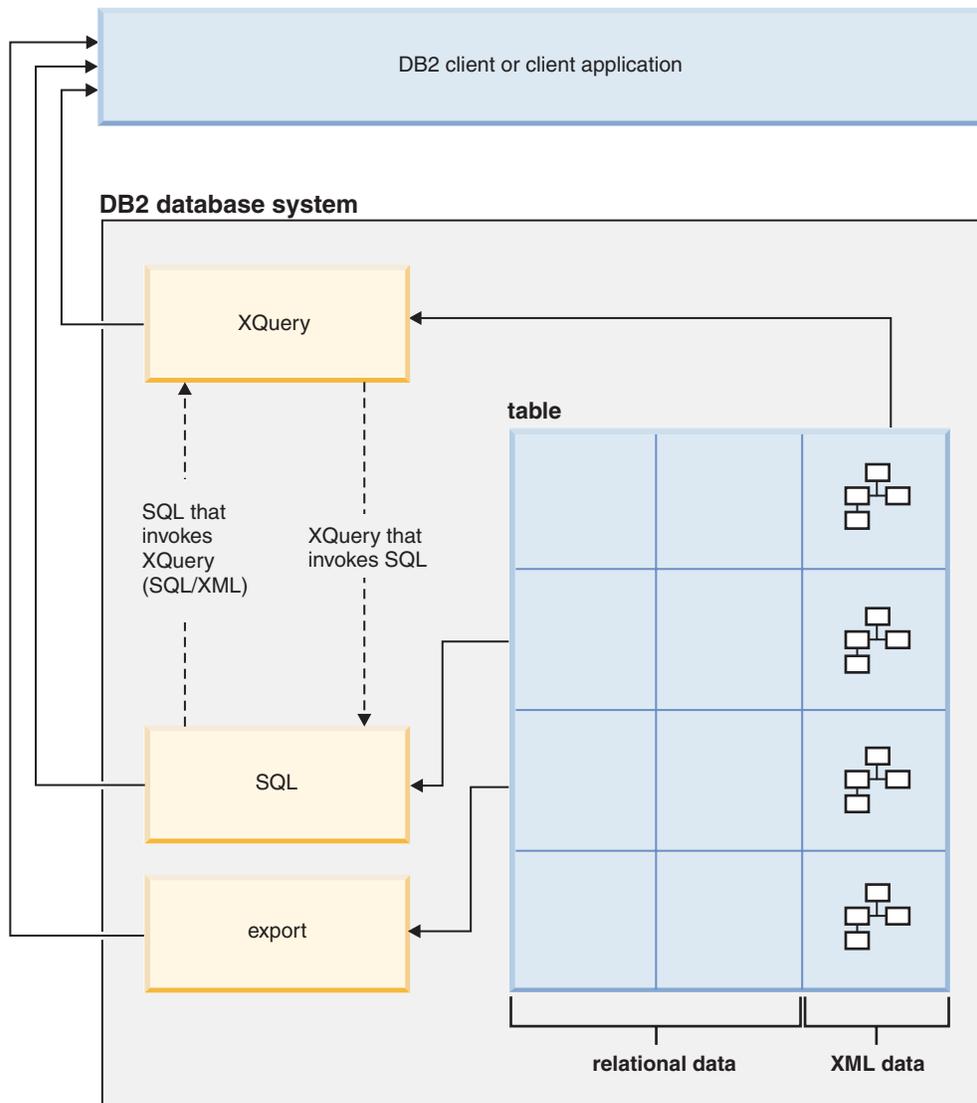


Figure 2. Methods for output of XML data

XQuery

XQuery is a language that enables you to query within XML documents. It addresses specific requirements of querying XML data that is highly variable in structure, unlike queries on relational data that expect a predictable structure.

XQuery can be invoked on its own, or it can invoke SQL to query XML stored in a DB2 database, through the `db2-fn:xmlcolumn` and `db2-fn:sqlquery` XQuery functions. `db2-fn:xmlcolumn` retrieves an entire XML column, whereas `db2-fn:sqlquery` retrieves XML values that are based on an SQL fullselect.

SQL

When querying XML data using an SQL fullselect, the query occurs at the column level. For this reason, only entire XML documents can be returned from the query; it is not possible to return fragments of an XML document using only SQL. To query within XML documents, XQuery must be used. XQuery can be invoked from SQL using the `XMLQUERY` or `XMLTABLE` SQL/XML functions, or the `XMLEXISTS` predicate. The `XMLQUERY` function returns the result of an XQuery expression as an XML sequence.

The XMLTABLE function returns the result of an XQuery expression as a table. The XMLEXISTS SQL predicate determines whether an XQuery expression returns a non-empty sequence.

Export XML documents can be exported from XML columns using the export utility. Exported XML data is stored separately from the exported relational data in the main data file. Details about each exported XML document is not stored directly in the main exported data file. The details are instead represented in the main data file by an XML data specifier (XDS).

Related concepts:

- “XML data specifier” on page 158
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133
- “Insertion into XML columns” on page 32
- “Registering XSR objects through stored procedures” on page 135
- “Registering XSR objects through the command line processor” on page 136
- “XQuery” in *IBM DB2 XQuery Reference*
- “Updates of XML columns” on page 34
- “XML data type” on page 17
- “XML validation” on page 41
- “XMLQUERY overview” on page 51
- “XMLTABLE overview” on page 61
- “Annotated XML schema decomposition” on page 235

Related reference:

- “Native XML data store roadmap” in *DB2 Information Center*
- “sqlquery function (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “xmlcolumn function (DB2 XQuery)” in *IBM DB2 XQuery Reference*

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. While this discussion explains some of the main differences between the models and the factors that apply to each, there are numerous factors that can determine the most suitable choice for your implementation. Use this discussion as a guideline, however, assess all factors that can impact your specific implementation.

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. If performance is more of an issue than flexibility, relational data might be the better choice.
- 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 the ratio of data complexity to volume is high
Many situations involve highly structured information in very small quantities. Representation of that data with a relational model can involve complex star schemas in which each dimension table is joined to many more dimension tables, and most of the tables have only a few rows. A better way to represent this data is to use a single table with an XML column, and to create views on that table, where each view represents a dimension.
- 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.

Related concepts:

- “Creation of tables with XML columns” on page 31
- “Native XML data store overview” on page 1
- “Tutorial for native XML data store” on page 18
- “Updates of XML columns” on page 34
- “XML data type” on page 17
- “Constraints” in *SQL Reference, Volume 1*
- “XQuery and XPath data model (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Related reference:

- Chapter 9, “Restrictions on native XML data store,” on page 233

XQuery and XPath data model

XQuery expressions operate on instances of the XQuery and XPath data model (XDM) and return instances of the data model. The XDM provides an abstract representation of one or more XML documents or fragments. The data model defines all permissible values of expressions in XQuery, including values that are used during intermediate calculations.

Parsing of XML data into the XDM and validating the data against a schema occur before data is processed by XQuery. During data model generation, the input XML document is parsed and converted into an instance of the XDM. The document can be parsed with or without validation.

The XDM is described in terms of sequences of atomic values and nodes.

Sequences and items

An instance of the XDM is 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 is a sequence:

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

Note: These examples use a notation to represent sequences that is consistent with the syntax that is used to construct sequences in XQuery. Each item in the sequence is separated by a comma. The entire sequence is enclosed in parentheses. A pair of empty parentheses represents an empty sequence. A single item that appears on its own is equivalent to a sequence that contains one item. For example, there is no distinction between the sequence (36) and the atomic value 36.

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

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.

Unlike nodes, 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.

The following examples are some of ways that atomic values are made:

- Extracted from nodes through a process called atomization. Atomization is used by expressions whenever a sequence of atomic values is required.
- Specified as a numeric or string literal. Literals are interpreted by XQuery as atomic values. For example, the following literals are interpreted as atomic values:
 - "this is a string" (type is xs:string)
 - 45 (type is xs:integer)
 - 1.44 (type is xs:decimal)
- Computed by constructor functions. For example, the following constructor function builds a value of type xs:date out of the string "2005-01-01":

```
xs:date("2005-01-01")
```
- Returned by the built-in functions `fn:true()` and `fn:false()`. These functions return the boolean values true and false. These values cannot be expressed as literals.
- Returned by many kinds of expressions, such as arithmetic expressions and logical expressions.

Node hierarchies

The nodes of a sequence form one or more *hierarchies*, or *trees*, that consist of a root node and all of the nodes that are reachable directly or indirectly from the root node. Every node belongs to exactly one hierarchy, and every hierarchy has exactly one root node. DB2 supports six node kinds: document, element, attribute, text, processing instruction, and comment.

The following XML document, `products.xml`, includes a root element, named `products`, which contains product elements. Each product element has an attribute

named pid (product ID) and a child element named description. The description element contains child elements named name and price.

```
<products>
<product xmlns="http://posample.org" pid="10">
<description>
<name>Fleece jacket</name>
<price>19.99</price>
</description>
</product>
<product xmlns="http://posample.org" pid="11">
<description>
<name>Nylon pants</name>
<price>9.99</price>
</description>
</product>
</products>
```

Figure 3 on page 12 shows a simplified representation of the data model for products.xml. The figure includes a document node (D), element nodes (E), attribute nodes (A), and text nodes (T).

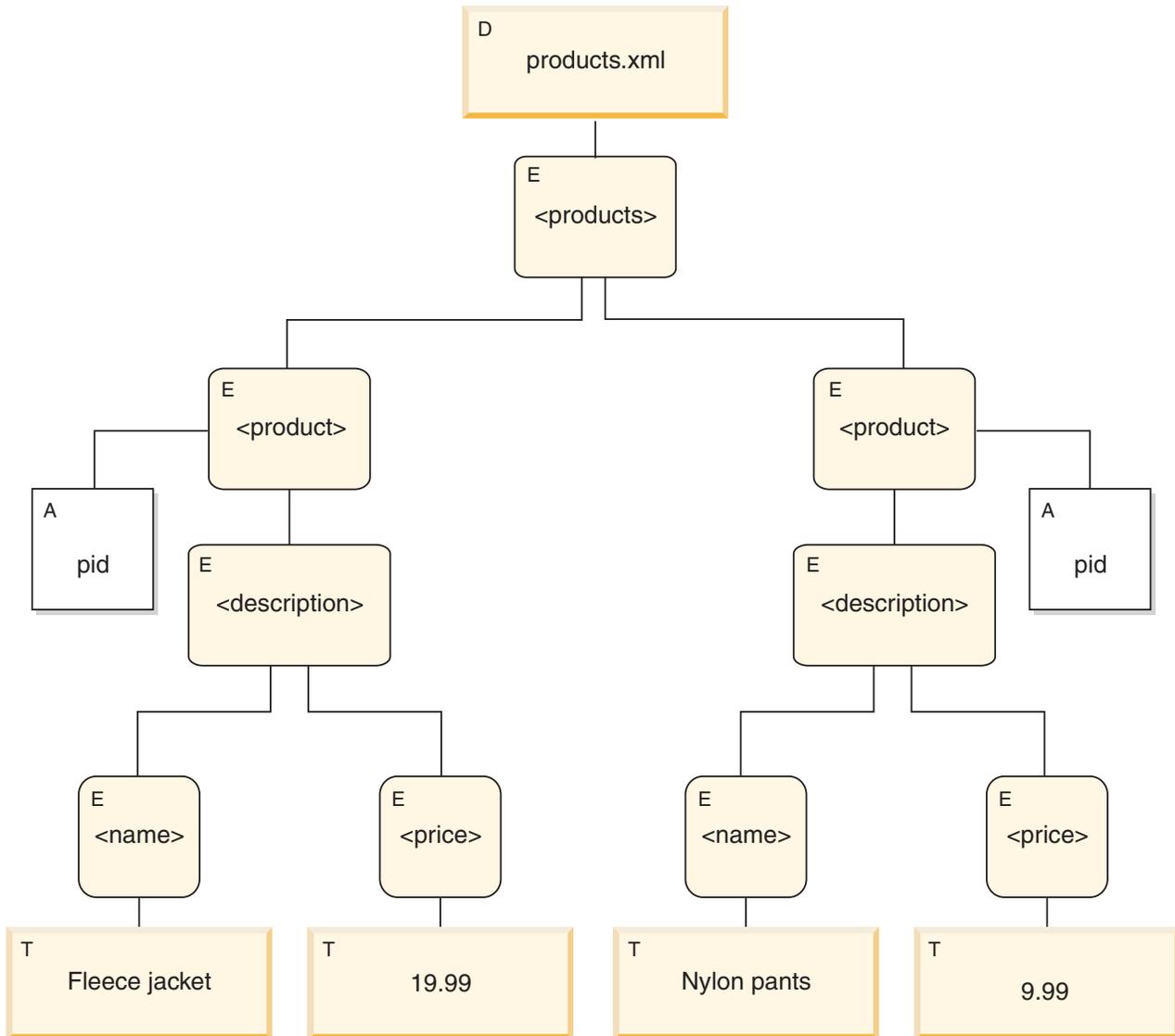


Figure 3. Data model diagram for products.xml document

As the example illustrates, a node can have other nodes as children, thus forming one or more *node hierarchies*. In the example, the element product is a child of products. The element description is a child of product. The elements name and price are children of the element description. The text node with the value Fleece Jacket is a child of the element name, and the text node 19.99 is a child of the element price.

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 (also known as the *type annotation*).
- **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.
- **in-scope namespaces.** The in-scope namespaces that are associated with the node.
- **content.** The content of the node.

Node kinds

DB2 supports six node kinds: document, element, attribute, text, processing instruction, and comment.

Document nodes

A document node encapsulates an XML document.

A document node can have zero or more children. The children can include element nodes, processing instruction nodes, comment nodes, and text nodes.

The string value of a document node is equal to the concatenated contents of all its descendant text nodes in document order. The type of the string value is `xs:string`. The typed value of a document node is the same as its string value, except that the type of the typed value is `xdt:untypedAtomic`.

A document node has the following node properties:

- children, possibly empty
- string-value
- typed-value

Document nodes can be constructed in XQuery expressions by using computed constructors. A sequence of document nodes can also be returned by the `db2-fn:xmlcolumn` function.

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, possibly empty
- type-name
- children, possibly empty
- attributes, possibly empty
- string-value
- typed-value

- in-scope-namespaces

Element nodes can be constructed in XQuery expressions by using direct or computed constructors.

The type-name property of an element node indicates the relationship between its typed value and its string value. For example, if an element node has the type-name property `xs:decimal` and the string value `"47.5"`, the typed value is the decimal value 47.5. If the type-name property of an element node is `xdt:untyped`, the element's typed value is equal to its string value and has the type `xdt:untypedAtomic`.

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, possibly empty
- type-name
- string-value
- typed-value

Attribute nodes can be constructed in XQuery expressions by using direct or computed constructors.

The type-name property of an attribute node indicates the relationship between its typed value and its string value. For example, if an attribute node has the type-name property `xs:decimal` and the string value `"47.5"`, its typed value is the decimal value 47.5.

Text nodes

A text node encapsulates XML character content.

A text node can have zero or one parent. Text nodes that are children of a document or element node never appear as adjacent siblings. When a document or element node is constructed, any adjacent text node 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, possibly empty
- parent, possibly empty

Text nodes can be constructed in XQuery expressions by computed constructors, or by the action of a direct element constructor.

Processing instruction nodes

A processing instruction node encapsulates an XML processing instruction.

A processing instruction node can have zero or one parent. The content of a processing instruction cannot include the string `?>`. The target of a processing instruction must be an NCName. (The target is used to identify the application to which the instruction is directed.)

A processing instruction node has the following node properties:

- target
- content
- parent, possibly empty

Processing instruction nodes can be constructed in XQuery expressions by using direct or computed constructors.

Comment nodes

A comment node encapsulates an XML comment.

A comment node can have zero or one parent. The content of a comment node cannot include the string `--` (two hyphens) or contain the hyphen character (`-`) as the last character.

A comment node has the following node properties:

- content
- parent, possibly empty

Comment nodes can be constructed in XQuery expressions by using direct or computed constructors.

Document order of nodes

All of the nodes in a hierarchy conform to an order, called *document order*, in which each node appears before its children. Document order corresponds to the order in which the nodes would appear if the node hierarchy were represented in serialized XML.

The nodes appear in the following order:

- The root node is the first node.
- Element nodes occur before their children.
- Attribute nodes immediately follow the element node with which they are associated. The relative order of attribute nodes is arbitrary, but this order does not change during the processing of a query.
- 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 identity

Each node has a unique identity. Two nodes are distinguishable even though their names and values might be the same. In contrast, atomic values do not have an identity.

Node identity is not the same as an ID-type attribute. An element in an XML document can be given an ID-type attribute by the document author. A node identity, however, is automatically assigned to every node by the system but is not directly visible to users.

Node identity is used to process the following types of expressions:

- Node comparisons. Node identity is used by the **is** operator to determine if two nodes have the same identity.
- Path expressions. Node identity is used by path expressions to eliminate duplicate nodes.
- Sequence expressions. Node identity is used by the **union**, **intersect**, or **except** operators to eliminate duplicate nodes.

Typed values and string values of nodes

Each node has both a *typed value* and a *string value*. These two node properties are used in the definitions of certain XQuery operations (such as atomization) and functions (such as `fn:data`, `fn:string`, and `fn:deep-equal`).

Table 1. String values and typed values of nodes

Node kind	String value	Typed value
Document	An instance of the <code>xs:string</code> data type that is the concatenated contents of all its descendant text nodes, in document order.	An instance of the <code>xdt:untypedAtomic</code> data type that is the concatenated contents of all its descendant text nodes, in document order.
Element in a validated document	<ul style="list-style-type: none"> • If validation assigned to the element a simple data type (such as <code>xs:decimal</code>) or a type that has simple content (such as a "temperature" type whose content is <code>xs:decimal</code>), the string value is the string that expresses the value of the element in the original XML document. • If validation assigned to the element a type that permits it to have mixed content (both text and child elements), the string value is an instance of the <code>xs:string</code> data type that is the concatenated contents of all its text node descendants, in document order. • If validation assigned to the element a type that permits no content (neither text nor child elements), the string value of the element is an empty string. • If validation assigned to the element a type that permits it to contain only child elements (no text), the string value of the element consists of the concatenated string values of all its text node descendants, in document order. 	<ul style="list-style-type: none"> • If validation assigned to the element a simple data type (such as <code>xs:decimal</code>) or a type that has simple content (such as a "temperature" type whose content is <code>xs:decimal</code>), the typed value is the result of casting the string value to the simple type that is assigned by the validation process (for example, <code>xs:decimal</code>). • If validation assigned to the element a type that permits it to have mixed content (both text and child elements), the typed value is an instance of the <code>xdt:untypedAtomic</code> data type that is the concatenated contents of all its text node descendants, in document order. • If validation assigned to the element a type that permits no content (neither text nor child elements), the typed value is an empty sequence. • If validation assigned to the element a type that permits it to contain only child elements (no text), the element has no typed value, and an attempt to extract its typed value (for example, by the <code>fn:data</code> function) results in an error.
Element in an unvalidated document	An instance of the <code>xs:string</code> data type that is the concatenated contents of all its text node descendants, in document order.	An instance of the <code>xdt:untypedAtomic</code> data type that is the concatenated contents of all its text node descendants, in document order.

Table 1. String values and typed values of nodes (continued)

Node kind	String value	Typed value
Attribute in a validated document	An instance of the xs:string data type that represents the attribute value in the original XML document.	The result of casting the string value into the type that was assigned to the attribute during validation. For example, if an attribute is validated as having the type xs:decimal, its string value might be the string "74.8" and its typed value might be 74.8 as a decimal number.
Attribute in an unvalidated document	An instance of the xs:string data type that represents the attribute value in the original XML document.	An instance of the xdt:untypedAtomic data type that represents the attribute value in the original XML document.
Text	The content as an instance of the xs:string data type.	The content as an instance of the xdt:untypedAtomic data type.
Comment	The content as an instance of the xs:string data type.	The content as an instance of the xs:string data type.
Processing instruction	The content as an instance of the xs:string data type.	The content as an instance of the xs:string data type.

XML data type

The DB2 V9.1 database system introduces a new data type called XML. This data type is used to define columns of a table that store XML values, where all stored XML values must be well-formed XML documents. The introduction of this native XML data type provides the ability to store well-formed XML documents in the database along side other relational data. All XML data is stored in the database in the UTF-8 code set.

XML values are processed in an internal representation that is not a string and not directly comparable to string values. An XML value can be transformed into a serialized string value representing the XML document using the XMLSERIALIZE function or by binding the value to an application variable of an XML, string, or binary type. Similarly, a string value that represents an XML document can be transformed to an XML value using the XMLPARSE function or by binding an application string, binary, or XML application type to an XML value. In SQL data change statements (such as INSERT) involving XML columns, a string or binary value that represents an XML document is transformed into an XML value using an injected XMLPARSE function. An XML value can be implicitly parsed or serialized when exchanged with application string and binary data types.

There is no architectural limit on the size of an XML value in a database. However, note that serialized XML data that is exchanged with a DB2 database is limited to 2 GB.

XML documents can be inserted, updated and deleted using SQL data manipulation statements. Validation of an XML document against an XML schema, typically performed during insert or update, is supported by the XML schema repository (XSR). The DB2 database system also provides mechanisms for constructing and querying XML values, as well as exporting and importing XML data. An index over XML data can be defined on an XML column, providing

improved search performance of XML data. The XML data in table or view columns can be retrieved as serialized string data through various application interfaces.

Related concepts:

- “Views” in *SQL Reference, Volume 1*
- “Tutorial for native XML data store” on page 18
- “Updates of XML columns” on page 34
- “XML data querying” on page 45
- “XML parsing” on page 37
- “Application programming language support for XML” on page 165
- “Comparison of the XML model and the relational model” on page 7
- “Creation of tables with XML columns” on page 31
- “Deletion of XML data from tables” on page 36
- “Insertion into XML columns” on page 32
- “Native XML data store overview” on page 1
- “Publishing XML values with SQL/XML” on page 81
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133
- “XML serialization” on page 86
- “XML validation” on page 41
- “XML data indexing overview” on page 93

Related reference:

- “XMLSERIALIZE scalar function” in *SQL Reference, Volume 1*
- “XMLPARSE scalar function” in *SQL Reference, Volume 1*
- “Native XML data store roadmap” in *DB2 Information Center*
- Chapter 9, “Restrictions on native XML data store,” on page 233
- “Supported territory codes and code pages” in *Administration Guide: Planning*

Tutorial for native XML data store

The XML data type introduced with the native XML data store enables you to define table columns that store in each row, a single, well-formed XML document. This tutorial demonstrates how to set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

After completing this tutorial, you will be able to do the following tasks:

- “Creating a DB2 database and table that can store XML data” on page 19
- “Creating indexes over XML data” on page 20
- “Inserting XML documents into XML typed columns” on page 20
- “Updating XML documents stored in an XML column” on page 21
- “Deleting rows based on the content of XML documents” on page 22
- “Querying XML data” on page 22
- “Validating XML documents against XML schemas” on page 26

Several application programming languages support the XML data type.

Important: Do not perform these tasks if you will be using the Database Partitioning Feature available with DB2 Enterprise Server Edition.

Restriction: You can create tables with XML columns only in databases defined with the UTF-8 code set and a compatible territory code. Exercise 1 shows how to set up a DB2 database with a UTF-8 code set.

Preparation:

Invoke the DB2 Command Line Processor by issuing the `db2 -td~` command in a system command processor window. The `-td` option specifies the tilde (~) as the statement termination character, rather than the semicolon (;) character, which is used by default. Choosing a termination character other than the semicolon ensures that statements or queries that use namespace declarations are not misinterpreted. A namespace declaration is terminated by a semicolon. (Examples throughout this tutorial use the ~ termination character.) On Windows® operating systems, the `db2cmd` command initializes the DB2 command-line environment.

The examples in the exercises can be entered at or copied and pasted into the DB2 Command Line Processor, in interactive mode. You can also use the Command Editor to issue the commands and statements interactively.

Namespaces: If the XML documents stored in the database contain namespaces, then all queries and associated operations that specify a namespace (such as creating an index over XML data with the `CREATE INDEX` statement) need to declare the same namespace in order to yield expected results. This requirement is standard namespace behavior as specified by the XML specification.

Exercise 1: Creating a DB2 database and table that can store XML data:

This exercise shows how to create a database named `xm1tut` with a UTF-8 code set and a table that contains an XML column.

Tables with XML columns must be created in databases defined with the UTF-8 code set and a compatible territory code.

To create a database that can store XML data, issue the following command to create a database named `xm1tut` with a UTF-8 code set:

```
CREATE DATABASE xm1tut USING CODESET UTF-8 TERRITORY US~
```

By default, automatic storage is enabled when you create a database. Automatic storage can result in improved performance and easier management of XML data, because it yields a database-managed space (DMS) table space that can be expanded as needed.

Connect to the database:

```
CONNECT TO xm1tut~
```

Now you can create a table named `Customer` that contains an XML column:

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

Note that specifying a primary key is optional and not required in order to store or index XML.

You can also add one or more XML columns to existing tables in UTF-8 databases with the ALTER TABLE SQL statement.

“Back to top” on page 18

Exercise 2: Creating indexes over XML data:

Indexes over XML data can improve the performance of queries on XML columns. Similar to a relational index, an index over XML data indexes a column. They differ, however, in that a relational index indexes an entire column, while an index over XML data indexes part of a column. You indicate which parts of an XML column are indexed by specifying an XML pattern, which is a limited XPath expression. You also need to specify a data type that the indexed values will be stored as. Generally, the type you choose should be the same type that is used in queries. If a node fails to cast to a specified index type, no index entry will be created, and no error will be returned.

You can only index a single XML column; composite indexes are not supported. You can have multiple indexes on an XML column however.

Note that not all clauses of the CREATE INDEX statement will apply to indexes over XML data. Refer to the CREATE INDEX statement for details.

Like relational indexes, it is recommended that you index XML elements or attributes that are frequently used in predicates and cross-document joins.

Issue the following statement to create an index over XML data:

```
CREATE UNIQUE INDEX cust_cid_xmlidx ON Customer(Info)
GENERATE KEY USING XMLPATTERN
  'declare default element namespace "http://posample.org"; /customerinfo/@Cid'
AS SQL DOUBLE~
```

This indexes the values of the Cid attribute of <customerinfo> elements from the Info column of the Customer table.

Note that the XML pattern specified is case-sensitive. If, for example, the XML documents contained the attribute "cid" instead of "Cid", then those documents would not match this index.

“Back to top” on page 18

Exercise 3: Inserting XML documents into XML typed columns:

Well-formed XML documents are inserted into XML typed columns using the INSERT SQL statement.

Typically, XML documents are inserted using application programs. While XML data can be inserted through applications using XML, binary, or character types, it is recommended that you use XML or binary types to avoid code page conversion issues.

This exercise shows how to insert XML documents into XML typed columns manually in the Command Line Processor, where the XML document is always a character literal. In most cases, string data cannot be directly assigned to a target with an XML data type; the data must first be parsed explicitly using the XMLPARSE function. In INSERT, UPDATE, or DELETE operations, however, string

data can be directly assigned to XML columns, without an explicit call to the XMLPARSE function. In these three cases, the string data is implicitly parsed. Refer to the XML parsing documentation for more information.

Insert three XML documents into the Customer table that you created in Exercise 1:

```
INSERT INTO Customer (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 Customer (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 Customer (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 as follows:

```
SELECT * from Customer~
```

“Back to top” on page 18

Exercise 4: Updating XML documents stored in an XML column:

To update an XML document stored in an XML column, you must perform a full-document update using the UPDATE SQL statement.

Update one of the documents inserted in Exercise 3 as follows (where the values of the <street>, <city>, and <pcode-zip> elements have changed):

```
UPDATE customer SET info =  
'<customerinfo xmlns="http://posample.org" Cid="1002">  
  <name>Jim Noodle</name>  
  <addr country="Canada">  
    <street>1150 Maple Drive</street>  
    <city>Newtown</city>  
    <prov-state>Ontario</prov-state>  
    <pcode-zip>Z9Z 2P2</pcode-zip>  
  </addr>  
  <phone type="work">905-555-7258</phone>  
</customerinfo>')~
```

```

</customerinfo>'
WHERE XMLEXISTS (
  'declare default element namespace "http://posample.org";
  $doc/customerinfo[@Cid = 1002]'
  passing INFO as "doc")~

```

The XMLEXISTS predicate ensures that only the document containing the attribute Cid="1002" is replaced. Notice how the predicate expression in XMLEXISTS, [@Cid = 1002], is not specified as a string comparison: [@Cid = "1002"]. This is because the index, created in Exercise 2, for the Cid attribute was defined with the DOUBLE data type. In order for the index to match this query, Cid cannot be specified as a string in the predicate expression.

You can confirm that the XML document was updated as follows:

```
SELECT * from Customer~
```

The record where Cid="1002" should contain the changed <street>, <city>, and <pcode-zip> values.

If XML documents can be identified by values in the non-XML columns of the same table, then you can also use SQL comparison predicates to identify rows for update. In this example, where the Cid value from the XML document is also stored in the CID column of the CUSTOMER table, an SQL comparison predicate on the CID column could have been used to identify the row. For demonstration purposes, the XMLEXISTS predicate was used in this example.

"Back to top" on page 18

Exercise 5: Deleting rows based on the content of XML documents:

XML documents are deleted using the DELETE SQL statement. The XMLEXISTS predicate can be used to identify particular documents to delete.

The following example deletes only those XML documents from the Info column that have a <customerinfo> element with an attribute Cid="1003":

```

DELETE FROM Customer
WHERE XMLEXISTS (
  'declare default element namespace "http://posample.org";
  $doc/customerinfo[@Cid = 1003]'
  passing INFO as "doc")~

```

If XML documents can be identified by values in the non-XML columns of the same table, then you can also use SQL comparison predicates to identify rows for deletion. In this example, where the Cid value from the XML document is also stored in the CID column of the CUSTOMER table, you could have performed the same operation with the following DELETE statement that applies an SQL comparison predicate on the Cid column, to identify the row: DELETE FROM customer WHERE Cid=1003. For demonstration purposes, the XMLEXISTS predicate was used in this example.

You can confirm that the XML document was deleted as follows:

```
SELECT * from Customer~
```

Two records are returned.

"Back to top" on page 18

Exercise 6: Querying XML data:

XML data can be queried using SQL (with the SELECT statement), XQuery (with XQuery expressions), or a combination of both.

When querying with SQL alone (without the use of any XQuery), you can only query at the column level. That is, you can return the entire XML document stored in the column, but you cannot query within the document or return fragments of a document. To query values within an XML document or return fragments of a document, you must use XQuery.

From within the contexts of both SQL and XQuery, you can invoke the other. In SQL, you can invoke XQuery using the XMLQUERY function. In XQuery, you can issue a fullselect using the db2-fn:sqlquery function.

Important: XQuery is case-sensitive, while SQL is case-insensitive. Names in XQuery, such as table and SQL schema names (which are both uppercase by default), must be carefully specified because of the language's case-sensitivity. This is particularly important when using XQuery with SQL. When invoking XQuery within SQL, be mindful that the XQuery expression remains case-sensitive, even though it is placed within the SQL context.

SQL:

Retrieving entire XML documents:

To retrieve, using only SQL, all of the XML documents stored in the column named Info and values from the Cid primary key column, issue the following SELECT statement:

```
SELECT Cid, Info FROM Customer~
```

This query returns the two stored XML documents.

Retrieving and filtering XML values:

The previous example showed how to query and return entire XML documents using only SQL. To query the actual values within the XML documents, you need to use XQuery. The XMLQUERY function enables you to invoke XQuery from the SQL context. The following example shows how to query within the XML documents in the Info column:

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

This SELECT statement returns the following constructed element:

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

In the XMLQUERY function, a default namespace is first specified. This namespace matches the namespace of the documents previously inserted. The for clause specifies iteration through the <customerinfo> elements in each document from the Info column. Notice that the Info column is specified using the passing clause,

which binds the Info column to the variable "doc" that is referenced in the for clause. The return clause then constructs an <out> element, which contains the <name> element from each iteration of the for clause.

The WHERE clause uses the XMLEXISTS predicate to consider only a subset of the documents in the Info column. This filtering yields only those documents that have a <city> element (along the path specified) with a value of "Toronto".

XQuery:

DB2 XQuery offers two built-in functions specifically for use with DB2 databases: db2-fn:sqlquery and db2-fn:xmlcolumn. db2-fn:sqlquery retrieves a sequence that is the result table of an SQL fullselect. db2-fn:xmlcolumn retrieves a sequence from an XML column.

A query that invokes XQuery directly must be prefixed with the case-insensitive "XQUERY" keyword.

The following examples show how to directly execute XQuery expressions that contain the db2-fn:xmlcolumn and db2-fn:sqlquery built-in functions.

Note: There are several options you can set to customize your Command Line Processor environment, particularly for the display of XQuery results. The -i option "pretty-prints" results from XQuery expressions to make them easier to read. You can set this option, if not already set, as follows:

```
UPDATE COMMAND OPTIONS USING i ON~
```

Retrieving entire XML documents:

To retrieve all of the XML documents previously inserted into the Info column, you can use XQuery alone or issue a fullselect from XQuery.

XQuery:

To retrieve all XML documents in the INFO column without the use of SQL, execute the following query:

```
XQUERY db2-fn:xmlcolumn ('CUSTOMER.INFO')~
```

Names in SQL statements are automatically converted to uppercase by default. When the Customer table was created using the CREATE TABLE SQL statement, therefore, the name of the table and columns were made uppercase. Because XQuery is case-sensitive, you must be careful to specify the table and column name in db2-fn:xmlcolumn with the correct case.

This query retrieves all XML documents stored in the Info column of the Customer table, and is equivalent to the SQL query SELECT Info FROM Customer.

Fullselect:

To retrieve all XML documents in the INFO column using a fullselect in the XQuery context, execute the following query:

```
XQUERY db2-fn:sqlquery ('SELECT Info FROM Customer')~
```

Notice how the Info and Customer names do not need to be specified in upper case. This is because the SELECT statement is processed in an SQL context, and is therefore case-insensitive.

This query is equivalent to the previous example, which used db2-fn:xmlcolumn. Both queries return all XML documents stored in the Info column of the Customer table.

Retrieving and filtering XML values:

Instead of retrieving entire XML documents, you can retrieve fragments of the document and filter on values present in the documents. This can be accomplished using XQuery alone or a fullselect in the XQuery context.

XQuery:

The following example shows how to use only XQuery to return elements containing <name> nodes for all documents in the Info column that have a <city> element (along the path specified) with a value of "Toronto".

```
XQUERY declare default element namespace "http://posample.org";
  for $d in db2-fn:xmlcolumn('CUSTOMER.INFO')/customerinfo
  where $d/addr/city="Toronto"
  return <out>{$d/name}</out>~
```

This query returns the following constructed element:

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

The db2-fn:xmlcolumn function retrieves a sequence from the INFO column of the CUSTOMER table. The for clause binds the variable \$d to each <customerinfo> element in the CUSTOMER.INFO column, and the where clause restricts the items to only those that have a <city> element (along the path specified) with a value of "Toronto". The return clause constructs the returned XML value, which is an element <out> that contains the <name> element for all documents that satisfy the condition specified in the where clause.

Fullselect:

The following example shows how to issue a fullselect within XQuery, using the db2-fn:sqlquery function.

```
XQUERY declare default element namespace "http://posample.org";
  for $d in db2-fn:sqlquery(
    'SELECT INFO
     FROM CUSTOMER
     WHERE Cid < 2000')/customerinfo
  where $d/addr/city="Toronto"
  return <out>{$d/name}</out>~
```

In this example, the set of XML documents being queried is first restricted, in the fullselect, by particular values in the non-XML Cid column. This demonstrates an advantage of db2-fn:sqlquery: it allows SQL predicates to be applied within XQuery. The documents that result from the SQL query are then further restricted in the where clause of the XQuery expression to those that have a <city> element (along the path specified) with a value of "Toronto".

This query yields the same results as in the previous example, which used db2-fn:xmlcolumn. Both queries return the constructed element:

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

“Back to top” on page 18

Exercise 7: Validating XML documents against XML schemas:

You can validate your XML documents against XML schemas only; DTD validation is not supported. (Although you cannot validate against DTDs, you can still insert documents that contain a DOCTYPE or that refer to DTDs.) There are tools available, such as those in IBM® Rational® Application Developer, that help you generate XML schemas from various sources, including DTDs, existing tables, or XML documents.

Before you can validate, you must register your XML schema with the built-in XML schema repository (XSR). This process involves registering each XML schema document that makes up the XML schema. Once all XML schema documents have been successfully registered, you must complete the registration. One method of registering an XML schema is through commands.

Register and complete registration of the posample.customer XML schema as follows, providing the absolute path to the sqllib/samples/xml directory on your system:

```
REGISTER XMLSCHEMA 'http://posample.org'
FROM 'file:///c:/sqllib/samples/xml>customer.xsd' AS posample.customer COMPLETE~
```

Because this XML schema consists of only one schema document, the registration and completion steps can be combined into a single command.

You can verify that the XML schema was successfully registered by querying the SYSCAT.XSROBJECTS catalog view, which contains information about objects stored in the XSR. This query and its result (formatted for clarity) are as follows:

```
SELECT OBJECTSCHEMA, OBJECTNAME FROM SYSCAT.XSROBJECTS~
```

OBJECTSCHEMA	OBJECTNAME
-----	-----
POSAMPLE	CUSTOMER

This XML schema is now available to be used for validation. Validation is typically performed during an INSERT or UPDATE operation. Perform validation using the XMLVALIDATE function. The INSERT or UPDATE operation on which XMLVALIDATE was specified, will only occur if the validation succeeds.

The following INSERT statement inserts a new XML document into the Info column of the Customer table, only if the document is valid according to the posample.customer XML schema previously registered.

```
INSERT INTO Customer(Cid, Info) VALUES (1003, XMLVALIDATE (XMLPARSE (DOCUMENT
'<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>' PRESERVE WHITESPACE )
ACCORDING TO XMLSCHEMA ID posample.customer ))~

```

XMLVALIDATE operates on XML data. Because the XML document in this example is passed as character data, and character data can only be assigned directly to XML in INSERT, UPDATE, or DELETE statements, the XMLPARSE function must be used in this case. The XMLPARSE function parses its argument as an XML document and returns an XML value.

To verify that the validation and insert were successful, query the Info column:

```
SELECT Info FROM Customer~
```

This query should return three XML documents, one of which is the document just inserted.

“Back to top” on page 18

Related concepts:

- “Command Editor overview” in *Online DB2 Information Center*
- “Database Partitioning Feature (DPF)” in *Administration Guide: Planning*
- “Automatic storage databases” in *Administration Guide: Implementation*
- “Addition of XML columns to existing tables” on page 32
- “Application programming language support for XML” on page 165
- “Creation of tables with XML columns” on page 31
- “Deletion of XML data from tables” on page 36
- “Insertion into XML columns” on page 32
- “Native XML data store overview” on page 1
- “Updates of XML columns” on page 34
- “XML data encoding” on page 217
- “XML data querying” on page 45
- “XML data type” on page 17
- “XML input and output overview” on page 3
- “XML parsing” on page 37
- “XML validation” on page 41
- “XMLQUERY overview” on page 51
- “XSR object registration” on page 134

Related reference:

- “CREATE DATABASE command” in *Command Reference*
- “db2 - Command line processor invocation command” in *Command Reference*
- “db2cmd - Open DB2 command window command” in *Command Reference*
- “sqlquery function (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “xmlcolumn function (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “SYSCAT.XSROBJECTS catalog view” in *SQL Reference, Volume 1*

- “XMLEXISTS predicate” in *SQL Reference, Volume 1*
- “XMLPARSE scalar function” in *SQL Reference, Volume 1*
- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “XMLVALIDATE scalar function” in *SQL Reference, Volume 1*
- “Supported territory codes and code pages” in *Administration Guide: Planning*
- Chapter 9, “Restrictions on native XML data store,” on page 233

XML information roadmap

An XML information roadmap, which offers additional XML resources, is available at: <http://www.ibm.com/developerworks/db2/roadmaps/db2-xml-roadmap.html>.

Chapter 2. XML storage

XML storage object overview

DB2 tables can store well formed XML documents in XML columns, alongside columns that contain relational data. In much the same way that LONG VARCHAR and LOB data are stored apart from the other contents of a table, DB2 stores XML data contained in table columns of the type XML in auxiliary XML storage objects. When stored in system managed space, the files associated with XML storage objects have the file type extension .xda.

XML storage objects are separate from, but dependent upon their parent table objects. For each XML value stored in a row of an XML table column, DB2 maintains a record, called an XML data specifier (XDS), which specifies where to retrieve the XML data stored on disk from the associated XML storage object.

You can store XML documents of up to 2 gigabytes in size in a database. Because XML data can be quite large, you may want to monitor the buffering activity for XML data separately from the buffering activity for other data. A number of monitor elements are available to help you gauge the buffer pool activity for XML storage objects.

Related concepts:

- “Preference of database managed table spaces for native XML data store performance” on page 131
- “Guidelines for storage requirements for XML documents” on page 29
- “Native XML data store overview” on page 1
- “XML data specifier” on page 158

Related reference:

- “Buffer pool activity monitor elements” in *System Monitor Guide and Reference*

Guidelines for storage requirements for XML documents

The amount of space that an XML document occupies in a DB2 database is determined by the initial size of the document in raw form and by a number of other properties. The following list includes the most important of these properties:

Document structure

XML documents that contain complex markup tagging require a larger amount of storage space than documents with simple markup. For example, an XML document that has many nested elements, each containing a small amount of text or having short attribute values, occupies more storage space than an XML document composed primarily of textual content.

Node names

The length of element names, attribute names, namespace prefixes and similar, non-content data also affect storage size. Any information unit of this type that exceeds 4 bytes in raw form is compressed for storage, resulting in comparatively greater storage efficiency for longer node names.

Ratio of attributes to elements

Typically, the more attributes that are used per element, the lower the amount of storage space that is required for the XML document.

Document codepage

XML documents with encoding that uses more than one byte per character occupy a larger amount storage space than documents using a single-byte character set.

Document validation

XML documents are annotated after having been validated against an XML schema. The addition of type information after validation results in an increased storage requirement.

Related concepts:

- “XML storage object overview” on page 29

Chapter 3. Working with 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. Databases in which you create tables with XML columns must be Unicode databases that exist in instances with only a single database partition defined.

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, serialized XML data that is exchanged with a DB2 database is limited to 2 GB, so the effective limit of an XML column is 2 GB.

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

Example: The sample database contains a table for 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)
```

Example: The VALIDATED predicate checks whether the value in the specified XML column has been validated. You can define a table check constraint on XML columns, using the VALIDATED predicate, to ensure that all documents inserted or updated in a table are valid.

```
CREATE TABLE TableValid (id BIGINT,  
                          xmlcol XML,  
                          CONSTRAINT valid_check CHECK (xmlcol IS VALIDATED))
```

Related concepts:

- “Partitioned databases” in *Administration Guide: Planning*
- “XML storage object overview” on page 29
- “Addition of XML columns to existing tables” on page 32
- “Deletion of XML data from tables” on page 36
- “Insertion into XML columns” on page 32
- “Native XML data store overview” on page 1
- “XML data type” on page 17
- “XML serialization” on page 86
- “Updates of XML columns” on page 34
- “Native XML Samples” on page 325

Related tasks:

- “Creating a Unicode database” in *Administration Guide: Planning*

Related reference:

- “VALIDATED predicate” in *SQL Reference, Volume 1*
- Chapter 9, “Restrictions on native XML data store,” on page 233

- “CREATE TABLE statement” in *SQL Reference, Volume 2*

Addition of XML columns to existing tables

To add XML columns to existing tables, you specify columns with the XML data type in the ALTER TABLE statement with the ADD clause. You can add XML columns only to tables that do not have type-1 indexes defined on them. (Note that type-1 indexes are deprecated indexes; new indexes since DB2 Universal Database Version 8.1 are created as type-2 indexes.) A table can have one or more XML columns. Tables to which you add XML columns must be in Unicode databases that exist in instances with only a single database partition defined.

Example The sample database contains a table for 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;
```

Related concepts:

- “Creation of tables with XML columns” on page 31
- “Insertion into XML columns” on page 32
- “Native XML data store overview” on page 1
- “Native XML Samples” on page 325
- “Partitioned databases” in *Administration Guide: Planning*
- “Updates of XML columns” on page 34
- “XML data type” on page 17

Related tasks:

- “Creating a Unicode database” in *Administration Guide: Planning*

Related reference:

- Chapter 9, “Restrictions on native XML data store,” on page 233
- “ALTER TABLE statement” in *SQL Reference, Volume 2*
- “CREATE TABLE statement” in *SQL Reference, Volume 2*
- “Conversion of type-1 indexes in migrated databases” in *Migration Guide*

Insertion into XML columns

To insert data into an XML column, use the SQL INSERT statement. The input to the XML column must be a well-formed XML document, as defined in the XML 1.0 specification. The application data type can be an XML, character, or binary type.

It is recommended that XML data be inserted 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 is in its serialized string format. When you insert the data into an XML column, it must be converted to its XML hierarchical format. If the application data type is an XML data type, the DB2 database server performs this operation implicitly. If the application data type is not an XML type, you can invoke the XMLPARSE function explicitly when you perform the insert operation, to convert the data from its serialized string format to the XML hierarchical format.

During document insertion, you might also want to validate the XML document against a registered XML schema. You can do that with the XMLVALIDATE function.

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 is in file c6.xml, and 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 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: In a static embedded C application, insert data from a binary XML 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);
```

Related concepts:

- “Application programming language support for XML” on page 165
- “Deletion of XML data from tables” on page 36
- “Encoding considerations for input of XML data to a database” on page 219
- “Native XML data store overview” on page 1
- “Updates of XML columns” on page 34
- “XML data encoding” on page 217

- “XML data type” on page 17
- “XML input and output overview” on page 3
- “XML parsing” on page 37
- “XML validation” on page 41

Related reference:

- “INSERT statement” in *SQL Reference, Volume 2*
- “CURRENT IMPLICIT XMLPARSE OPTION special register” in *SQL Reference, Volume 1*
- “XMLPARSE scalar function” in *SQL Reference, Volume 1*
- “XMLVALIDATE scalar function” in *SQL Reference, Volume 1*
- Chapter 9, “Restrictions on native XML data store,” on page 233

Updates of XML columns

To update data in an XML column, use the SQL UPDATE statement. Include a WHERE clause when you want to update specific rows. The entire column value will be replaced. The input to the XML column must be a well-formed XML document, as defined in the XML 1.0 specification. The application data type can be an XML, character, or binary type.

XML data in an application is in its serialized string format. When you update an XML column, the data must be converted to its XML hierarchical format. If the application data type is an XML data type, you must let the DB2 database server perform this operation implicitly. If the application data type is not an XML type, you can invoke the XMLPARSE function explicitly when you perform the update operation, to convert the data from its serialized string format 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 with the 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>
</customerinfo>
```

```

    </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 MyCustomer SET Info=? WHERE Cid=?";
updateStmt = conn.prepareStatement(sqls);
updateStmt.setInt(1, cid);
File file = new File("c7.xml");
updateStmt.setBinaryStream(2, new FileInputStream(file), (int)file.length());
updateStmt.executeUpdate();

```

Example: In an embedded C application, update data in an XML column from a binary XML 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";
                    $doc/customerinfo[@Cid = $c]'
                    passing INFO as "doc", cast(:cid as integer) as "c");

```

Related concepts:

- “Application programming language support for XML” on page 165
- “Insertion into XML columns” on page 32
- “Native XML data store overview” on page 1
- “XML data encoding” on page 217
- “XML data type” on page 17
- “XML input and output overview” on page 3
- “XML parsing” on page 37
- “XML validation” on page 41

Related reference:

- “`XMLEXISTS` predicate” in *SQL Reference, Volume 1*
- “`XMLPARSE` scalar function” in *SQL Reference, Volume 1*
- “`XMLVALIDATE` scalar function” in *SQL Reference, Volume 1*

- Chapter 9, “Restrictions on native XML data store,” on page 233

Deletion of XML data from tables

To delete rows that contain XML documents, use the DELETE SQL statement. 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 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 deleted if the XQuery expression returns a non-empty sequence.

An XML column must either be NULL or contain a well-formed XML document. To delete a value from an XML column without deleting the row, use the UPDATE SQL statement with SET NULL, to set the column to NULL, if the column is defined as nullable.

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. This statement deletes the row that has a customer ID of 1002.

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

Example: Delete the XML document in the row of MyCustomer for which the value of the city element is Markham, but leave the row. This statement should delete the XML data from the Info column for the row that has a customer ID of 1002.

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

Related concepts:

- “Insertion into XML columns” on page 32
- “Native XML data store overview” on page 1
- “Updates of XML columns” on page 34
- “XML data type” on page 17
- “XML input and output overview” on page 3

Related reference:

- “XMLEXISTS predicate” in *SQL Reference, Volume 1*
- “DELETE statement” in *SQL Reference, Volume 2*
- “UPDATE statement” in *SQL Reference, Volume 2*

- Chapter 9, “Restrictions on native XML data store,” on page 233

XML support in triggers

Use the CREATE TRIGGER SQL statement to create BEFORE UPDATE or AFTER UPDATE triggers on XML columns, or to create INSERT or DELETE triggers on tables that include XML columns. The trigger body cannot reference transition variables of type XML.

Example: Create an AFTER UPDATE trigger on the XML column Info of table MyCustomer, which is a copy of the sample Customer table.

```
CREATE TRIGGER UPDAFTR
  AFTER UPDATE OF Info
  ON MyCustomer
  REFERENCING NEW AS N
  FOR EACH ROW
  BEGIN ATOMIC
    INSERT INTO CustLog VALUES(N.CID, CURRENT TIMESTAMP, 'Update');
  END
```

Example: Create an INSERT trigger on the MyCustomer table.

```
CREATE TRIGGER INSAFTR
  AFTER INSERT ON Customer
  REFERENCING NEW AS N
  FOR EACH ROW
  BEGIN ATOMIC
    INSERT INTO CustLog VALUES(N.CID, CURRENT TIMESTAMP, 'Insert');
  END
```

Related concepts:

- “Native XML data store overview” on page 1
- “Triggers” in *SQL Reference, Volume 1*
- “XML data type” on page 17

Related reference:

- “CREATE TRIGGER statement” in *SQL Reference, Volume 2*
- Chapter 9, “Restrictions on native XML data store,” on page 233

XML parsing

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

You can let the DB2 database manager perform parsing implicitly, or you can 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.
You must use implicit parsing in this case.
- 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 SQL compiler 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 use assign the result to an XML column or use it as a stored procedure parameter of type XML.

The XMLPARSE function takes a non-XML, character or binary data type as input. For embedded dynamic SQL applications, you need to cast the parameter marker that represents the input document for XMLPARSE to the appropriate data type. For example:

```
INSERT INTO MyCustomer (Cid, Info)
VALUES (?, xmlparse(document cast(? as clob(1k)) preserve whitespace))
```

For static embedded SQL applications, a host variable argument of the XMLPARSE function cannot be declared as an XML type (XML AS BLOB, XML AS CLOB, or XML AS DBCLOB type).

XML parsing and whitespace handling:

During implicit or 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.

With implicit XML parsing:

- If the input data type is not an XML type or is not cast to an XML data type, the DB2 database manager always strips whitespace.
- If the input data type is an XML data type, you can use the CURRENT IMPLICIT XMLPARSE OPTION special register to control preservation of boundary whitespace. You can set this special register to STRIP WHITESPACE or PRESERVE WHITESPACE. The default is stripping of boundary whitespace.

If you use XML validation, the DB2 database manager ignores the CURRENT IMPLICIT XMLPARSE OPTION special register and uses only the validation rules to determine stripping or preservation of whitespace in the following cases:

```
xmlvalidate(? ACCORDING TO XMLSCHEMA ID schemaname)
xmlvalidate(?)
xmlvalidate(:hvxml ACCORDING TO XMLSCHEMA ID schemaname)
xmlvalidate(:hvxml)
xmlvalidate(cast(? as xml) ACCORDING TO XMLSCHEMA ID schemaname)
xmlvalidate(cast(? as xml))
```

In these cases, ? represents XML data, and :hvxml is an XML host variable.

See XML validation for information on how XML validation influences whitespace handling.

The XML standard specifies an xml:space attribute that controls the stripping or preservation of whitespace within XML data. xml:space attributes override any whitespace settings for implicit or explicit XML parsing.

For example, in the following document, the spaces immediately before and after 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 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) or references an external 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 and external DTDs
- Expands entity references and parameter entities

Example: 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 the carriage returns, 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: 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 need to 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: 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
  Product ( 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>

```

Related concepts:

- “Differences in an XML document after storage and retrieval” on page 88
- “Native XML data store overview” on page 1
- “XML data in JDBC applications” on page 177
- “XML data type” on page 17
- “XML validation” on page 41
- “Native XML Samples” on page 325

Related tasks:

- “Declaring XML host variables in embedded SQL applications” on page 171

Related reference:

- “CURRENT IMPLICIT XMLPARSE OPTION special register” in *SQL Reference, Volume 1*
- Chapter 9, “Restrictions on native XML data store,” on page 233
- “XMLPARSE scalar function” in *SQL Reference, Volume 1*
- “Native XML data store roadmap” in *DB2 Information Center*
- “SET CURRENT IMPLICIT XMLPARSE OPTION statement” in *SQL Reference, Volume 2*

XML validation

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

In addition, XML validation adds type annotations to element nodes, attribute nodes and atomic values, and strips off *ignorable whitespace* in the XML document. Validation is optional, but highly recommended.

You can use the XMLVALIDATE function to validate an XML document. Usually, you invoke XMLVALIDATE when you insert or update an XML document in a DB2 database. However, you can also invoke XMLVALIDATE on an XML document that is not in a database.

Before you can invoke 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.

XML validation and ignorable whitespace:

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:sequence>
  </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 elements that contains a non-string type.

In the XMLVALIDATE function, you can explicitly specify the XML schema document to use for validation. If you do not specify an XML schema document, the DB2 database server looks in the input document for an xsi:schemaLocation or xsi:noNamespaceSchemaLocation attribute that identifies the XML schema document. xsi:schemaLocation or xsi:noNamespaceSchemaLocation attributes are defined by the XML Schema specification, and are called *XML schema hints*. An xsi:schemaLocation attribute contains one or more pairs of values that help to locate the XML schema document. The first value in each pair is a namespace, and the second value is a hint that indicates where to find the XML schema for the namespace. An xsi:noNamespaceSchemaLocation value contains only a hint. If an XML schema document is specified in the XMLVALIDATE function, it overrides the xsi:schemaLocation or xsi:noNamespaceSchemaLocation attribute.

The following examples assume that schema product is registered in the XML schema repository (XSR). You might use CLP statements like this to complete the registration:

```
REGISTER XMLSCHEMA http://posample.org/product.xsd FROM product.xsd \
AS myschema.product
COMPLETE XMLSCHEMA myschema.product
```

Alternatively, because the XML schema consists of a single schema document, you can use a single statement to register the XML schema and complete registration:

```
REGISTER XMLSCHEMA http://posample.org/product.xsd FROM product.xsd \
AS myschema.product COMPLETE
```

Example: Suppose that you create table MyProduct like this:

```
CREATE TABLE MyProduct LIKE Product
```

You want to insert the following document into XML column Info in the MyProduct table using a dynamic SQL application, and you want to 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>
```

In your INSERT statement, the XMLVALIDATE function specifies the XML schema to use for validation:

```
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',
XMLVALIDATE(? ACCORDING TO XMLSCHEMA ID myschema.product))
```

When you retrieve the stored data, you can see where XMLVALIDATE 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 XMLVALIDATE removes it.

If you need to ensure that you insert only documents that are validated into an XML column or retrieve only documents that are validated from an XML column, use the VALIDATED predicate.

To verify that all inserted documents are validated, you create a check constraint that contains the VALIDATED predicate on the XML column. To retrieve only validated documents from an XML column, you use the VALIDATED predicate in a WHERE clause.

Example: Suppose that you want to retrieve only validated XML documents from the Info column of the MyCustomer table. Execute SELECT statements like this one:

```
SELECT Info FROM MyCustomer WHERE Info IS VALIDATED
```

Example: Suppose that you want to insert only validated XML documents into the Info column of the MyCustomer table. You can define a check constraint to enforce this condition. Alter the MyCustomer table in the following way:

```
ALTER TABLE MyCustomer ADD CONSTRAINT CK_VALIDATED CHECK (Info IS VALIDATED)
```

Issuing this statement, however, makes the use of the VALIDATED predicate in the previous example unnecessary, as only valid documents would be successfully inserted or updated in the table.

Example: Suppose that you want to validate the following document with the customer schema, but you do not want to store it in a database.

```
<customerinfo xml:space="default"
  xmlns="http://posample.org"
  Cid='1011'>
  <name>Kathy Smith</name>
  <addr country='Canada'>
  <street>25 Rosewood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>M6W 1E6</pcode-zip>
  </addr>
  <phone type='work'>416-555-6676</phone>
</customerinfo>
```

Assume that you have assigned the document to an application variable. You can use a VALUES statement like this to do the validation:

```
VALUES XMLVALIDATE(? according to xmlschema id myschema.customer)
```

This document is valid according to the XML schema, so the VALUES statement returns a result table that contains the document. If the document is not valid, VALUES returns an SQL error.

Related concepts:

- “Native XML Samples” on page 325
- “Native XML data store overview” on page 1
- “XML data type” on page 17
- “XML parsing” on page 37
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133

Related reference:

- “COMPLETE XMLSCHEMA ” on page 150
- “INSERT statement” in *SQL Reference, Volume 2*
- “REGISTER XMLSCHEMA ” on page 147
- “VALIDATED predicate” in *SQL Reference, Volume 1*
- “VALUES statement” in *SQL Reference, Volume 2*
- “XMLVALIDATE scalar function” in *SQL Reference, Volume 1*
- “Native XML data store roadmap” in *DB2 Information Center*
- Chapter 9, “Restrictions on native XML data store,” on page 233

Querying and retrieving XML

XML data querying

You can query or retrieve XML data stored in the database in several ways:

- XQuery
- XQuery that invokes SQL
- SQL only
- SQL that executes XQuery expressions

These various methods allow you to query or retrieve XML and other relational data from either an SQL or XQuery context.

Pieces of or entire XML documents can be queried and retrieved using these methods. Queries can return fragments or entire XML documents, and results returned from queries can be limited by using predicates. Because queries on XML data return XML sequences, a query's result can be used in the construction of XML data as well.

Related concepts:

- "Native XML data store overview" on page 1
- "Comparison of methods for querying XML data" on page 49
- "Data retrieval with XQuery" on page 70
- "Full-text search in XML documents" on page 79
- "Introduction to querying XML data with SQL" on page 48
- "Publishing XML values with SQL/XML" on page 81
- "XML data type" on page 17
- "XML input and output overview" on page 3

Related reference:

- "Native XML data store roadmap" in *DB2 Information Center*
- Chapter 9, "Restrictions on native XML data store," on page 233

Introduction to XQuery

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

Unlike relational data, which is predictable and has a regular structure, XML data is highly variable. XML data is often unpredictable, sparse, and self-describing.

Because the structure of XML data is unpredictable, the queries that you need to perform on XML data often differ from typical relational queries. The XQuery language provides the flexibility required to perform these kinds of operations. For example, you might need to create XML queries that perform the following operations:

- Search XML data for objects that are at unknown levels of the hierarchy.
- Perform structural transformations on the data (for example, you might want to invert a hierarchy).
- Return results that have mixed types.

In XQuery, expressions are the main building blocks of a query. Expressions can be nested and form the body of a query. A query can also have a prolog before this body. The *prolog* contains a series of declarations that define the processing environment for the query. The *query body* consists of an expression that defines the result of the query. This expression can be composed of multiple XQuery expressions that are combined using operators or keywords.

Figure 4 illustrates the structure of a typical query. In this example, the prolog contains two declarations: a version declaration, which specifies the version of the XQuery syntax to use to process the query, and a default namespace declaration that specifies the namespace URI to use for unprefixed element and type names. The query body contains an expression that constructs a `price_list` element. The content of the `price_list` element is a list of product elements that are sorted in descending order by price.



Figure 4. Structure of a typical query in XQuery

Retrieving DB2 data with XQuery functions

In XQuery, a query can call one of the following functions to obtain input XML data from a DB2 database: `db2-fn:sqlquery` and `db2-fn:xmlcolumn`.

The function `db2-fn:xmlcolumn` retrieves an entire XML column, whereas `db2-fn:sqlquery` retrieves XML values that are based on an SQL fullselect.

db2-fn:xmlcolumn

The `db2-fn:xmlcolumn` function takes a string literal argument that identifies an XML column in a table or a view and returns a sequence of XML values that are in that column. The argument of this function is case sensitive. The string literal argument must be a qualified column name of type XML. This function allows you to extract a whole column of XML data without applying a search condition.

In the following example, the query uses the `db2-fn:xmlcolumn` function to get all of the purchase orders in the `PURCHASE_ORDER` column of the `BUSINESS.ORDERS` table. The query then operates on this input data to extract the cities from the shipping address in these purchase orders. The result of the query is a list of all cities to which orders are shipped:

```
db2-fn:xmlcolumn('BUSINESS.ORDERS.PURCHASE_ORDER')/shipping_address/city
```

db2-fn:sqlquery

The `db2-fn:sqlquery` function takes a string argument that represents a fullselect and returns an XML sequence that is a concatenation of the XML values that are returned by the fullselect. The fullselect must specify a single-column result set, and the column must have a data type of XML. Specifying a fullselect allows you to use the power of SQL to present XML data to XQuery.

In the following example, a table called `BUSINESS.ORDERS` contains an XML column called `PURCHASE_ORDER`. The query in the example uses the `db2-fn:sqlquery` function to call SQL to get all of the purchase orders where the ship date is June 15, 2005. The query then operates on this input data to extract the cities from the shipping addresses in these purchase orders. The result of the query is a list of all of the cities to which orders are shipped on June 15:

```
db2-fn:sqlquery("
SELECT purchase_order FROM business.orders
WHERE ship_date = '2005-06-15' ")/shipping_address/city
```

Important: An XML sequence that is returned by the `db2-fn:sqlquery` or `db2-fn:xmlcolumn` function can contain any XML values, including atomic values and nodes. These functions do not always return a sequence of well-formed documents. For example, the function might return a single atomic value, like 36, as an instance of the XML data type.

SQL and XQuery have different conventions for case-sensitivity of names. You should be aware of these differences when using the `db2-fn:sqlquery` and `db2-fn:xmlcolumn` functions.

SQL is not a case-sensitive language

By default, all ordinary identifiers, which are used in SQL statements, are automatically converted to uppercase. Therefore, the names of SQL tables and columns are customarily uppercase names, such as `BUSINESS.ORDERS` and `PURCHASE_ORDER` in the previous examples. In an SQL statement, these columns can be referenced by using lowercase names, such as `business.orders` and `purchase_order`, which are automatically converted to uppercase during processing of the SQL statement. (You can also create a case-sensitive name that is called a *delimited identifier* in SQL by enclosing the name in double quotation marks.)

XQuery is a case-sensitive language

XQuery does not convert lowercase names to uppercase. This difference can lead to some confusion when XQuery and SQL are used together. The string that is passed to `db2-fn:sqlquery` is interpreted as an SQL query and is parsed by the SQL parser, which converts all names to uppercase. Thus, in the `db2-fn:sqlquery` example, the table name `business.orders` and the column names `purchase_order` and `ship_date` can appear in either uppercase or lowercase. The operand of `db2-fn:xmlcolumn`, however, is not an SQL query. The operand is a case-sensitive XQuery string literal that represents the name of a column. Because the actual name of the column is `BUSINESS.ORDERS.PURCHASE_ORDER`, this name must be specified in uppercase in the operand of `db2-fn:xmlcolumn`.

Introduction to querying XML data with SQL

XML data can be queried using an SQL fullselect or with the SQL/XML query functions of XMLQUERY and XMLTABLE. The XMLEXISTS predicate can also be used in SQL queries on XML data.

When querying XML data using only SQL, without any XQuery, you can only query at the column level by issuing a fullselect. For this reason, only entire XML documents can be returned from the query; it is not possible to return fragments of a document using only SQL.

To query within XML documents, you need to use XQuery. XQuery can be invoked from SQL using any of the following SQL/XML functions or predicate:

XMLQUERY

An SQL scalar function that returns the result of an XQuery expression as an XML sequence.

XMLTABLE

An SQL table function that returns the result of an XQuery expression as a table.

XMLEXISTS

An SQL predicate that determines whether an XQuery expression returns a non-empty sequence.

Related concepts:

- “Expressions (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Sequences and items (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Full-text search in XML documents” on page 79
- “Native XML data store overview” on page 1
- “XML data querying” on page 45
- “XML data type” on page 17
- “XMLEXISTS predicate when querying XML data” on page 66
- “XMLQUERY overview” on page 51
- “XMLTABLE overview” on page 61

Related reference:

- “XMLEXISTS predicate” in *SQL Reference, Volume 1*
- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “XMLTABLE table function” in *SQL Reference, Volume 1*
- “Native XML data store roadmap” in *DB2 Information Center*
- Chapter 9, “Restrictions on native XML data store,” on page 233

Comparison of XQuery to SQL

DB2 supports storing well-formed XML data in a column of a table and retrieving the XML data from the database by using SQL, XQuery, or a combination of SQL and XQuery. Both languages are supported as primary query languages, and both languages provide functions for invoking the other language.

XQuery

A query that invokes XQuery directly begins with the keyword XQUERY. This keyword indicates that XQuery is being used and that the DB2 server must therefore use case sensitivity rules that apply to the XQuery

language. Error handling is based on the interfaces that are used to process XQuery expressions. XQuery errors are reported with an SQLCODE and SQLSTATE in the same way that SQL error errors are reported. No warnings are returned from processing XQuery expressions. XQuery obtains data by calling functions that extract XML data from DB2 tables and views. XQuery can also be invoked from an SQL query. In this case, the SQL query can pass XML data to XQuery in the form of bound variables. XQuery supports various expressions for processing XML data and for constructing new XML objects such as elements and attributes. The programming interface to XQuery provides facilities similar to those of SQL to prepare queries and retrieve query results.

SQL SQL provides capabilities to define and instantiate values of the XML data type. Strings that contain well-formed XML documents can be parsed into XML values, optionally validated against an XML schema, and inserted or updated in tables. Alternatively, XML values can be constructed by using SQL constructor functions, which convert other relational data into XML values. Functions are also provided to query XML data by using XQuery and to convert XML data into a relational table for use within an SQL query. Data can be cast between SQL and XML data types in addition to serializing XML values into string data.

SQL/XML provides the following functions and predicates for calling XQuery from SQL:

XMLQUERY

XMLQUERY is a scalar function that takes an XQuery expression as an argument and returns an XML sequence. The function includes optional parameters that can be used to pass SQL values to the XQuery expression as XQuery variables. The XML values that are returned by XMLQUERY can be further processed within the context of the SQL query.

XMLTABLE

XMLTABLE is a table function that uses XQuery expressions to generate an SQL table from XML data, which can be further processed by SQL.

XMLEXISTS

XMLEXISTS is an SQL predicate that determines if an XQuery expression returns a sequence of one or more items (and not an empty sequence).

Comparison of methods for querying XML data

Because XML data can be queried in a number of ways, using XQuery, SQL, or a combination of these, the method to choose can differ depending on your situation. The following sections describe conditions that are advantageous for a particular query method.

XQuery only:

Querying with XQuery alone can be a suitable choice when:

- applications access only XML data, without the need to query non-XML relational data
- migrating queries previously written in XQuery to DB2 Database for Linux, UNIX, and Windows

- returning query results to be used as values for constructing XML documents
- the query author is more familiar with XQuery than SQL

XQuery that invokes SQL:

Querying with XQuery that invokes SQL can be a suitable choice when (in addition to the scenarios identified in the previous section on using XQuery only):

- queries involve XML data and relational data; SQL predicates and indexes defined on the relational columns can be leveraged in the query
- you want to apply XQuery expressions to the results of:
 - UDF calls, as these cannot be invoked directly from XQuery
 - XML values constructed from relational data using SQL/XML publishing functions
 - queries that use DB2 Net Search Extender V9.1, where DB2 Net Search Extender V9.1 is a separately installable product that offers full text search, but which must be used with SQL

SQL only:

When retrieving XML data using only SQL, without any XQuery, you can only query at the XML column level. For this reason, only entire XML documents can be returned from the query. This usage is suitable for when you want to retrieve entire XML documents, and when you do not need to query based on values within the stored documents, where the predicates of your query are on other non-XML columns of the table.

SQL/XML functions that execute XQuery expressions:

The SQL/XML functions XMLQUERY and XMLTABLE, as well as the XMLEXISTS predicate, enable XQuery expressions to be executed from within the SQL context. Executing XQuery within SQL can be a suitable choice when:

- existing SQL applications need to be enabled for querying within XML documents. To query within XML documents, XQuery expressions need to be executed, which can be done using SQL/XML
- applications querying XML data need to pass parameter markers to the XQuery expression. (The parameter markers are first bound to XQuery variables in XMLQUERY or XMLTABLE.)
- the query author is more familiar with SQL than XQuery
- both relational and XML data needs to be returned in a single query
- you need to join XML and relational data
- you want to group or aggregate XML data. You can apply the GROUP BY or ORDER BY clauses of a subselect to the XML data (for example, after the XML data has been retrieved and collected in table format by using the XMLTABLE function)

Related concepts:

- “Comparison of XQuery to SQL” on page 48
- “Full-text search in XML documents” on page 79
- “Introduction to querying XML data with SQL” on page 48
- “Introduction to XQuery” on page 45
- “Publishing XML values with SQL/XML” on page 81

- “Retrieving DB2 data with XQuery functions” on page 46
- “XQuery” in *IBM DB2 XQuery Reference*

Related reference:

- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “XMLEXISTS predicate” in *SQL Reference, Volume 1*
- “XMLTABLE table function” in *SQL Reference, Volume 1*

XMLQUERY scalar function

XMLQUERY overview

XMLQUERY is an SQL scalar function that enables you to 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 contain one or more items.

By executing XQuery expressions from within the SQL context, you can:

- operate on parts of stored XML documents, instead of entire XML documents (only XQuery can query within an XML document; SQL alone queries at the whole document level)
- enable XML data to participate in SQL queries
- operate on both relational and XML data
- apply further SQL processing to the returned XML values (for example, ordering results with the ORDER BY clause of a subselect)

Refer to the documentation on the comparison of querying methods for more details.

Note that XQuery is case-sensitive, so the XQuery expressions and variables specified in XMLQUERY must be carefully specified.

Related concepts:

- “Native XML Samples” on page 325
- “Sequences and items (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Casting of XMLQUERY results to non-XML types” on page 55
- “Comparison of methods for querying XML data” on page 49
- “Constant and parameter marker passing to XMLEXISTS and XMLQUERY” on page 69
- “Empty sequences returned by XMLQUERY” on page 53
- “Introduction to querying XML data with SQL” on page 48
- “Native XML data store overview” on page 1
- “Non-empty sequences returned by XMLQUERY” on page 52
- “XML data type” on page 17

Related reference:

- Chapter 9, “Restrictions on native XML data store,” on page 233
- “Subselect” in *SQL Reference, Volume 1*
- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “Native XML data store roadmap” in *DB2 Information Center*

Non-empty sequences returned by XMLQUERY

If evaluation of the XQuery expression specified in XMLQUERY results in a non-empty sequence, then a non-empty sequence is returned as the result of XMLQUERY.

For example, if the following two XML documents are stored in the XML column INFO of the CUSTOMER table,

```
<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>
```

and this query was issued:

```
SELECT XMLQUERY ('declare default element namespace "http://posample.org";
                 $d/customerinfo/phone' passing INFO as "d")
FROM CUSTOMER
```

the table that results contains two rows as follows (the table has been formatted for clarity):

Table 2. Result table

<phone xmlns="http://posample.org" type="work">905-555-7258</phone>
<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>

Notice that the first row contains a sequence of one <phone> element, while the second row has a sequence of four <phone> elements. This result occurs because the second XML document contains four <phone> elements, and XMLQUERY returns a sequence of all elements that satisfy the XQuery expression. (Note that the result in the second row is not a well-formed document. Ensure that any application receiving this result can properly handle this behavior.)

The previous example shows how XMLQUERY is commonly used: applied to one XML document at a time, where each row in the resulting table represents the result from one document. XMLQUERY, however, can also be applied to multiple documents at once, as is the case when multiple documents are contained in a

single sequence. In this case, the results from applying XMLQUERY to all documents in the sequence are returned in a single row.

For example, assume that the same documents presented above are stored in the INFO column of the CUSTOMER table. The db2-fn:xmlcolumn function in the following query returns one sequence that contains the two XML documents in the INFO column.

```
VALUES (XMLQUERY ('declare default element namespace "http://posample.org";
                 db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo/phone'))
```

XMLQUERY is then applied to this single sequence of XML documents, and the resulting table contains only one row, as follows:

Table 3. Result table

<pre><phone xmlns="http://posample.org" type="work">905-555-7258</phone><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></pre>

All <phone> elements from the XML documents in the INFO column are returned in a single row, because XMLQUERY operates on a single value: the sequence of XML documents returned from db2-fn:xmlcolumn.

Related concepts:

- “Expressions (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Sequences and items (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Casting of XMLQUERY results to non-XML types” on page 55
- “Constant and parameter marker passing to XMLEXISTS and XMLQUERY” on page 69
- “Empty sequences returned by XMLQUERY” on page 53
- “XML data type” on page 17
- “XMLQUERY overview” on page 51

Related reference:

- “xmlcolumn function (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “Native XML - Application Development samples” on page 328

Empty sequences returned by XMLQUERY

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

For example, in the following query, XMLQUERY will return 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";
                    $d//addr[city="Aurora"]' passing INFO as "d") AS ADDRESS
FROM CUSTOMER
```

Assume that there are three rows of the CUSTOMER table, but only one XML document that contains a <city> element with the value of “Aurora”. The following table would result from the previous SELECT statement (the output has been formatted for clarity).

Table 4. Result table

CID	ADDRESS
1001	
1002	
1003	<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>

Notice how empty sequences of zero-length serialized XML, rather than NULL values, are returned for rows that do not have a <city> element with the value of "Aurora". The <addr> element is returned in the third row, however, because it satisfies the XQuery expression. In the third row, a non-empty sequence is returned.

You can avoid returning rows that contain empty sequences by applying a predicate, such as XMLEXISTS, in the WHERE clause of your statement, rather than in the SELECT clause. For example, the previous query can be rewritten as follows, moving the filtering predicate from the XMLQUERY function, to the WHERE clause:

```
SELECT Cid, 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="Aurora"]' passing c.INFO as "d")
```

The table that results from this query is as follows:

Table 5. Result table

CID	ADDRESS
1003	<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>

XMLQUERY is commonly used in a SELECT clause to return fragments of selected documents. Predicates specified in the XQuery expression of XMLQUERY do not filter rows from the result set, they are only used to determine what fragments are returned. To actually eliminate rows from your result set, you need to apply a predicate in the WHERE clause. The XMLEXISTS predicate can be used to apply predicates that depend on values within stored XML documents.

Related concepts:

- "Expressions (DB2 XQuery)" in *IBM DB2 XQuery Reference*
- "Sequences and items (DB2 XQuery)" in *IBM DB2 XQuery Reference*
- "Casting of XMLQUERY results to non-XML types" on page 55
- "Constant and parameter marker passing to XMLEXISTS and XMLQUERY" on page 69
- "Non-empty sequences returned by XMLQUERY" on page 52
- "XML data type" on page 17
- "XMLEXISTS predicate when querying XML data" on page 66
- "XMLQUERY overview" on page 51

Related reference:

- “XMLEXISTS predicate” in *SQL Reference, Volume 1*
- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “Native XML - Application Development samples” on page 328

Casting of XMLQUERY results to non-XML types

If you want to return the results of XMLQUERY to the SQL context for further processing, such as for comparison or ordering operations, you need to cast the XML value that is returned to a compatible SQL type. The XMLCAST specification enables you to cast between XML and non-XML values.

Note that you can only cast the result of XMLQUERY to an SQL data type when the XQuery expression specified in XMLQUERY returns a sequence that contains one item that has been atomized.

Example: Comparing XML values to non-XML values in a query:

In the following query, the sequence returned by XMLQUERY is cast from its XML type to a character type, so that it can be compared with the NAME column of the PRODUCT table. (If the XML value that results from XMLQUERY is not a serialized string, then the XMLCAST operation could fail.)

```
SELECT R.Pid
FROM PURCHASEORDER P, PRODUCT R
WHERE R.NAME =
      XMLCAST( XMLQUERY ('declare default element namespace "http://posample.org";
                        $d/PurchaseOrder/itemlist/item/product/name'
                        PASSING P.PORDER AS "d") AS VARCHAR(128))
```

Example: Ordering by XMLQUERY results:

In the following query, product IDs are returned in an order sorted by the value of the <name> element of the product’s description, which is stored as an XML document. Because SQL cannot sort on XML values, the sequence must be cast to a value that SQL can order on, in this case, character.

```
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))
```

Related concepts:

- “Atomization (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Expressions (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Sequences and items (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “XMLQUERY overview” on page 51

Related reference:

- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “Casting between data types” on page 56
- “XMLCAST specifications” in *SQL Reference, Volume 1*

Casting between data types

There are many occasions where a value with a given data type needs to be *cast* to a different data type or to the same data type with a different length, precision, or scale. Data type promotion is one example where the promotion of one data type to another data type requires that the value be cast to the new data type. A data type that can be cast to another data type is *castable* from the source data type to the target data type.

The casting of one data type to another can occur implicitly or explicitly. The cast functions, CAST specification, or XMLCAST specification can be used to explicitly change a data type, depending on the data types involved. The database manager might implicitly cast data types during assignments that involve a distinct type. In addition, when a sourced user-defined function is created, the data types of the parameters of the source function must be castable to the data types of the function that is being created.

The supported casts between built-in data types are shown in Table 6 on page 57. The first column represents the data type of the cast operand (source data type), and the data types across the top represent the target data type of the cast operation. A 'Y' indicates that the CAST specification can be used for the combination of source and target data types. Cases in which only the XMLCAST specification can be used are noted.

In a Unicode database, if a truncation occurs when a character or graphic string is cast to another data type, a warning returns if any nonblank characters are truncated. This truncation behavior is unlike the assignment of character or graphic strings to a target when an error occurs if any nonblank characters are truncated.

The following casts involving distinct types are supported (using the CAST specification unless noted otherwise):

- Cast from distinct type *DT* to its source data type *S*
- Cast from the source data type *S* of distinct type *DT* to distinct type *DT*
- Cast from distinct type *DT* to the same distinct type *DT*
- Cast from a data type *A* to distinct type *DT* where *A* is promotable to the source data type *S* of distinct type *DT*
- Cast from an INTEGER to distinct type *DT* with a source data type SMALLINT
- Cast from a DOUBLE to distinct type *DT* with a source data type REAL
- Cast from a VARCHAR to distinct type *DT* with a source data type CHAR
- Cast from a VARGRAPHIC to distinct type *DT* with a source data type GRAPHIC
- For a Unicode database, cast from a VARCHAR or a VARGRAPHIC to distinct type *DT* with a source data type CHAR or GRAPHIC
- Cast from a distinct type *DT* with a source data type *S* to XML using the XMLCAST specification
- Cast from an XML to a distinct type *DT* with a source data type of any built-in data type, using the XMLCAST specification depending on the XML schema data type of the XML value

FOR BIT DATA character types cannot be cast to CLOB.

It is not possible to cast a structured type value to something else. A structured type *ST* should not need to be cast to one of its supertypes, because all methods on

the supertypes of *ST* are applicable to *ST*. If the desired operation is only applicable to a subtype of *ST*, use the subtype-treatment expression to treat *ST* as one of its subtypes.

When a user-defined data type involved in a cast is not qualified by a schema name, the *SQL path* is used to find the first schema that includes the user-defined data type by that name.

The following casts involving reference types are supported:

- cast from reference type *RT* to its representation data type *S*
- cast from the representation data type *S* of reference type *RT* to reference type *RT*
- cast from reference type *RT* with target type *T* to a reference type *RS* with target type *S* where *S* is a supertype of *T*.
- cast from a data type *A* to reference type *RT*, where *A* is promotable to the representation data type *S* of reference type *RT*.

When the target type of a reference data type involved in a cast is not qualified by a schema name, the *SQL path* is used to find the first schema that includes the user-defined data type by that name.

Table 6. Supported Casts between Built-in Data Types

Source Data Type	Target Data Type																		
	S	M	I	D		V	L	V	G	R	L		T	I	M				
SMALLINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	Y ³		
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	Y ³		
BIGINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	Y ³		
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	Y ³		
REAL	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	Y ³		
DOUBLE	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	Y ³		
CHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y ²	Y ¹	Y ¹	-	-	Y	Y	Y	Y	Y ⁴
VARCHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y ²	Y ¹	Y ¹	-	-	Y	Y	Y	Y	Y ⁴
LONG VARCHAR	-	-	-	-	-	-	Y	Y	Y	Y ²	-	-	Y ¹	Y ¹	-	-	-	Y	Y ³
CLOB	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	Y ¹	-	-	-	Y	Y ⁴
GRAPHIC	-	-	-	-	-	-	Y ¹	Y ¹	-	-	Y	Y	Y	Y	Y ¹	Y ¹	Y ¹	Y	Y ³
VARGRAPHIC	-	-	-	-	-	-	Y ¹	Y ¹	-	-	Y	Y	Y	Y	Y ¹	Y ¹	Y ¹	Y	Y ³
LONG VARGRAPHIC	-	-	-	-	-	-	-	-	Y ¹	Y ¹	Y	Y	Y	Y	-	-	-	Y	Y ³
DBCLOB	-	-	-	-	-	-	-	-	-	Y ¹	Y	Y	Y	Y	-	-	-	Y	Y ³
DATE	-	Y	Y	Y	-	-	Y	Y	-	-	Y ¹	Y ¹	-	-	Y	-	-	-	Y ³
TIME	-	Y	Y	Y	-	-	Y	Y	-	-	Y ¹	Y ¹	-	-	-	Y	-	-	Y ³

Table 6. Supported Casts between Built-in Data Types (continued)

Source Data Type	Target Data Type																		
	S	M	I	D		V	V	G	V	A	R	L		T					
	A	N	B	E	D	A	A	R	R	A	R	N	D	E					
	L	T	I	C	O	R	R	A	A	G	B			S					
	L	E	G	I	R	U	C	C	C	P	P	V	C	D	T	T	B		
	I	G	I	M	E	B	H	H	H	L	H	H	A	L	A	I	A	L	X
	N	E	N	A	A	L	A	A	A	O	I	I	R	O	T	M	M	O	M
	T	R	T	L	L	E	R	R	R	B	C	C	G	B	E	E	P	B	L
TIMESTAMP	-	-	Y	Y	-	-	Y	Y	-	-	Y ¹	Y ¹	-	-	Y	Y	Y	-	Y ³
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	Y ⁴
XML	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵

Notes

- See the description preceding the table for information on supported casts involving user-defined types and reference types.
- Only a DATALINK type can be cast to a DATALINK type.
- It is not possible to cast a structured type value to anything else.

¹ Cast is only supported for Unicode databases.

² FOR BIT DATA character types cannot be cast to CLOB.

³ Cast can only be performed using XMLCAST.

⁴ An XMLPARSE function is implicitly processed to convert a string to XML on assignment (INSERT or UPDATE) of a string to an XML column. The string must be a well-formed XML document for the assignment to succeed.

⁵ Cast can only be performed using XMLCAST and depends on the underlying XML schema data type of the XML value. For details, see "XMLCAST".

Casting non-XML values to XML values:

Table 7. Supported Casts from Non-XML Values to XML Values

Source Data Type	Target Data Type	
	XML	Resulting XML Schema Type
SMALLINT	Y	xs:short
INTEGER	Y	xs:int
BIGINT	Y	xs:long
DECIMAL or NUMERIC	Y	xs:decimal
REAL	Y	xs:float
DOUBLE	Y	xs:double
CHAR	Y	xs:string
VARCHAR	Y	xs:string
LONG VARCHAR	Y	xs:string
CLOB	Y	xs:string
GRAPHIC	Y	xs:string
VARGRAPHIC	Y	xs:string

Table 7. Supported Casts from Non-XML Values to XML Values (continued)

Source Data Type	Target Data Type	
	XML	Resulting XML Schema Type
LONG VARCHAR	Y	xs:string
DBCLOB	Y	xs:string
DATE	Y	xs:date
TIME	Y	xs:time
TIMESTAMP	Y	xs:dateTime
BLOB	Y	xs:base64Binary
character type FOR BIT DATA	Y	xs:base64Binary
distinct type		use this chart with the source type of the distinct type

When character string values are cast to XML values, the resulting xs:string atomic value cannot contain illegal XML characters (SQLSTATE 0N002). If the input character string is not in Unicode, the input characters are converted to Unicode.

Casting to SQL binary types results in XQuery atomic values with the type xs:base64Binary.

Casting XML values to non-XML values: An XMLCAST from an XML value to a non-XML value can be described as two casts: an XQuery cast that converts the source XML value to an XQuery type corresponding to the SQL target type, followed by a cast from the corresponding XQuery type to the actual SQL type.

An XMLCAST is supported if the target type has a corresponding XQuery target type that is supported, and if there is a supported XQuery cast from the source value's type to the corresponding XQuery target type. The target type that is used in the XQuery cast is based on the corresponding XQuery target type and might contain some additional restrictions.

The following table lists the XQuery types that result from such conversion.

Table 8. Supported Casts from XML Values to Non-XML Values

Target Data Type	Source Data Type	
	XML	Corresponding XQuery Target Type
SMALLINT	Y	xs:short
INTEGER	Y	xs:int
BIGINT	Y	xs:long
DECIMAL or NUMERIC	Y	xs:decimal
REAL	Y	xs:float
DOUBLE	Y	xs:double
FLOAT	Y	xs:double or xs:float
CHAR	Y	xs:string
VARCHAR	Y	xs:string
LONG VARCHAR	N	not castable
CLOB	Y	xs:string

Table 8. Supported Casts from XML Values to Non-XML Values (continued)

Target Data Type	Source Data Type	
	XML	Corresponding XQuery Target Type
GRAPHIC	Y	xs:string
VARGRAPHIC	Y	xs:string
LONG VARGRAPHIC	N	not castable
DBCLOB	Y	xs:string
DATE	Y	xs:date
TIME (without time zone)	Y	xs:time
TIMESTAMP (without time zone)	Y	xs:dateTime
BLOB	Y	xs:base64Binary
CHAR FOR BIT DATA	N	not castable
VARCHAR FOR BIT DATA	Y	xs:base64Binary
distinct type		use this chart with the source type of the distinct type
row reference structured / ADT other	N	not castable

In the following restriction cases, a derived by restriction XML schema data type is effectively used as the target data type for the XQuery cast.

- XML values that are to be converted to string types must fit within the length limits of those DB2 types without truncation of any characters or bytes. The name used for the derived XML schema type is the uppercase SQL type name followed by an underscore character and the maximum length of the string; for example, VARCHAR_20 if the XMLCAST target data type is VARCHAR(20).
- XML values that are to be converted to DECIMAL values must fit within the precision of the specified DECIMAL values, and must not contain more non-zero digits after the decimal point than the scale. The name used for the derived XML schema type is DECIMAL_precision_scale, where *precision* is the precision of the target SQL data type, and *scale* is the scale of the target SQL data type; for example, DECIMAL_9_2 if the XMLCAST target data type is DECIMAL(9,2).
- XML values that are to be converted to TIME values cannot contain a seconds component with non-zero digits after the decimal point. The name used for the derived XML schema type is TIME.

The derived XML schema type name only appears in a message if an XML value does not conform to one of these restrictions. This type name helps one to understand the error message, and does not correspond to any defined XQuery type. If the input value does not conform to the base type of the derived XML schema type (the corresponding XQuery target type), the error message might indicate that type instead. Because this derived XML schema type name format might change in the future, it should not be used as a programming interface.

Before an XML value is processed by the XQuery cast, any document node in the sequence is removed and each direct child of the removed document node becomes an item in the sequence. If the document node has multiple direct children nodes, the revised sequence will have more items than the original sequence. The XML value without any document nodes is then atomized using the XQuery fn:data function, with the resulting atomized sequence value used in the XQuery cast. If the atomized sequence value is an empty sequence, a null value is

returned from the cast without any further processing. If there are multiple items in the atomized sequence value, an error is returned (SQLSTATE 10507).

If the target type of XMLCAST is the SQL data type DATE, TIME, or TIMESTAMP, the resulting XML value from the XQuery cast is also adjusted to UTC, and the time zone component of the value is removed.

When the corresponding XQuery target type value is converted to the SQL target type, binary XML data types, such as xs:base64Binary or xs:hexBinary, are converted from character form to actual binary data.

If an xs:double or xs:float value of INF, -INF, or NaN is cast (using XMLCAST) to an SQL data type DOUBLE or REAL value, an error is returned (SQLSTATE 22003). An xs:double or xs:float value of -0 is converted to +0.

The target type can be a user-defined distinct type if the source operand is not a user-defined distinct type. In this case, the source value is cast to the source type of the user-defined distinct type (that is, the target type) using the XMLCAST specification, and then this value is cast to the user-defined distinct type using the CAST specification.

Related concepts:

- “Atomization (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Related reference:

- “CREATE FUNCTION statement” in *SQL Reference, Volume 2*
- “Assignments and comparisons” in *SQL Reference, Volume 1*
- “CAST specifications” in *SQL Reference, Volume 1*
- “CURRENT PATH special register” in *SQL Reference, Volume 1*
- “Promotion of data types” in *SQL Reference, Volume 1*
- “XMLCAST specifications” in *SQL Reference, Volume 1*
- “Type casting (DB2 XQuery)” in *IBM DB2 XQuery Reference*

XMLTABLE table function

XMLTABLE overview

XMLTABLE is an SQL table function that returns a table from the evaluation of XQuery expressions. XQuery expressions normally return values as a sequence, however, XMLTABLE allows you to execute an XQuery expression and return values as a table instead. The table that is returned can contain columns of any SQL data type, including XML.

Like the XMLQUERY function, you can pass variables to the XQuery expression specified in XMLTABLE. The result of the XQuery expression is used to generate the column values of the resulting table. The structure of the resulting table is defined by the COLUMNS clause of XMLTABLE. In this clause, you define characteristics of the column by specifying the column name, data type, and how the column value is generated.

The column value of the resulting table can be generated by specifying an XQuery expression in the PATH clause of XMLTABLE. If an XQuery expression is not specified for the PATH clause, the column name is used as the XQuery expression to generate the column value, and the result of the XQuery expression specified

earlier in XMLTABLE becomes the external context item when generating the column value. An optional default clause can also be specified to provide a default value for the column, for the case when the XQuery expression of the PATH clause that generates the column value returns an empty sequence.

If the column type in the resulting table is not XML, and the result of the XQuery expression that defines the value of the column is not an empty sequence, XMLCAST is implicitly used to convert the XML value to a value of the target data type.

The XMLTABLE function allows you to optionally declare namespaces. If you specify namespaces with the XMLNAMESPACES declaration, then these namespace bindings apply to all XQuery expressions in the XMLTABLE function call. If you declare namespace bindings without using the XMLNAMESPACES declaration, then the bindings apply only to the row XQuery expression, which follows the namespace declaration.

Advantages:

Returning a table, instead of a sequence, enables operations, such as the following, to be performed from within an SQL query context:

- iterate 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 "db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo" in XMLTABLE.

```
SELECT X.*
FROM XMLTABLE (xmlnamespaces (DEFAULT "http://posample.org"),
               'db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo'
               COLUMNS "CUSTNAME" CHAR(30) PATH 'name',
                       "PHONENUM" XML PATH 'phone')
              as X
```

- insert values from stored XML documents into tables (refer to the XMLTABLE example on inserting values)
- sort on values from an XML doc

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 XMLTABLE (xmlnamespaces (DEFAULT "http://posample.org"),
               'db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo'
               COLUMNS "CUSTNAME" CHAR(30) PATH 'name',
                       "PHONENUM" XML PATH 'phone')
              as X
ORDER BY X.CUSTNAME
```

- store some XML values as relational and some as XML (refer to the XMLTABLE example on inserting values)

Important: If the XQuery expression specified in the PATH option of XMLTABLE returns:

- a sequence of more than one item, then the data type of the column must be XML. If you are inserting values returned from XMLTABLE into XML columns, ensure that the values inserted are well-formed XML documents. Refer to the XMLTABLE example on inserting values for an example of handling sequences that return more than one item.

- an empty sequence, then a NULL value is returned for that column value.

Related concepts:

- “Introduction to querying XML data with SQL” on page 48
- “XMLQUERY overview” on page 51
- “XMLTABLE example: Returning one row for each occurrence of an item” on page 65
- “XMLTABLE example: Inserting values returned from XMLTABLE” on page 63
- “Native XML Samples” on page 325
- “Expressions (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Sequences and items (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “XML namespaces and qualified names (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Related reference:

- “XMLNAMESPACES declaration” in *SQL Reference, Volume 1*
- “XMLTABLE table function” in *SQL Reference, Volume 1*
- “xmlcolumn function (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Native XML data store roadmap” in *DB2 Information Center*

XMLTABLE example: Inserting values returned from XMLTABLE

The XMLTABLE SQL table function can be used to retrieve values from within stored XML documents, which can then be inserted into a table. 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. (A more general type of decomposition is available with the annotated XML schema decomposition functionality. With annotated XML schema decomposition, you can decompose multiple XML documents at once into multiple tables.)

For example, if the following two XML documents were stored in a table named CUSTOMER:

```
<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>
```

and you wanted to insert values from these documents into a table defined as:
 CREATE TABLE CUSTPHONE (custname char(30), type char(30), numbers XML)

then the following INSERT statement using XMLTABLE populates CUSTPHONE with values from the XML documents:

```
INSERT INTO CUSTPHONE
  SELECT X.*
  FROM XMLTABLE (XMLNAMESPACES (DEFAULT 'http://posample.org'),
    'db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo'
    COLUMNS
      "CUSTNAME" CHAR(30) PATH 'name',
      "PHONENUM" XML PATH 'document{<allphones>{phone}</allphones>}'
  ) as X
```

Notice that the XQuery node constructor function "document{<allphones>{phone}</allphones>}" is specified in the path expression for the PHONENUM column in XMLTABLE. The document constructor is needed because values inserted into XML columns (the NUMBERS column, in this case), must be well-formed XML documents. In this example, all <phone> elements for the <customerinfo> document with Cid="1003" are returned in a single sequence that contains four items:

```
{<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>}
```

This sequence on its own is not a well-formed XML document, and so cannot be inserted into the NUMBERS XML column. To ensure that the phone values are successfully inserted, all items of the sequence are constructed into a single well-formed document.

The resulting table is as follows (the output has been formatted for clarity):

Table 9. Result table

CUSTNAME	NUMBER
Kathy Smith	<allphones xmlns="http://posample.org"><phone type="work">905-555-7258</phone></allphones>
Robert Shoemaker	<allphones xmlns="http://posample.org"><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></allphones>

Related concepts:

- "Annotated XML schema decomposition" on page 235
- "XMLTABLE example: Returning one row for each occurrence of an item" on page 65
- "XMLTABLE overview" on page 61
- "Expressions (DB2 XQuery)" in *IBM DB2 XQuery Reference*
- "Sequences and items (DB2 XQuery)" in *IBM DB2 XQuery Reference*

Related reference:

- "XMLTABLE table function" in *SQL Reference, Volume 1*
- "Document node constructors (DB2 XQuery)" in *IBM DB2 XQuery Reference*

XMLTABLE example: Returning one row for each occurrence of an item

If your XML documents contain multiple occurrences of an element and you want to generate a row for each occurrence of this element, you can use XMLTABLE to achieve this effect.

For example, if the following two XML documents were stored in a table named CUSTOMER:

```
<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>
```

to create a table where every <phone> value is stored in a separate row, use XMLTABLE as follows:

```
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" CHAR(15) PATH '.'
  ) as X
```

This query yields the following result for the two XML documents:

Table 10. Result table

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

Notice how each <phone> element for the XML document with the name "Robert Shoemaker" are returned in a separate row.

For the same documents, you can also extract the <phone> elements as XML, as follows:

```

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

```

This query yields the following result for the two XML documents (the output has been formatted for clarity):

Table 11. Result table

CUSTNAME	PHONETYPE	PHONENUM
Kathy Smith	work	<phone xmlns="http://posample.org" type="work">416-555-1358</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>

Related concepts:

- “Introduction to querying XML data with SQL” on page 48
- “XMLTABLE example: Inserting values returned from XMLTABLE” on page 63
- “XMLTABLE overview” on page 61
- “Expressions (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Sequences and items (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Related reference:

- “XMLTABLE table function” in *SQL Reference, Volume 1*

XMLEXISTS predicate

XMLEXISTS predicate when querying XML data

The XMLEXISTS predicate determines whether an XQuery expression returns a sequence of one or more items. If the XQuery expression specified in this predicate returns an empty sequence, XMLEXISTS returns false; otherwise, true is returned.

The XMLEXISTS predicate can be used in the WHERE clauses of SELECT statements. This usage means that values from stored XML documents can be used to restrict the set of rows that the SELECT query operates on.

For example, the following SQL query shows how the XMLEXISTS predicate can be used to restrict the rows returned, to only those that contain an XML document with a <city> element that has the value “Toronto”. (Note that XQuery expressions are case-sensitive, while SQL is case-insensitive.)

```

SELECT Cid
FROM CUSTOMER
WHERE XMLEXISTS ('declare default element namespace "http://posample.org";
                $d//addr[city="Toronto"]' passing INFO as "d")

```

Notice how you can pass values to XQuery variables in the XQuery expression of XMLEXISTS. In this case, the XQuery variable \$d is bound to the documents of the INFO column of the CUSTOMER table.

Ensure that the XQuery expression in XMLEXISTS is correctly specified in order to return expected results. For example, assume that there are multiple documents stored in the XML INFO column of the CUSTOMER table, but only one document contains a Cid attribute (along the path specified) with a value of 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>
```

The following two queries return different results, because of the slight difference in the XQuery expressions:

```
SELECT *
FROM CUSTOMER
WHERE XMLEXISTS ('declare default element namespace "http://posample.org";
                $d/customerinfo[@Cid=1000]' passing INFO as "d")
```

```
SELECT *
FROM CUSTOMER
WHERE XMLEXISTS ('declare default element namespace "http://posample.org";
                $d/customerinfo/@Cid=1000' passing INFO as "d")
```

The first query returns the row containing the XML document presented above, as expected. The second query, however, returns all rows of the CUSTOMER table because the XMLEXISTS predicate always returns true for the XQuery expression specified. The XQuery expression in the second query returns a sequence of boolean items, which is a non-empty sequence, causing XMLEXISTS to always return true. This then causes every row in the CUSTOMER table to be selected, which is not the intended result.

Related concepts:

- “Constant and parameter marker passing to XMLEXISTS and XMLQUERY” on page 69
- “Expressions (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Native XML Samples” on page 325
- “Sequences and items (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Related reference:

- “XMLEXISTS predicate” in *SQL Reference, Volume 1*
- “Variable references in DB2 XQuery” in *IBM DB2 XQuery Reference*
- “Native XML data store roadmap” in *DB2 Information Center*

XMLEXISTS predicate usage

If XMLEXISTS includes an XPath expression with a value predicate (*expression*), enclose the predicate in square brackets, such that [*expression*] is the result. Enclosing the value predicate in square brackets ensures evaluation of *expression* in accordance with what may be semantically expected.

Example 1. The following scenario demonstrates how a non-empty sequence causes XMLEXISTS to evaluate *true*, even though the non-empty sequence itself consists of the single value *false*. No index matching takes place and the query returns a much greater set of results than expected. The issue is avoided by bracketing value predicates appropriately with with square brackets ([]).

Consider a table, an index and two queries:

```
CREATE TABLE mytable (id BIGINT, xmlcol XML);
CREATE INDEX myidx ON mytable(xmlcol)
GENERATE KEY USING XMLPATTERN '//text()' AS SQL VARCHAR(255);

SELECT xmlcol FROM mytable
WHERE XMLEXISTS('$doc/CUSTOMER/ORDERS/ORDERKEY/text()='A512' '
PASSING xmlcol AS "doc")

SELECT xmlcol FROM mytable
WHERE XMLEXISTS('$doc/CUSTOMER[ORDERS/ORDERKEY/text()='A512"] '
PASSING xmlcol AS "doc") ;
```

The cause for this behavior is as follows: XMLEXISTS evaluates the XQuery expression and returns *false* (for XMLEXISTS) if the result is the empty sequence, and *true* (for XMLEXISTS) if the result is a non-empty sequence. This is followed by a possibly non-intuitive next step in the query evaluation: In the first query the expression instructs to 'compare the order key to A512'. The result of that expression is either the value *false* or *true* depending on the actual value of the order key. Therefore, the XMLEXISTS function always sees a return sequence with a single item in it, that is, an item that is *false* or an item that is *true*. Since a sequence with one item is a non-empty sequence, XMLEXISTS **always** returns *true* (for XMLEXISTS) overall and therefore the query returns all rows. Indexes cannot be leveraged if XMLEXISTS is used such that all rows are qualified.

Below are 5 examples of non-empty sequences, where 3 are sequences which contain just 1 item:

```
(42, 3,4,78, 1966)
(true)
(abd, def)
(false)
(5)
```

Any such non-empty sequence will cause XMLEXISTS to return the value *true* (for XMLEXISTS), even if the non-empty sequence it encounters itself returns (*false*).

In the second query, the expression inside XMLEXISTS instructs to 'return the customers which contain an order with an orderkey equal to A512'. If no such customer exists in a document, then the result is indeed an empty sequence. This query will use an index and it will return the expected results.

When the entire *expression* is placed in square brackets, the meaning is fixed to be 'return XML data if [*expression*]', and you should always be returned an empty sequence if the XML data does not satisfy *expression*.

Since the value comparison is always within square brackets, each of the following sample fragments for XMLEXISTS predicate usage works as expected:

```
... WHERE XMLEXISTS('$doc[CUSTOMER/ORDERS/ORDERKEY/text()='A512"] '
PASSING xmlcol as "doc") ;
... WHERE XMLEXISTS('$doc/CUSTOMER[ORDERS/ORDERKEY/text()='A512"] '
PASSING xmlcol AS "doc") ;
... WHERE XMLEXISTS('$doc/CUSTOMER/ORDERS[ORDERKEY/text()='A512"] '
PASSING xmlcol AS "doc") ;
```

The guideline also works for queries where there is no value comparison, for example, when you want to return the documents for all customers which happen to have a *COMMENT* child element:

```
... WHERE XMLEXISTS('$doc[CUSTOMER/COMMENT ] '
      PASSING xmlcol AS "doc") ;
```

Related concepts:

- “Guidelines for matching indexes with queries overview” on page 72

Related reference:

- “CREATE INDEX ” on page 108
- “XMLEXISTS predicate” in *SQL Reference, Volume 1*

Constant and parameter marker passing to XMLEXISTS and XMLQUERY

XQuery variables can be specified as part of the XQuery expression in XMLEXISTS and XMLQUERY. Values are passed into these variables through the passing clause. These values are SQL expressions.

Because the values passed to the XQuery expression are non-XML values, they must be cast, either implicitly or explicitly, to types supported by DB2 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. This constant 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 parameter marker must be explicitly cast to a data type because the type of the parameter marker cannot be determined. The parameter marker that is explicitly cast to an SQL VARCHAR type is then implicitly cast to the xs:string XML schema 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",
      CAST (? AS VARCHAR(128)) AS "cityName")
```

The method of passing constants and parameter markers to XMLQUERY is the same as that of XMLEXISTS, however, the XMLEXISTS usage is more common. This is because parameterized predicates in XMLQUERY, when used in SELECT clauses, do not eliminate any rows from the result set. Instead, the predicates are

used to determine which fragments of a document are returned. To actually eliminate rows from a result set, the XMLEXISTS predicate should be used in the WHERE clause. Rows that contain empty sequences are therefore not returned as part of the result set. The examples discussed here show this more common usage with XMLEXISTS.

Refer to the documentation on casting between data types for more information on supported casts.

Related concepts:

- “XMLEXISTS predicate when querying XML data” on page 66
- “XMLQUERY overview” on page 51
- “Type system (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “XQuery” in *IBM DB2 XQuery Reference*
- “Expressions (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Related reference:

- “Casting between data types” on page 56
- “Expressions” in *SQL Reference, Volume 1*
- “XMLEXISTS predicate” in *SQL Reference, Volume 1*
- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “Variable references in DB2 XQuery” in *IBM DB2 XQuery Reference*

Data retrieval with XQuery

The XQuery specification defines the result of an XQuery expression as a sequence that contains 0, 1, or more items. XQuery expressions can be executed in DB2 either using SQL as the primary language, with the XMLQUERY SQL function, or using XQuery as the primary language. When an XQuery expression is executed using either method, an XML sequence is returned. How the resulting sequence appears in a result set, however, differs depending on whether SQL or XQuery is used as the primary language.

- XQuery as the primary language: When an XQuery expression is executed using XQuery as the primary language, the result is returned to a client application as a result table with one column, which is of type XML. Each row in this result table is an item of the sequence that resulted from the evaluation of the XQuery expression. When an application fetches from this result table using a cursor, each fetch, in effect, retrieves a serialized item of the resulting sequence.
- SQL as the primary language, using XMLQUERY: XMLQUERY is a scalar function that returns an XML value. The value that is returned is a sequence of 0, 1, or more items. All items of the resulting sequence are returned to an application as a single serialized value.

The example that follows illustrates the difference between the result sets from these two querying methods.

To fetch results from queries that use XQuery or XMLQUERY, fetch results from within your application, as you normally would any other result set. Bind your application variable to the result set and fetch until the end of the result set. If the XQuery expression (issued directly or through XMLQUERY) returned an empty sequence, then the row in the result set is also empty.

Example: Difference in result sets from XQuery and XMLQUERY:

If the following two XML documents are stored in an XML column, to retrieve all `<phone>` elements, you can use either XQuery or XMLQUERY. The result sets that are returned by these two methods differ, however, and should be handled accordingly by the application when fetching from the result set.

```
<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>

<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>
```

Executing an XQuery expression using XQuery as the primary language yields 5 rows in the result set, as follows:

```
XQUERY declare default element namespace "http://posample.org";
db2-fn:xmlcolumn ('CUSTOMER.INFO')/customerinfo/phone
```

Table 12. Result set using XQuery as the primary language

<code><phone xmlns="http://posample.org" type="work">416-555-1358</phone></code>
<code><phone xmlns="http://posample.org" type="work">905-555-2937</phone></code>
<code><phone xmlns="http://posample.org" type="home">416-555-2937</phone></code>
<code><phone xmlns="http://posample.org" type="cell">905-555-8743</phone></code>
<code><phone xmlns="http://posample.org" type="cottage">613-555-3278</phone></code>

Executing an XQuery expression through XMLQUERY yields 2 rows in the result set, as follows, where all `<phone>` elements of the second row in the table are concatenated in a single scalar value (an XML sequence):

```
SELECT XMLQUERY ('declare default element namespace "http://posample.org";
  $doc/customerinfo/phone' PASSING INFO AS "doc")
FROM CUSTOMER
```

Table 13. Result set using SQL as the primary language

<code><phone xmlns="http://posample.org" type="work">416-555-1358</phone></code>
<code><phone xmlns="http://posample.org" type="work">905-555-2937</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></code>

Notice that the second row of this result set contains a value that is not a well-formed XML document.

These differences in the result sets exist because XMLQUERY is a scalar function. It executes on each row of the table and the resulting sequence from a row of the table, forms a row of the result set. XQuery, however, returns each item of a sequence as a separate row of the result set.

Managing query result sets:

If your application requires that the XML values returned when querying with XQuery are necessarily well-formed XML documents (for example, if you plan to insert these values into a column of type XML), then you can ensure that the values are well-formed XML documents by including an element or document constructor in your XQuery expression.

For example, the previous SQL query can be modified to include an XQuery document node constructor that ensures the resulting rows all contain well-formed documents:

```
SELECT XMLQUERY ('declare default element namespace "http://posample.org";
  document{<phonelist>{$doc/customerinfo/phone}</phonelist>}'
  PASSING INFO AS "doc")
FROM CUSTOMER
```

The table that results from this query, assuming the same documents presented previously exist in database, is as follows.

Table 14. Using element constructors to yield well-formed documents

<phonelist xmlns="http://posample.org"><phone type="work">416-555-1358</phone></phonelist>
<phonelist xmlns="http://posample.org"><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></phonelist>

Related concepts:

- “Application programming language support for XML” on page 165
- “Empty sequences returned by XMLQUERY” on page 53
- “XML serialization” on page 86
- “Non-empty sequences returned by XMLQUERY” on page 52
- “XMLQUERY overview” on page 51
- “Expressions (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “Retrieving DB2 data with XQuery functions” on page 46
- “Sequences and items (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Related reference:

- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “Document node constructors (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Guidelines for matching indexes with queries

Guidelines for matching indexes with queries overview

This section provides you with some guidelines and examples for matching queries with indexes over XML data. Whether a query can make use of an index depends on whether the index or indexes you created are compatible with your query (also known as index matching), and whether the optimizer chooses to perform an

index scan during query evaluation. The access plan of the Explain facility will tell you if query evaluation involved an index scan.

A query must meet at least the following conditions before it can use an index over XML data:

- The data types in the query search condition match the indexed data types.
- The query search condition includes a subset of the nodes that are indexed.

SQL and XQuery optimizer:

The optimizer plans the evaluation of queries and chooses which indexes to use during evaluation. During query compilation, a query is matched with all the patterns in the XML index definitions to find candidate indexes that contain sufficient information to answer some part of the query.

The optimizer may take one of the following steps during query evaluation:

- Scan the table containing the XML documents, without using an index
- Use a relational index
- Use relational index ANDing or index ORing
- Use a new XML index operator
- Use an index over XML data for the evaluation of a single XML pattern
- Use index over XML data ANDing and ORing for the evaluation of complex XML patterns from a single query

Explain facility:

The Explain facility and the Visual Explain tool can provide you with the access plan that is chosen to evaluate your query. When you look at an access plan, the following operators will tell you whether one or several indexes were used during query evaluation:

IXAND

ANDs the row IDs from two or more index scans.

XISCAN

Scans an index over XML data.

XANDOR

Allows ANDed predicates to be applied to multiple XML indexes.

Related concepts:

- “Considerations when specifying text()” on page 75
- “Data type of literals” on page 76
- “Explain facility” in *Performance Guide*
- “Indeterminate query evaluation” on page 79
- “Join predicate conversion” on page 77
- “Operator” in *Administration Guide: Implementation*
- “Restrictiveness of index definitions” on page 74
- “Sample queries against indexes over XML data” on page 124
- “XMLEXISTS predicate usage” on page 67

Restrictiveness of index definitions

Example 1. An index over XML data for a query with a range predicate: This query retrieves company information for employees with a salary greater than 35000 from the table *companyinfo* with the XML column *companydocs*.

```
SELECT companydocs FROM companyinfo
WHERE XMLEXISTS('$x/company/emp[@salary > 35000]'
PASSING companydocs AS "x")
```

To be compatible, the index over XML data needs to include employee salary attribute nodes among the indexed nodes, and to store values as DOUBLE type.

The query could use either one of the following indexes over XML data, for example:

```
CREATE INDEX empindex on companyinfo(companydocs)
GENERATE KEY USING XMLPATTERN '//@salary' AS SQL DOUBLE

CREATE INDEX empindex on companyinfo(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@salary'
AS SQL DOUBLE
```

Example 2. An index over XML data that can be used by different queries: This query retrieves company information for employees with the employee ID 31664.

```
SELECT companydocs FROM companyinfo
WHERE XMLEXISTS('$x/company/emp[@id="31664"]'
PASSING companydocs AS "x")
```

This query retrieves company information for departments with the ID K55.

```
SELECT companydocs FROM companyinfo
WHERE XMLEXISTS('$x/company/emp/dept[@id="K55"]'
PASSING companydocs AS "x")
```

To be compatible with both queries, the index over XML data needs to include employee ID attribute nodes and department ID attribute nodes among the indexed nodes, and to store values in the index as a VARCHAR type.

The queries can use this index over XML data:

```
CREATE INDEX empdeptindex on companyinfo(companydocs)
GENERATE KEY USING XMLPATTERN '//@id' AS SQL VARCHAR(25)
```

Example 3. Including namespaces when restricting the XQuery/XPath predicate: Consider the following table with an XML column that contains customer information, and an index that is created on the XML column:

```
CREATE TABLE customer(xmlcol XML) %

CREATE UNIQUE INDEX customer_id_index ON customer(xmlcol)
GENERATE KEY USING XMLPATTERN
'DECLARE DEFAULT ELEMENT NAMESPACE
"http://mynamespace.org/cust";/Customer/@id'
AS SQL DOUBLE %
```

Note: The statement terminator used in this section is the percentage sign (%), since the semicolon (;) already serves as the namespace delimiter.

The following query fails to match the index:

```
SELECT xmlcol FROM customer
WHERE XMLEXISTS('$xmlcol/*:Customer[@id=1042]'
PASSING xmlcol AS "xmlcol") %
```

In order for the query to be able to use the index, the query must be as restrictive as or more restrictive than the index. The index `customer_id_index` covers only customer elements in one particular namespace (`http://mynamespace.org/cust`). Since `*` is used in the query to denote any namespace, the index is not used. This can be counter-intuitive if one expects `*` to match the namespace in the index definition.

For the query to make use of the index, either the index needs to become less restrictive, or the query needs to become more restrictive.

The following less restrictive index `customer_id_index2` could be used successfully with the same query:

```
CREATE UNIQUE INDEX customer_id_index2 ON customer(xmlcol)
    GENERATE KEY USING XMLPATTERN '/*:Customer/@id' AS SQL DOUBLE %
```

The following more restrictive query can make use of the initial index `customer_id_index`:

```
SELECT xmlcol FROM customer
WHERE XMLEXISTS('
  DECLARE NAMESPACE ns = "http://mynamespace.org/cust";
  $xmlcol/ns:Customer[@id=1042]'
  PASSING xmlcol AS "xmlcol") %
```

When the appropriate namespace is specified explicitly in the query, the index `customer_id_index` can be used, since the query is now just as restrictive as the index. The index `customer_id_index2` could also be used, since it is less restrictive than the query in this example.

Considerations when specifying text()

The inclusion of the `text()` node with XML pattern expressions may affect the generation of index entries. Use `/text()` consistently in index definitions and predicates

Example 1. Consider the following sample XML document fragment:

```
<company name="Company1">
  <emp id="31201" salary="60000" gender="Female">
    <name><first>Laura</first><last>Brown</last></name>
    <dept id="M25">
      Finance
    </dept>
  </emp>
</company>
```

If the following index is created with `text()` specified at the end of the pattern, no index entries will be inserted because the `name` elements in the sample XML document fragments do not contain text themselves. Text is found only in the child elements, `first` and `last`.

```
CREATE INDEX nameindex on company(companydocs)
    GENERATE KEY USING XMLPATTERN '/company/emp/name/text()' AS SQL
    VARCHAR(30)
```

However, if the next index is created with the element `name` specified at the end of the pattern, the text from the `first` and `last` child elements will be concatenated in the inserted index entries.

```
CREATE INDEX nameindex on company(companydocs)
    GENERATE KEY USING XMLPATTERN '/company/emp/name'
    AS SQL VARCHAR(30)
```

The presence or absence of the text() node will affect index entry generation for non leaf elements, but not for leaf elements. If you are indexing leaf elements, specifying text() is not recommended. If you do specify text(), queries must also use text() for successful index matching. In addition, schema validation applies only to elements and not to text nodes.

Caution must be used when specifying XML patterns that can match elements which are non leaf nodes without text(). The concatenation of descendant element text nodes can cause unexpected results. Especially specifying `//*` with an XML pattern will most likely index a non leaf element.

In some cases, concatenation can be useful for indexes using VARCHAR. For example, an index on `/title` in the document fragment below may be useful for ignoring the bold formatting within the title:

```
<title>This is a <bold>great</bold> book about XML</title>
```

A query predicate to look for a specific employee name could be written as follows:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[name='LauraBrown']
```

Whitespace is significant in the predicate and the document. If a space is inserted between 'Laura' and Brown' in the predicate, nothing will be returned for the following query, since the sample XML document fragment itself does not contain a space between the first and last names:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[name='Laura Brown']
```

Example 2. An index over XML data for a query with a compound equal predicate: This query retrieves company information for employees who are in the Finance or Marketing departments.

```
SELECT companydocs FROM companyinfo WHERE
  XMLExists('$x/company/emp[dept/text()='Finance'
  or dept/text()='Marketing']')
  PASSING companydocs AS "x")
```

To be compatible, the index over XML data needs to index the text node for the department of each employee among the indexed nodes and to store values as a VARCHAR type.

The query can use this index over XML data:

```
CREATE INDEX empindex on companyinfo(companydocs)
  GENERATE KEY USING XMLPATTERN '/company/emp/dept/text()'
  AS SQL VARCHAR(30)
```

Related concepts:

- “Guidelines for matching indexes with queries overview” on page 72

Data type of literals

The data types of literals need to match the data type of the index in order for the query to be able to make use of the index.

Example: This query retrieves company information for employees with the ID 31201.

```
SELECT companydocs FROM companyinfo
  WHERE XMLEXISTS('$x/company/emp[@id="31201"]')
  PASSING companydocs AS "x")
```

To be compatible, the index over XML data needs to include the employee ID attribute nodes among the indexed nodes, and to store values in the index as a VARCHAR type.

The query can use this index over XML data, for example:

```
CREATE INDEX empindex on companyinfo(companydocs)
  GENERATE KEY USING XMLPATTERN '/company/emp/@id'
  AS SQL VARCHAR(5)
```

If a similar index were defined AS SQL DOUBLE it could not be used by the query, since the query predicate includes a string comparison. The double quotation marks used in the predicate `@id="31201"` make it a string comparison, which can only be evaluated with a string index (VARCHAR), not with a numeric index (DOUBLE).

To highlight the difference between numeric predicates and string predicates, consider the following inequality predicates:

```
@id > 3
@id > "3"
```

The numeric predicate `@id > 3` is different from the string predicate `@id > "3"`. The numeric predicate `@id > 3` would be fulfilled by an `@id` value of `10`, but the string predicate `@id > "3"` would not be, because in a string comparison `"3"` is greater than `"10"`.

Related concepts:

- “Guidelines for matching indexes with queries overview” on page 72

Join predicate conversion

Join predicates should be converted to the appropriate data type on both sides.

Example 1. Consider two tables with XML columns for customer information and purchase orders, respectively:

```
CREATE TABLE customer(info XML);
CREATE TABLE PurchaseOrder(POrder XML);
```

The XML documents that contain customer information include an attribute `@cid`, the numeric customer ID (`cid`). The XML documents that contain purchase order information also include `@cid` attributes, so that each order is uniquely associated with a particular customer. Since we expect customers and orders to be frequently searched by `cid`, it makes sense to define indexes:

```
CREATE UNIQUE INDEX idx1 ON customer(info)
  GENERATE KEY USING XMLPATTERN '/customerinfo/@cid' AS SQL DOUBLE;

CREATE INDEX idx2 ON PurchaseOrder(POrder)
  GENERATE KEY USING XMLPATTERN '/porder/@cid' AS SQL DOUBLE;
```

We wish to find the purchase orders for all customers in a specific ZIP code. Intuitively, we can write the query like this:

```
XQUERY
  for $i in db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo
  for $j in db2-fn:xmlcolumn("PURCHASEORDER.PORDER")/porder[@cid = $i/@cid]
  where $i/zipcode = "95141"
  return $j;
```

Note that the join predicate `@cid = $i/@cid` requires the purchase order's *cid* to be equal to the customer *cid*.

This query returns the correct result, but neither of the two indexes can be used. The query is executed as a nested loop join with table scans for both tables. To avoid repeated table scans, a single table scan on *customer* to find all customers within ZIP code *95141* is preferable, followed by index lookups into the purchase order table using `@cid`. Note that it is necessary to scan the customer table since we have no index on *zipcode*.

The index is not used, because it would be incorrect to do so. If the index were used, DB2 may miss some matching purchase orders and return an incomplete result. This is because some values in the `@cid` attribute could potentially be non numeric. For example, `@cid` could equal *YPS* and thus not be included in the numeric index which is defined AS SQL DOUBLE.

Note: If a value of an indexed node cannot be converted to the specified index data type, the index entry for that value is not inserted and no error or warning is raised.

It is possible to enable the index to be used which is particularly desirable if we are certain that all our `@cid` values are numeric. The index will be used if we convert the join predicate explicitly to match the type of the index:

```
XQUERY
  for $i in db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo
  for $j in db2-fn:xmlcolumn("PURCHASEORDER.PORDER")/porder
  where $i/@cid/xs:double(.) = $j/@cid/xs:double(.)
  and $i/zipcode = "95141"
  return $j;
```

Intuitively, the conversion advises DB2 that it should consider only matching `@cid` attributes which are convertible to DOUBLE. With this directive, we can be sure that all required matches are represented in the index defined AS SQL DOUBLE, and thus that it is safe to use that index. If there does exist a non numeric `@cid` value in one of the documents, then the conversion will fail with a run-time error.

Note that within XQuery, casting works only for singletons. Especially for elements (*a*, *b*, and *c* in the example below), it is recommended that you convert them like this:

```
/a/b/c/xs:double(.)
```

If you were to convert the elements as follows, a run-time error would result if multiple elements *c* exist under any given element *b*:

```
/a/b/xs:double(c)
```

For indexes that are defined AS SQL VARCHAR, the corresponding join predicates need to convert the compared values to the `xs:string` data type using the `fn:string()` function. The same applies to DATE and TIMESTAMP indexes. The following example shows how the `fn:string()` function is used in a string join:

```
XQUERY
  for $i in db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo
  for $j in db2-fn:xmlcolumn("PURCHASEORDER.PORDER")/porder
  where $i/zipcode/fn:string(.) = $j/supplier/zip/fn:string(.)
  return <pair>{$i}{$j}</pair>
```

The conversion rules for join predicates are summarized in the table below:

Table 15. Conversion rules for join predicates

Index SQL type	Convert join predicate to XML type
DOUBLE	xs:double
VARCHAR <i>integer</i> , VARCHAR HASHED	xs:string
DATE	xs:date
TIMESTAMP	xs:dateTime

Related concepts:

- “Guidelines for matching indexes with queries overview” on page 72
- “Sample queries against indexes over XML data” on page 124

Indeterminate query evaluation

A query may evaluate indeterminately and return an error when no index scan is involved. The same query may return matching XML data without error when query evaluation involves an index scan, because non-castable XML fragments that cause the error are left out of the index.

Example 1. Consider the following query that attempts to fetch the employee whose ID is 17:

```
for $i in db2-fn:xmlcolumn("T.DOC")
  where $i/emp/id = 17
  return $i
```

The table *T.DOC* contains the following two XML fragments:

```
<emp><id>ABC</id></emp>
```

```
<emp><id>17</id></emp>
```

This query will return an error, unless an index over XML data is used to find the matching documents in the table. If a table scan is used, the predicate is also applied to the non-matching document in the table and a run-time error will result, because the value *ABC* cannot be converted to a number. If an index over XML data that indexes employee IDs exists on the same table, the same query will return the second XML fragment without error whilst the non-castable XML fragment was left out of the index.

The access plan provided by the Explain facility will show whether the evaluation of a query involved an index scan.

Related concepts:

- “Guidelines for matching indexes with queries overview” on page 72

Full-text search in XML documents

Full-text search on natively stored XML data is possible through DB2 Net Search Extender V9.1. DB2 Net Search Extender V9.1 is part of a separately orderable feature.

DB2 Net Search Extender fully supports the XML data type. It provides full-text indexing of documents stored in XML columns. By creating a text index on an

XML column, you can query all text within an XML document and perform searches such as proximity or wildcard searches.

The following example shows a simple full-text search that finds the word "marketing" anywhere within the path /dept/description of XML documents stored in the DEPTDOC column:

```
SELECT DEPTDOC
FROM DEPT
WHERE contains (DEPTDOC, SECTIONS("/dept/description") "marketing") = 1
```

The contains function, provided by DB2 Net Search Extender, searches for the string "marketing" in any text under the path /dept/description, including element or attribute names, and element or attribute values.

To use full-text search, SQL must be used. The results from the SQL query, however, can still be returned to an XQuery context for further processing. The following example shows how the results from an SQL query that used full-text search can participate in an XQuery expression:

```
XQUERY for $i in db2-fn:sqlquery ('SELECT DEPTDOC FROM DEPT
WHERE contains
(DEPTDOC, SECTIONS("/dept/description") "marketing") = 1')//employee
return $i/name
```

In this example, the results of the SQL query, which exploited full-text search, are returned to the for clause of the XQuery FLWOR expression. The for clause then returns all <employee> elements, and the return clause returns the <name> elements within the retrieved <employee> elements.

For more information on full-text search, refer to the DB2 Net Search Extender documentation and see the Web at www.ibm.com/software/data/db2/extendernetsearch.

Related concepts:

- "Native XML data store overview" on page 1
- "XML data querying" on page 45
- "XML data type" on page 17

Related reference:

- "FLWOR expressions (DB2 XQuery)" in *IBM DB2 XQuery Reference*

Retrieval of data in XML columns to earlier DB2 clients

If you retrieve data from an XML column to a client that is at an earlier release than DB2 Version 9.1, 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 serialized string representation of the XML data, with a complete XML declaration.

If you want to receive the data as a data type other than a BLOB data type, use one of the following methods:

- To retrieve the data as CLOB data, ask the administrator of the database server to use the db2set command to set the DB2_MAP_XML_AS_CLOB_FOR_DLC registry variable to YES on the server.

Important: When you set the DB2_MAP_XML_AS_CLOB_FOR_DLC registry variable to YES on the database server, all DB2 clients at earlier release levels that connect to any database within the instance receive XML data as CLOB data.

Important: When DB2_MAP_XML_AS_CLOB_FOR_DLC registry variable to YES on the database server, clients receive CLOB data that is the serialized string representation of the XML data, **without** an XML declaration.

- To retrieve the data as CLOB, CHAR, or VARCHAR data, 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 or CLOB 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 or CLOB data.

Related concepts:

- “Native XML data store overview” on page 1
- “XML data type” on page 17
- “XML serialization” on page 86
- “Application programming language support for XML” on page 165

Related reference:

- “Miscellaneous variables” in *Performance Guide*
- Chapter 9, “Restrictions on native XML data store,” on page 233
- “XMLSERIALIZE scalar function” in *SQL Reference, Volume 1*
- “LIKE predicate” in *SQL Reference, Volume 1*

Publishing

Publishing XML values with SQL/XML

The following SQL/XML publishing functions can be used together to construct XML values.

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 XQuery 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 XQuery 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 serialized representation of XML, however, every document that is to be stored in a DB2 table must contain a document node. Note that the XMLELEMENT function does not create a document node, only an element node. Thus when constructing XML documents that are to be inserted, it is not sufficient to create only an element node. The document must contain a document node.

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, XMLFOREST, and XMLTABLE functions.

XMLPI scalar function

Returns an XML value with a single XQuery processing instruction node.

XMLTEXT scalar function

Returns an XML value with a single XQuery text node having the input argument as the content.

Construct XML values, which do not necessarily have to be well-formed XML documents, by combining the functions that correspond to the components you want in the resulting the XML value. The functions must be specified in the order that you want the results to appear.

Example: Construct an XML document with constant values:

As a simple example, consider the following XML element:

```
<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, perform the following steps:

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 forest of elements that are named "child1" and "child2" using the XMLFOREST function, within the XMLELEMENT function call for <elem1>.

These steps are combined into the following query:

```
VALUES XMLELEMENT (NAME "elem1",
                  XMLNAMESPACES (DEFAULT 'http://posample.org'),
                  XMLATTRIBUTES ('111' AS "id"),
                  XMLCOMMENT ('example document'),
                  XMLFOREST('abc' as "child1",
                           'def' as "child2"))
```

Example: Construct an XML document with values from a single table:

This example shows how an XML document can be constructed from values stored in a single table. In the following query, each <item> element is constructed with values from the name column of the PRODUCT table, using the XMLELEMENT function. All <item> elements are then aggregated, using XMLAGG, within the constructed <allProducts> element. A namespace is also added to the <allProducts> element, with the XMLNAMESPACES function.

```
SELECT XMLELEMENT (NAME "allProducts",
                  XMLNAMESPACES (DEFAULT 'http://posample.org'),
                  XMLAGG(XMLELEMENT (NAME "item", p.name)))
FROM Product p
<allProducts xmlns="http://posample.org">
  <item>Snow Shovel, Basic 22 inch</item>
  <item>Snow Shovel, Deluxe 24 inch</item>
  <item>Snow Shovel, Super Deluxe 26 inch</item>
  <item>Ice Scraper, Windshield 4 inch</item>
</allProducts>
```

Example: Construct an XML document with values from multiple tables:

This example shows how an XML document can be constructed from values stored in multiple tables. In the following query, <prod> elements are constructed from a forest of elements, which are called name and numInStock, using the XMLFOREST function. This forest is built with values from the PRODUCT and INVENTORY tables. All <prod> elements are then aggregated within the constructed <saleProducts> element.

```
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
```

The previous query yields the following XML document:

```
<saleProducts xmlns="http://posample.org">
  <prod id="100-100-01">
    <name>Snow Shovel, Basic 22 inch</name>
    <numInStock>5</numInStock>
  </prod>
  <prod id="100-101-01">
    <name>Snow Shovel, Deluxe 24 inch</name>
    <numInStock>25</numInStock>
  </prod>
  <prod id="100-103-01">
```

```

        <name>Snow Shovel, Super Deluxe 26 inch</name>
        <numInStock>55</numInStock>
    </prod>
    <prod id="100-201-01">
        <name>Ice Scraper, Windshield 4 inch</name>
        <numInStock>99</numInStock>
    </prod>
</saleProducts>

```

Values constructed using the SQL/XML publishing functions are returned as XML. Depending on what you want to do with the XML value, you might need to explicitly serialize the value to convert it to another SQL data type. Refer to the documentation on XML serialization for details.

Null element values:

When an XML value is constructed using XMLELEMENT or XMLFOREST, it is possible that a null value is encountered when determining the element's content. The EMPTY ON NULL and NULL ON NULL options of XMLELEMENT and XMLFOREST allow you to specify whether an empty element or no element is generated when an element's content is null. The default null handling for XMLELEMENT is EMPTY ON NULL. The default null handling for XMLFOREST is NULL ON NULL.

For example, assume that the LOCATION column of the INVENTORY table contains a null value in one row. The following query therefore does not return the <loc> element, because XMLFOREST treats nulls as null by default:

```

SELECT XMLELEMENT (NAME "newElem",
                  XMLATTRIBUTES (PID AS "prodID"),
                  XMLFOREST (QUANTITY as "quantity",
                           LOCATION as "loc"))
FROM INVENTORY
<newElem prodID="100-100-01"><quantity>5</quantity></newElem>

```

The same query, with the EMPTY ON NULL option specified, returns an empty <loc> element:

```

SELECT XMLELEMENT (NAME "newElem",
                  XMLATTRIBUTES (PID AS "prodID"),
                  XMLFOREST (QUANTITY as "quantity",
                           LOCATION as "loc" OPTION EMPTY ON NULL))
FROM INVENTORY
<newElem prodID="100-100-01"><quantity>5</quantity><loc/></newElem>

```

Related concepts:

- "Native XML Samples" on page 325
- "Native XML data store overview" on page 1
- "Special character handling in SQL/XML publishing functions" on page 85
- "XML data type" on page 17
- "XML serialization" on page 86

Related reference:

- Chapter 9, "Restrictions on native XML data store," on page 233
- "XMLDOCUMENT scalar function" in *SQL Reference, Volume 1*
- "XMLELEMENT scalar function" in *SQL Reference, Volume 1*
- "XMLFOREST scalar function" in *SQL Reference, Volume 1*

- “XMLNAMESPACES declaration” in *SQL Reference, Volume 1*
- “XMLAGG aggregate function” in *SQL Reference, Volume 1*
- “XMLATTRIBUTES scalar function” in *SQL Reference, Volume 1*
- “XMLCOMMENT scalar function” in *SQL Reference, Volume 1*
- “XMLCONCAT scalar function” in *SQL Reference, Volume 1*
- “XMLPI scalar function” in *SQL Reference, Volume 1*
- “XMLTEXT scalar function” in *SQL Reference, Volume 1*

Special character handling in SQL/XML publishing functions

SQL/XML publishing functions have default behavior for handling special characters.

SQL values to XML values:

Certain characters are considered special characters within XML documents, and must appear in their escaped format, using their entity representation. These special characters are as follows:

Table 16. Special characters and their entity representations

Special character	Entity representation
<	<
>	>
&	&
"	"

When publishing SQL values as XML values using the SQL/XML publishing functions, these special characters are escaped and replaced with their predefined entities.

SQL identifiers and QNames:

When publishing or constructing XML values from SQL values, it can be necessary 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 permitted in a QName. This difference means that some characters used in SQL identifiers will not be valid in QNames. These characters are therefore substituted with their entity representation in the QName.

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@work.

Note that 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. Refer to the W3C XML namespace specifications for more details on valid names.

Related concepts:

- “XML data type” on page 17

- “Native XML data store overview” on page 1
- “Publishing XML values with SQL/XML” on page 81

Related reference:

- Chapter 9, “Restrictions on native XML data store,” on page 233
- “Identifiers” in *SQL Reference, Volume 1*
- “QName data type (DB2 XQuery)” in *IBM DB2 XQuery Reference*
- “XMLATTRIBUTES scalar function” in *SQL Reference, Volume 1*
- “XMLELEMENT scalar function” in *SQL Reference, Volume 1*
- “XMLFOREST scalar function” in *SQL Reference, Volume 1*

XML serialization

XML serialization is the process of converting XML data from its representation in the XQuery and XPath data model, which is the hierarchical format it has in a DB2 database, to the serialized string format that it has in an application.

You can let the DB2 database manager perform serialization implicitly, or you can invoke the XMLSERIALIZE function to explicitly request XML serialization. The most common usage of XML serialization is when XML data is sent from the database server to the client.

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 CLI 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 DB2 Driver for JDBC and SQLJ adds an XML declaration.

Implicit serialization is the preferred method in most cases because it is simpler to code, and sending XML data to the client allows the DB2 client to handle the XML data properly. Explicit serialization requires additional handling, as described below, which is automatically handled by the client during implicit serialization.

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 character or binary data type.
- Whether the output data should include the following explicit encoding specification (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 serialized 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

most likely will 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 because it is more efficient to send data to the client as XML data. 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 should 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 one of the following data types before sending the data to the client:
 - The BLOB data type, by default
 - The CLOB data type, if you use the db2set command to set the DB2_MAP_XML_AS_CLOB_FOR_DLC registry variable to YES on the server

If you want the retrieved data to be some other data type, you can execute XMLSERIALIZE.

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, implicitly serialize the customerinfo document for customer ID '1000' and retrieve the serialized document into a binary XML host variable. 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;
```

Related concepts:

- “Application programming language support for XML” on page 165

- “Differences in an XML document after storage and retrieval” on page 88
- “Encoding considerations for retrieval of XML data from a database” on page 219
- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “Native XML data store overview” on page 1
- “Retrieval of data in XML columns to earlier DB2 clients” on page 80
- “XML data encoding” on page 217
- “XML data type” on page 17
- “XQuery and XPath data model (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Related reference:

- “Native XML data store roadmap” in *DB2 Information Center*
- “XMLSERIALIZE scalar function” in *SQL Reference, Volume 1*
- “Miscellaneous variables” in *Performance Guide*
- Chapter 9, “Restrictions on native XML data store,” on page 233

Differences in an XML document after storage and retrieval

When you store a copy of an XML document in a DB2 database and then retrieve that copy from the database, the retrieved document might not be exactly the same as the original document. This behavior is defined by the XML and SQL/XML standard, and is the same as the behavior of the Xerces open source XML parser.

Some of the changes to the document occur when the document is stored. Those changes are:

- If you execute XMLVALIDATE, the database server:
 - Adds default values and type annotations from the XML schema that is specified in the XMLVALIDATE invocation to the input document
 - Strips ignorable whitespace from the input document
- If you do not request XML 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 CLI 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 DB2 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 their predefined XML entities. Those characters and their predefined entities are:

Character	Unicode value	Entity representation
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 numeric representations. Those characters and their numeric representations are:

Character	Unicode value	Entity representation
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 ".
- 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:

- “Application programming language support for XML” on page 165
- “Insertion into XML columns” on page 32
- “Updates of XML columns” on page 34
- “XML data encoding” on page 217
- “XML data querying” on page 45
- “XML serialization” on page 86
- “XML validation” on page 41

Related reference:

- “XMLSERIALIZE scalar function” in *SQL Reference, Volume 1*
- “XMLVALIDATE scalar function” in *SQL Reference, Volume 1*

Data types for archiving XML documents

Although you can store XML serialized string data in a column of any binary or character type, non-XML columns should be used only for archiving XML data. The best column data type for archiving XML data is a binary data type, such as BLOB. Use of a character column for archiving introduces code page conversion, which can make a document inconsistent with its original form.

Related concepts:

- “XML data encoding” on page 217
- “XML data type” on page 17
- “XML serialization” on page 86

Related reference:

- “Data types” in *SQL Reference, Volume 1*

Tools that support XML

Various DB2 tools provide support for working with the XML data type.

Developer Workbench: Support for XML includes the following:

- **Stored procedure support:** You can create and run stored procedures that contain XML data types as input or output parameters.
- **Data output support:** You can view documents contained in XML columns as a tree or text.
- **SQL builder support:** You can build SQL expressions with XML functions and run SQL statements that contain XML host variables.
- **XML schema support:** You can manage schema documents in the XML schema repository (XSR), including registering and dropping schemas, as well as editing schema documents.
- **XML document validation support:** You can perform validation of XML documents against schemas registered in the XSR.
- **XQuery builder:** You can build the XQuery statements visually by dragging and dropping nodes that represent elements in an XML schema or document. For each node, you can specify predicates, expressions, clauses, and sorting preferences. XQuery builder then generates the query for you. Alternatively, you can write your own statements or modify the generated statements directly in the provided editing environment. After you have created your query, you can run it directly from the Developer Workbench to test it.

DB2 Control Center: The DB2 Control Center supports the native XML data type for many of its administrative functions. This allows database administrators to work with XML data alongside relational data from within a single GUI tool.

Examples of supported administrative tasks are:

- Creating tables with XML columns
- Creating indexes over XML columns using the new Create Index wizard
- Viewing the contents of XML documents stored in XML columns
- Working with the XML schemas, DTDs, and external entities required to validate and process XML documents
- Collecting statistics on tables containing XML columns

Command line processor: Several DB2 commands support the native storage of XML data. You can work with XML data alongside relational data from the DB2 command line processor (CLP). Examples of tasks that you can perform from the CLP include:

- Issuing XQuery statements by prefixing them with the XQUERY keyword.
- Importing and export XML data.
- Collecting statistics on XML columns.

- Calling stored procedures with IN, OUT, or INOUT parameters of XML data type.
- Working with the XML schemas, DTDs, and external entities required to validate and process XML documents.
- Reorganizing indexes over XML data and tables containing XML columns.
- Decomposing XML documents.

DB2 Development Add-In for Microsoft Visual Studio .NET: You can use the Development Add-In for Microsoft Visual Studio .NET to create tables with XML columns and indexes over XML data. You create an XML column as you would any other column in this tool. Simply specify the data type as XML. You can create an index by using the XML Index Designer in this tool. You do not have to manually specify the XML pattern expression as required by the CREATE INDEX syntax for an index over XML data. Instead, you can graphically select the XML nodes that you want to index from either a tree representation of a registered XML schema, a document from the XML column, or an XML schema in a local file. The tool generates the XML pattern expression for you. Alternatively, you can manually specify the XML pattern expression. After you have specified all of the other index attributes, the tool generates the index for you.

EXPLAIN: You can issue the EXPLAIN statement on XQuery statements and SQL/XML statements to quickly see the access plan for these statements, including whether DB2 uses indexes. To issue the EXPLAIN statement for an XQuery statement, use the XQuery keyword followed by an XQuery statement that is enclosed in single or double quotation marks, as in the following example:

```
EXPLAIN PLAN SELECTION FOR XQUERY 'for $c in
db2-fn:xmlcolumn("XISCAN TABLE.XMLCOL" )/a[@x="1"]/b[@y="2"] return $c'
```

DB2 captures the access plan information in the EXPLAIN tables. The expected sequence size for any XML columns are stored in the SEQUENCE_SIZES column of the EXPLAIN_STREAM table. You might also notice data in the EXPLAIN_PREDICATE table for several predicates that you do not recognize. These predicates are generated by DB2 during the EXPLAIN operation to evaluate XPath expressions that are used in an index scan. You do not need to evaluate this predicate information. These predicates not part of the optimizer plan and thus have a value of -1 in the PREDICATE_ID and FILTER_FACTOR columns.

Alternatively, you can avoid manually interpreting the EXPLAIN tables by using the Visual Explain tool to view graphical depictions of these access plans. The following nodes are displayed in the graphs to show XML operations:

XISCAN

Indicates that the DB2 database server used an index over XML data to access the data.

XSCAN

Indicates that the DB2 database server scanned XML documents in an XML column.

XANDOR

Indicates that the DB2 database server applied the AND and OR predicates to the results of multiple index scans.

Related concepts:

- “Control Center overview” in *Administration Guide: Implementation*
- “Visual Explain overview” in *Administration Guide: Implementation*

- “Native XML data store overview” on page 1
- “DB2 Command Line Processor (CLP)” in *Developing SQL and External Routines*
- “DB2 integration in Visual Studio” in *Developing ADO.NET and OLE DB Applications*
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133

Related reference:

- “Native XML data store roadmap” in *DB2 Information Center*
- “EXPLAIN statement” in *SQL Reference, Volume 2*
- “IXSCAN operator” in *Performance Guide*
- “XANDOR operator” in *Performance Guide*
- “XISCAN operator” in *Performance Guide*
- Chapter 9, “Restrictions on native XML data store,” on page 233

Chapter 4. XML data indexing

Overview

XML data indexing overview

An index over XML data 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 you specify, an index over XML data 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 index over XML data 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, multiple index keys may be inserted into the index for a single document.

You create an index over XML data using the CREATE INDEX statement, and drop an index over XML data 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. The UNIQUE keyword also has a different meaning for indexes over XML data.

Example: Creating an index over XML data: Suppose that table companyinfo has an XML column named companydocs, which contains XML document fragments like these:

Document for Company1:

```
<company name="Company1">
  <emp id="31201" salary="60000" gender="Female">
    <name>
      <first>Laura</first>
      <last>Brown</last>
    </name>
    <dept id="M25">
      Finance
    </dept>
  </emp>
</company>
```

Document for Company2:

```
<company name="Company2">
  <emp id="31664" salary="60000" gender="Male">
    <name>
      <first>Chris</first>
      <last>Murphy</last>
    </name>
    <dept id="M55">
      Marketing
    </dept>
  </emp>
</company>
```

```

    </dept>
  </emp>
  <emp id="42366" salary="50000" gender="Female">
    <name>
      <first>Nicole</first>
      <last>Murphy</last>
    </name>
    <dept id="K55">
      Sales
    </dept>
  </emp>
</company>

```

Users of the companyinfo table often retrieve employee information using the employee ID. You might use an index like this one to make that retrieval more efficient:

```

CREATE INDEX empindex on companyinfo(companydocs)
  GENERATE KEY USING XMLPATTERN '/company/emp/@id'
  AS SQL DOUBLE

```

Figure 5. Example of an index over XML data

Notes to Figure 5:

- 1** The index over XML data is defined on the companydocs column of the companyinfo table. companydocs 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 id attribute of each employee element.
- 3** AS SQL DOUBLE indicates that indexed values are stored as DOUBLE values.

Related concepts:

- “Index XML pattern expressions” on page 94
- “UNIQUE keyword semantics” on page 105

Related reference:

- “CREATE INDEX ” on page 108

Index XML pattern expressions

Only those parts of an XML document stored in an XML column that satisfy an XML pattern expression are indexed. To index on an XML pattern, you provide an index specification clause together with 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 index over XML data. Alternatively, you can specify the clause GENERATE KEYS USING XMLPATTERN.

Only one index specification clause is allowed per CREATE INDEX statement. Multiple XML indexes may be created on an XML column in succession.

XML pattern expressions:

To identify those parts of the document that will be indexed, an XML pattern is used to specify a set of nodes within the XML document. This pattern expression is

similar to the path expression defined in the XQuery language, but it differs in that only a subset of the XQuery language is supported.

Path expression steps are separated by the forward slash (/). The double forward slash (//) which is the abbreviated syntax for /descendant-or-self::node()/ may also be specified. In each step, a forward axis (child::, @, attribute::, descendant::, self::, and descendant-or-self::) is chosen, followed by an XML name test or XML kind test. If no forward axis is specified, the child axis is used as the default.

If the XML name test is used, a qualified XML name or a wildcard is used to specify the node name to match for the step in the path. Instead of matching node names, the XML kind test can also be used to specify what kind of nodes to match in the pattern: text nodes, comment nodes, processing instruction nodes, or any other type of node.

Below are some examples of different pattern expressions.

1. CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@id' AS SQL DOUBLE
2. CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/child::company/child::emp/attribute::id'
AS SQL DOUBLE
3. CREATE INDEX idindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '//@id' AS SQL DOUBLE
4. CREATE INDEX idindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/descendant-or-self::node()/attribute::id'
AS SQL DOUBLE
5. CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/name/last/text()' AS SQL
VARCHAR(25)

Note: Statement 1 and statement 2 are logically equivalent. Statement 2 uses the unabbreviated syntax. Statement 3 and statement 4 are logically equivalent. Statement 4 uses the unabbreviated syntax.

Qualifying Paths and Nodes:

Consider a table named "company" with XML documents stored in an XML column (companydocs). The XML documents have a hierarchy with the two paths: '/company/emp/dept/@id' and '/company/emp/@id'. If the XML pattern specifies a single path, then a set of nodes in the document may qualify.

For example, if users wished to search for a specific employee id attribute (@id) on the employee elements, they could create an index on the XML pattern '/company/emp/@id'. Then queries with predicates of the form '/company/emp[@id=42366]' could utilize the index on an XML column. In this case, the XMLPATTERN '/company/emp/@id' in the CREATE INDEX statement specifies a single path that refers to many different nodes in the document, since every employee element in the document may have an employee id attribute.

```
CREATE INDEX empindex on company(companydocs)  
GENERATE KEY USING XMLPATTERN '/company/emp/@id' AS SQL DOUBLE
```

If the XML pattern uses wildcard expressions, the descendant axis, or the descendant-or-self axis, then a set of paths and nodes may qualify. In the following example, the descendant-or-self axis is specified, so that the XML pattern '//@id' references paths for both department id attributes and employee id attributes since they both contain @id.

```
CREATE INDEX idindex on company(companydocs)
    GENERATE KEYS USING XMLPATTERN '//@id' AS SQL DOUBLE
```

Related reference:

- “CREATE INDEX ” on page 108

XML namespace declarations

Qualified XML names (QNames) are used to define the element and attribute tags in XML pattern expressions. The qualifier for a QName is a namespace prefix which has been associated with a namespace URI.

The XML pattern can be specified using an optional namespace declaration to map a namespace prefix to a namespace URI string literal or to define a default namespace URI for the XML pattern. The namespace prefix is then used to qualify names of elements and attributes in the XML pattern to match them with the namespace used in the document.

In the example below, the shorthand namespace prefix *m* is mapped to `http://www.mycompanyname.com/`

```
CREATE INDEX empindex on department(deptdocs)
    GENERATE KEYS USING XMLPATTERN
    'declare namespace m="http://www.mycompanyname.com/";
    /m:company/m:emp/m:name/m:last' AS SQL VARCHAR(30)
```

Note that if this CREATE INDEX statement is issued from the command line processor (CLP), the embedded semicolon becomes problematic, because the semicolon is the default statement terminator. To avoid this issue, use one of the following workarounds:

- Ensure the semicolon is not the last non whitespace character on the line (by adding an empty XQuery comment after the semicolon, for example).
- Change the default statement terminator in the CLP from the command line.

Multiple namespace declarations can also be specified in the same XMLPATTERN expression, but the namespace prefix must be unique within the list of namespace declarations. In addition, the user has the option to declare a default namespace for elements that do not have a prefix. If a namespace or namespace prefix is not explicitly specified for an element, then the default namespace will be used. Default namespace declarations do not apply to attributes. If the user does not specify a default namespace, then the namespace will be *no namespace*. Only one default namespace can be declared. This namespace declaration behavior follows XQuery rules.

The previous example can also be written using a default namespace:

```
CREATE INDEX empindex on department(deptdocs)
    GENERATE KEY USING XMLPATTERN
    'declare default element namespace "http://www.mycompany.com/";
    /company/emp/name/last') AS SQL VARCHAR(30)
```

In the next example, the *@id* attribute has the *no namespace* namespace, since the default namespace `http://www.mycompany.com/` applies only to the *company* and *emp* elements, but not the *@id* attribute. This follows basic XQuery rules, since in an XML document default namespace declarations do not apply to attributes.

```
CREATE INDEX empindex on department(deptdocs)
GENERATE KEY USING XMLPATTERN
'declare default element namespace "http://www.mycompany.com/";
/company/emp/@id' AS SQL VARCHAR(30)
```

Since the *@id* attribute should have the same namespace as the *company* and *emp* elements, the statement could be rewritten as:

```
CREATE INDEX empindex on department(deptdocs)
GENERATE KEY USING XMLPATTERN
'declare default element namespace "http://www.mycompany.com/";
declare namespace m="http://www.mycompanyname.com/";
/company/emp/@m:id' AS SQL VARCHAR(30)
```

The namespace prefix used to create the index and the namespace prefix used in the instance documents do not need to match to be indexed, but the fully expanded QName does need to match. The value of the namespace to which the prefix expands is important, not the prefix name itself. For example, if the namespace prefix for the index is defined as *m*="http://www.mycompany.com/" and the namespace prefix used in the instance document is *c*="http://www.mycompany.com/", then *c:company/c:emp/@id* in the instance documents would be indexed, since both shorthand namespace prefixes *m* and *c* expand to the same namespace.

Related concepts:

- “Index XML pattern expressions” on page 94

Related reference:

- “QName data type (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Data types associated with index XML pattern expressions

Every XML pattern expression specified in the CREATE INDEX statement must be associated with a data type. Four SQL data types are supported: VARCHAR, DATE, TIMESTAMP, and DOUBLE.

You can choose to interpret the result of the expression as multiple data types. For example, the value 123 has a character representation but it can also be interpreted as the number 123. If you would like to index the path */company/emp/@id* as both a character string and as a numeric string, then two indexes must be created, one for the VARCHAR data type and one for the DOUBLE data type. The values in the document are converted to the specified data type for each index.

The following example shows how to create two indexes with different data types on the same XML column *deptdocs*:

```
CREATE INDEX empindex1 on department(deptdocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@id' AS SQL VARCHAR(10)
CREATE INDEX empindex2 on department(deptdocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@id' AS SQL DOUBLE
```

Description of supported SQL data types:

VARCHAR

VARCHAR data will be stored in the index on an XML column in the UTF-8 code page.

VARCHAR(integer)

If the data type VARCHAR is used with the specified length *integer* (in

bytes), the specified length is treated as a constraint. If documents are inserted into the table or exist in the table at the same time the index is created, then the document insertion or index creation will fail if there are nodes to be indexed with values longer than the specified length. If the insertion or creation succeeds, the index is guaranteed to store all character string values in their entirety and it can support both range scans and equality lookups. The length *integer* is a value in the range from 1 to the page size dependent maximum. See the CREATE INDEX statement for the list of maximum allowed lengths. XQuery semantics are used for string comparisons, where trailing blanks are significant. This differs from SQL semantics, where trailing blanks are insignificant during comparisons.

```
CREATE INDEX empindex1 on department(deptdocs)
  GENERATE KEY USING XMLPATTERN '/company/emp/@id' AS SQL VARCHAR(50)
```

VARCHAR HASHED

VARCHAR HASHED may be specified to handle indexing of character strings with arbitrary lengths. If documents contain character strings to be indexed that exceed the maximum length *integer* allowed for the index based on the page size-dependent maximum, then you can specify VARCHAR HASHED instead. In this case, the system generates an 8 byte hash code over the entire string and there is no limit on the length of the indexed string. Range scans cannot be performed if you specify VARCHAR HASHED, since the index contains hash codes instead of the actual character data. Indexes using these hashed character strings may be used only for equality lookups. XQuery semantics are used for string equality comparisons, where trailing blanks are significant. This differs from SQL semantics, where trailing blanks are insignificant during comparisons. The hash on the string preserves XQuery semantics for equality, and not SQL semantics for equality.

```
CREATE INDEX empindex on company(companydocs)
  GENERATE KEY USING XMLPATTERN '/company/emp/name/last' AS SQL
  VARCHAR HASHED
```

DOUBLE

All numeric values will be converted and stored in the index as the DOUBLE data type. Unbounded decimal types and 64 bit integers may lose precision when they are stored as a DOUBLE. The values for the index SQL data type DOUBLE may include the special numeric values *NaN*, *INF*, *-INF*, *+0*, and *-0*, even though the SQL data type DOUBLE itself does not support these values.

DATE

DATE data type values will be normalized to UTC (Coordinated Universal Time) or Zulu time before being stored in the index. Note that the XML schema data type for DATE allows greater precision than the SQL data type. If an out-of-range value is encountered, an error is returned.

TIMESTAMP

TIMESTAMP data type values will be normalized to UTC (Coordinated Universal Time) or Zulu time before being stored in the index. Note that the XML schema data type for timestamps allows greater precision than the SQL data type. If an out-of range value is encountered, an error is returned.

Related concepts:

- “Indexing elements with complex schema types” on page 99
- “Data type conversion for indexes over XML data” on page 102
- “Index XML pattern expressions” on page 94

Related reference:

- “CREATE INDEX ” on page 108
- “XMLCAST specifications” in *SQL Reference, Volume 1*

Indexing elements with complex schema types

Complex types:

This section is applicable only to documents that have been validated against a schema.

An element with a simple schema type may have only a single text node child. No attributes or element children are allowed. In contrast, complex types in schemas may have attributes and can have 4 different content values based on the presence or absence of text nodes and child elements:

Table 17. Content values for complex types in XML schemas

	Child element (yes)	Child element (no)
Text node (yes)	Mixed content	Simple content
Text node (no)	Complex content	Empty content

The semantics for obtaining the value to index varies, depending on the data type used. Indexes using the VARCHAR data type are defined by the fn:string function and not xs:string. Because fn:string will always return a result, a zero-length string value is indexed for empty content and the element’s text node is indexed for simple content. For complex content and for mixed content, the text nodes of any descendant elements are concatenated without any additional blanks in between.

In contrast, indexes using the data types DOUBLE, DATE, and TIMESTAMP follow the semantics for xs:double, xs:date, and xs:dateTime respectively. These data types must atomize the value, but values with complex content cannot be atomized. As a result, complex content cases do not have a valid value to index. Values with empty content also do not have a valid value to index. Consequently, if an element matches the index pattern but it is a complex type with empty content or a complex type with complex content, then the element will be ignored and not indexed. No error or warning will be issued.

The semantics for the simple and mixed content cases are similar for all the data types, except that the resulting value must be converted to the correct data type for indexes using DOUBLE, DATE, or TIMESTAMP in order for the value to be indexed. For simple content, the element’s text node is indexed if the node’s typed value can be converted correctly to the index data type. For mixed content, the typed value of the element node is an untypedAtomic string that is the concatenation of the child text nodes with the text nodes of any descendant elements. However, the resulting untypedAtomic string can only be indexed if it can be converted to the index data type.

Content for indexes using the VARCHAR data type:

Table 18. Content for indexes using the VARCHAR data type

Content value	Indexing result
Empty	The content is indexed with a zero-length string value.

Table 18. Content for indexes using the VARCHAR data type (continued)

Content value	Indexing result
Simple	The node is indexed with a string value that is the concatenation of all descendant text nodes.
Complex	The node is indexed with a string value that is the concatenation of all descendant text nodes.
Mixed	The node is indexed with a string value that is the concatenation of all descendant text nodes.

Content for indexes using the data types DOUBLE, DATE, and TIMESTAMP:

Table 19. Content for indexes using the data types DOUBLE, DATE, and TIMESTAMP

Content value	Indexing result
Empty	The node is ignored and not indexed.
Simple	The content is indexed, if the node's typed value can be converted correctly to the index data type.
Complex	The node is ignored and not indexed.
Mixed	The node is indexed if the node's typed value (that is, the concatenation of all descendant text nodes) can be cast correctly to the index data type.

In the following example, the XML schema defines the element <top> as the complex type *n1:topType*. This schema defines a simple type and complex types for each of the 4 different content types. The element names have been chosen to reflect the type and content they represent. The document conforms to the schema definition. An index using the VARCHAR data type and an index using the DOUBLE data type are both created on the xmlpattern */** in order to match all elements in the document. The tables below show how the index entries may differ between the two indexes.

Sample XML schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://foo.com" xmlns:n1="http://foo.com">
  <xsd:element name="top" type="n1:topType"/>

  <xsd:complexType name="complexEmptyType">
  </xsd:complexType>
  <xsd:complexType name="complexSimpleType">
  <xsd:simpleContent>
  <xsd:extension base="xsd:integer">
  <xsd:attribute name="attr" type="xsd:integer"/>
  </xsd:extension>
  </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="complexComplexType">
  <xsd:sequence>
  <xsd:element name="complexchild" type="xsd:integer"/>
  </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="complexMixedType1" mixed="true">
```

```

    <xsd:sequence>
      <xsd:element name="mixedchild1" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
<xsd:complexType name="complexMixedType2" mixed="true">
  <xsd:sequence>
    <xsd:element name="mixedchild2" type="xsd:integer"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="topType">
  <xsd:sequence>
    <xsd:element name="simple" type="xsd:integer"/>
    <xsd:element name="complexempty" type="n1:complexType"/>
    <xsd:element name="complexsimple" type="n1:complexType"/>
    <xsd:element name="complexcomplex" type="n1:complexType"/>
    <xsd:element name="complexmixed1" type="n1:complexType1"/>
    <xsd:element name="complexmixed2" type="n1:complexType2"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

Sample XML document::

```

<?xml version="1.0"?>
<x:top xmlns:x="http://foo.com">
  <simple>1</simple>
  <complexempty/>
  <complexsimple attr="5">2</complexsimple>
  <complexcomplex><complexchild>3</complexchild></complexcomplex>
  <complexmixed1>hello<mixedchild1>4</mixedchild1></complexmixed1>
  <complexmixed2>5<mixedchild2>6</mixedchild2></complexmixed2>
</x:top>

```

Sample indexes::

```

CREATE INDEX IXVARCHAR ON T1(XMLDOC)
  GENERATE KEY USING XMLPATTERN '//*' AS SQL VARCHAR(20);

```

```

CREATE INDEX IXDOUBLE ON T1(XMLDOC)
  GENERATE KEY USING XMLPATTERN '//*' AS SQL DOUBLE;

```

Index IXVARCHAR (VARCHAR):

Table 20.

Indexed Element	Indexed value
<top>	123hello456 (concatenated text)
<simple>	1
<complexempty>	zero length string
<complexsimple>	2
<complexchild>	3
<complexcomplex>	3 (concatenated with text from <complexchild>)
<complexmixed1>	hello4 (hello from <complexmixed1> concatenated with 4 from <mixedchild1>)
<mixedchild1>	4
<complexmixed2>	56 (5 from <complexmixed2> concatenated with 6 from <mixedchild2>)
<mixedchild2>	6

Index IXDOUBLE (DOUBLE):

Table 21.

Indexed Element	Indexed value
<top>	Ignored (complex type with complex content)
<simple>	1.000000e+00
<complexempty>	Ignored (complex type with empty content)
<complexsimple>	2.000000e+00
<complexchild>	3.000000e+00
<complexcomplex>	Ignored (complex type with complex content)
<complexmixed1>	Ignored (<i>hello</i> from <complexmixed1> concatenated with <i>4</i> from <mixedchild1> : <i>hello4</i> is an invalid DOUBLE value)
<mixedchild1>	4.000000e+00
<complexmixed2>	5.600000e+01 (<i>5</i> from <complexmixed2> concatenated with <i>6</i> from <mixedchild2> : <i>56</i> converts successfully to DOUBLE)
<mixedchild2>	6.000000e+00

Related concepts:

- “Data types associated with index XML pattern expressions” on page 97
- “XML data indexing overview” on page 93

Data type conversion for indexes over XML data

Before values can be inserted into the index over XML data, they are first converted to the index XML type corresponding to the index SQL data type. For VARCHAR(*integer*) and VARCHAR HASHED, the value is converted to an xs:string value using the XQuery function fn:string. The length attribute of VARCHAR(*integer*) is applied as a constraint to the resulting xs:string value. An index SQL data type of VARCHAR HASHED applies a hash algorithm to the resulting xs:string value to generate a hash code that is inserted into the index. Data for VARCHAR types is stored directly in the index without first being normalized to the schema data type.

For DOUBLE, DATE, and TIMESTAMP indexes, the value is converted to the index XML type using the XQuery cast expression. DATE and TIMESTAMP data type values will be normalized to UTC (Coordinated Universal Time) or Zulu time before being stored in the index. XML data that is valid according to XQuery rules that cannot be converted to the index data type due to system limitations will result in an indexing error (decimal values with greater precision than supported, for example). The values for the index SQL data type DOUBLE may include the special numeric values *NaN*, *INF*, *-INF*, *+0*, and *-0*, even though the SQL data type DOUBLE itself does not support these values.

Corresponding index data types:

Table 22. Corresponding index data types

XML data type	SQL data type
xs:string	VARCHAR(<i>integer</i>) and VARCHAR HASHED
xs:double	DOUBLE
xs:date	DATE
xs:dateTime	TIMESTAMP

Conversion to non VARCHAR XML index data types without XML schema:

If no XML schema is present, then the document data is untyped and values are not validated by the parser. The source value is converted to the target index XML type with special numeric values handled for the target index SQL type if necessary.

Conversion to the non VARCHAR XML index data types with an XML schema:

If an XML schema is present, the structure of the input documents is first validated by the parser. Data types of the elements and attributes are constrained to the schema's specification. If the document contains values that don't match the schema's specification, the parser will reject the document. For example, if the schema specifies xs:float and the value is *Laura*, then the document will be rejected.

If the parser validates a document source value against the schema successfully, then the following steps occur:

1. The value is converted to the DB2 binary representation for the schema data type.
2. The value is converted to the DB2 binary representation for the index XML data type.
3. Special XML values are handled for the target index SQL data type, if necessary.

Index entries are always inserted after schema validation has completed and the value has been converted to the index data type. For example, if the schema validates an input value *12* to have the type annotation of xs:string and the index was created with a DOUBLE data type, then the conversion will succeed and the value *12* will be inserted into the index. The insertion succeeds even though the value's xs:string data type does not match the index's DOUBLE data type, because the value converts successfully to the index's DOUBLE data type. However, if the schema validates an input value *ABC* to have the type annotation of xs:string and the index was created with a DOUBLE data type, then the conversion will fail and the value of *ABC* will not be inserted into the index.

Related concepts:

- "Data types associated with index XML pattern expressions" on page 97
- "Guidelines for matching indexes with queries overview" on page 72
- "XML data indexing overview" on page 93

Related reference:

- "Limits for XQuery data types" in *IBM DB2 XQuery Reference*

- “SQL and XQuery limits” in *SQL Reference, Volume 1*

XML schemas and index key generation

If an XML schema is used, then the structure of XML documents to be stored in an XML column will be validated so that the data types of the elements and attributes in the XML documents are constrained against the XML schema. If a document does not match the schema’s specifications, then the document is rejected by the parser. For example, if the schema specifies that an attribute is constrained to the DOUBLE data type and the value of the document’s attribute is ABC, then the document is rejected. If an XML schema is not used, then document data is not validated by the parser and is considered to be untyped data.

You should examine your XML schema, so that you can create indexes on an XML column with data types that match your XML schema’s data type specifications. The queries that you want to run should also be taken into account when deciding which XML patterns to choose for your indexes.

For example, suppose the following XML schema is used:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="product" type="ProdType"/>
  <xsd:simpleType name="ColorType">
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value='20' />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="ProdType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="SKU" type="xsd:string" />
      <xsd:element name="price" type="xsd:integer" />
      <xsd:element name="comment" type="xsd:string" />
    </xsd:sequence>
    <xsd:attribute name="color" type="ColorType" />
    <xsd:attribute name="weight" type="xsd:integer" />
  </xsd:complexType>
</xsd:schema>
```

After looking at the queries you need to issue, you may decide they need indexes on price and color. Analyzing the queries will help you decide what XML pattern expressions to include in your CREATE INDEX statement. The XML schema provides guidance on what data type to pick for the index: you can tell that the price element is an integer so the numeric data type of DOUBLE can be chosen for the index *priceindex* and the color attribute is a string so the data type of VARCHAR can be chosen for the index *colorindex*.

```
db2-fn:xmlcolumn('COMPANY.PRODUCTDOCS')/product[price > 5.00]
db2-fn:xmlcolumn('COMPANY.PRODUCTDOCS')/product[@color = 'pink']
CREATE INDEX priceindex on company(productdocs)
      GENERATE KEY USING XMLPATTERN '/product/price' AS DOUBLE
CREATE INDEX colorindex on company(productdocs)
      GENERATE KEY USING XMLPATTERN '//@color' AS SQL VARCHAR(80)
```

The schema may also specify other constraints for the string data type, such as *maxLength* which is shown in the example under *ColorType* where the string is restricted to 20 unicode characters. Since the CREATE INDEX statement specifies the VARCHAR length in bytes and not in characters, the schema length may be

multiplied by a factor of 4 to calculate the maximum number of bytes that will be required to store the longest string allowed by the schema in the index. In this case, $4 * 20 = 80$ so VARCHAR(80) is chosen for *colorindex*.

If the schema does not specify a length restriction for a string data type and the maximum string lengths for the values in the documents are not known, then you can use the maximum length allowed by the page size used by the index. An index stores strings of varying lengths, but since only the actual number of bytes required for each string is stored there is no storage penalty for specifying a longer maximum length than is needed. However, larger key buffers in memory do need to be allocated to handle the maximum key size during index scans. See the CREATE INDEX statement for the list of maximum allowed lengths for an index on an XML column specifying the VARCHAR data type.

If the maximum length for the VARCHAR data type is not sufficiently long to index the your document values, then the VARCHAR HASHED data type may be used, which does not have a length limit. However, indexes using VARCHAR HASHED can only be used for equality look-ups and not for range scans. Note that documents which contain strings longer than the length specified for VARCHAR(*integer*) will be rejected.

The XML schema may also specify default attribute and element values. If there are no corresponding values specified in the XML document and the document is validated, then the default values from the schema are used when the document is stored. These default values will be indexed, along with the other values that were in the original input document. If the document is not validated, then the default values are not added to document and they are not indexed.

Related concepts:

- “Data types associated with index XML pattern expressions” on page 97

Related reference:

- “CREATE INDEX ” on page 108

UNIQUE keyword semantics

The same UNIQUE keyword that is used for indexes on non-XML columns is also used for indexes on XML columns, but it has a different meaning. 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 nodes that are qualified by the XML pattern. A single document may have multiple nodes that are inserted into a unique index; their values must be unique in that document and in all other documents. Note also that some documents may not have any nodes that are added to an index; uniqueness is not enforced for these documents.

Uniqueness is enforced for the data type of the index, the XML path to the node, and the value of the node after the XML value has been converted to the SQL data type that is specified for the index.

Caution should be used when specifying the UNIQUE keyword. Because converting to the specified data type for the index might result in a loss of precision or range, or different values might be hashed to the same key value, multiple values that appear to be unique in the XML document might result in duplicate key errors. Situations under which duplicate key errors can occur are:

- When VARCHAR HASHED is specified, unique character strings may hash to the same hash code and result in a duplicate key error.
- For numerical values, a loss of precision or values that are beyond the range of the DOUBLE data type can cause duplicate key errors during insertion. For example, big integer and unbounded decimal values may lose precision when they are stored as the DOUBLE data type in the index.

If VARCHAR(*integer*) is specified, then the entire character string from the XML document is stored in the index so that incorrect duplicate key errors cannot occur. In addition, uniqueness of character strings follows XQuery semantics, where trailing blanks are significant. Therefore, values that would be duplicates in SQL but differ in trailing blanks are considered unique values in an index over XML data.

```
CREATE UNIQUE INDEX EMPINDEX ON company(companydocs)
  GENERATE KEY USING XMLPATTERN '/company/emp/name/last' AS SQL
  VARCHAR(100)
```

For UNIQUE indexes, the XML pattern must specify a single complete path and may not contain any of the following:

- a descendant axis
- a descendant-or-self axis
- a /descendant-or-self::node()/ (//)
- any wildcards for the XML name test
- a node() or processing instruction() for the XML kind test

Related concepts:

- “Data types associated with index XML pattern expressions” on page 97

Related reference:

- “CREATE INDEX ” on page 108

Database objects associated with XML data indexing

Logical and physical indexes over XML data

When you create an index over XML data, two B-tree indexes are created, a logical index and a physical index. The logical index contains the XML pattern information specified by the CREATE INDEX statement. The physical index has DB2 generated key columns to support the logical index and contains the indexed document values, converted to the data type specified in the *xmltype-clause* of the CREATE INDEX statement.

You work with an index over XML data at the logical level (with the CREATE INDEX and DROP INDEX statements, for example). Processing of the underlying physical index by DB2 is transparent to you. Note that the physical index is not recognized by any application programming interface that returns index metadata.

In the SYSCAT.INDEXES catalog view, the logical index has the index name you specified in the CREATE INDEX statement and the index type *XVIL*. The physical index has a system generated name and the index type *XVIP*. The logical index is always created and assigned an index ID (IID) first. The physical index is created immediately afterwards and is assigned the next consecutive index ID.

The relationship between logical and physical indexes is demonstrated in the following example: Consider two indexes over XML data, *EMPINDEX* and *IDINDEX*. For *EMPINDEX*, the logical index has the name *EMPINDEX*, the index ID 3, and the index type *XVIL*. The corresponding physical index has the system generated name *SQL060414134408390*, the index ID 4, and the index type *XVIP*.

Table 23. The relationship between logical and physical indexes

Index name (INDNAME)	Index ID (IID)	Table name (TABNAME)	Index type (INDEXTYPE)
SQL060414133259940	1	COMPANY	XRGN
SQL060414133300150	2	COMPANY	XPTH
EMPINDEX	3	COMPANY	XVIL
SQL060414134408390	4	COMPANY	XVIP
IDINDEX	5	COMPANY	XVIL
SQL060414134408620	6	COMPANY	XVIP

Catalog views:

Please refer to the Related reference section for more information on each of these catalog views.

SYSCAT.INDEXES

Each row represents an index, including logical and physical indexes over XML data.

SYSCAT.INDEXXMLPATTERNS

Each row represents a pattern clause in an index over XML data.

Auditing:

Indexes on XML columns use the existing index object type for audits. Only the logical index is audited, and not the physical index.

Related concepts:

- “Data types associated with index XML pattern expressions” on page 97
- “XML data indexing overview” on page 93

Related reference:

- “CREATE INDEX ” on page 108
- “SYSCAT.INDEXES catalog view” in *SQL Reference, Volume 1*
- “SYSCAT.INDEXXMLPATTERNS catalog view” in *SQL Reference, Volume 1*

Other database objects associated with XML columns

XML path index and XML regions index:

There are two internal and system generated indexes associated with XML columns. Whenever you create an XML column, an XML path index is automatically created by DB2 on the XML column. DB2 also creates a single XML regions index for all XML columns within a table.

The XML path index records all unique paths which exist within XML documents stored within an XML column.

The XML regions index captures how an XML document is divided up internally into regions, which are sets of nodes within a page. When an XML document is represented as nodes, each node is a record in a page. Since regions are sets of nodes within a page, the number of regions index entries can be reduced, and performance may be improved, if a larger page size that can store more nodes within a page is used.

Both the XML path and the XML regions indexes are recorded in SYSCAT.INDEXES. Note that these indexes are not recognized by any application programming interface that returns index metadata.

These internal indexes associated with XML columns are distinct from indexes you create over XML columns (also known as indexes over XML data). For indexing XML data as stored in XML columns, you work only with the logical indexes on XML columns only, using the CREATE INDEX and DROP INDEX statements, for example.

Catalog view:

SYSCAT.INDEXES

Each row represents an index, including XML path and XML regions indexes. The XML path index is shown as *XPTH* in SYSCAT.INDEXES.INDEXTYPE, and the XML regions index is shown as *XRGN* in SYSCAT.INDEXES.INDEXTYPE. Please refer to the Related reference section for more information on this catalog view.

Related concepts:

- “Logical and physical indexes over XML data” on page 106

Related reference:

- “CURRENT LOCK TIMEOUT special register” in *SQL Reference, Volume 1*
- “SYSCAT.INDEXES catalog view” in *SQL Reference, Volume 1*

CREATE INDEX

The CREATE INDEX statement is used to:

- Define an index on a DB2 table. An index can be defined on XML data, or on relational data.
- Create an index specification (metadata that indicates to the optimizer that a data source table has an index)

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

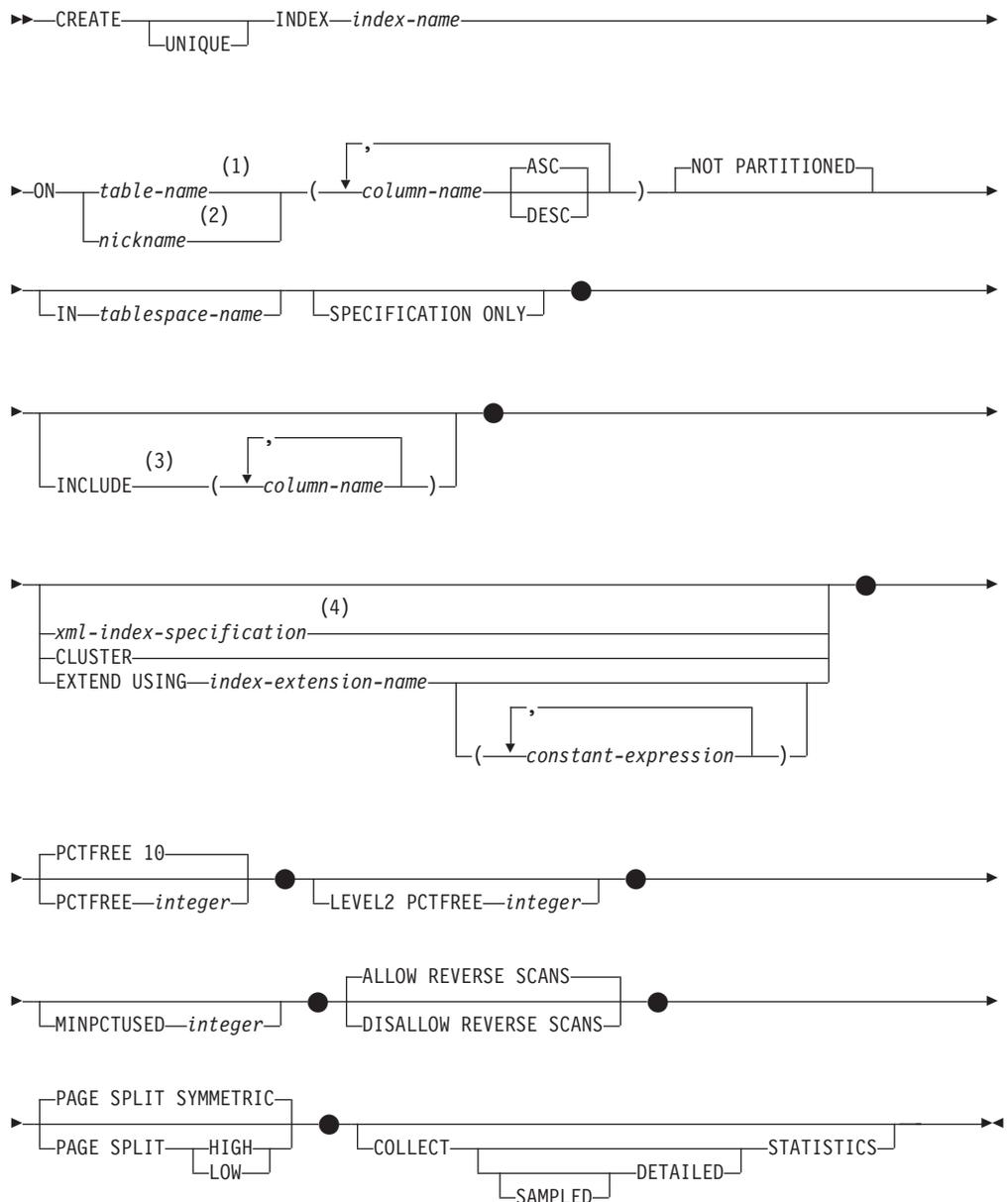
The privileges held by the authorization ID of the statement must include at least one of the following:

- One of:
 - CONTROL privilege on the table or nickname on which the index is defined

- INDEX privilege on the table or nickname on which the index is defined and one of:
 - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the index does not exist
 - CREATEIN privilege on the schema, if the schema name of the index refers to an existing schema
- SYSADM or DBADM authority

No explicit privilege is required to create an index on a declared temporary table.

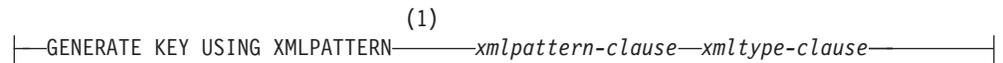
Syntax:



Notes:

- 1 In a federated system, *table-name* must identify a table in the federated database. It cannot identify a data source table.
- 2 If *nickname* is specified, the CREATE INDEX statement creates an index specification. In this case, INCLUDE, CLUSTER, EXTEND USING, PCTFREE, MINPCTUSED, DISALLOW REVERSE SCANS, ALLOW REVERSE SCANS, PAGE SPLIT, or COLLECT STATISTICS cannot be specified.
- 3 The INCLUDE clause can only be specified if UNIQUE is specified.
- 4 If *xml-index-specification* is specified, *column-name* DESC, INCLUDE, or CLUSTER cannot be specified.

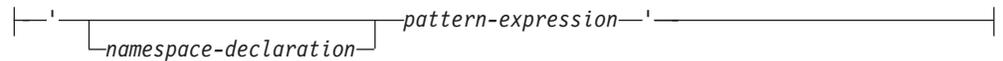
xml-index-specification:



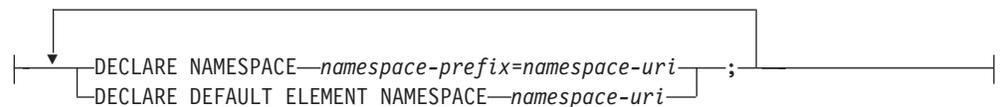
Notes:

- 1 The alternative syntax GENERATE KEYS USING XMLPATTERN can be used.

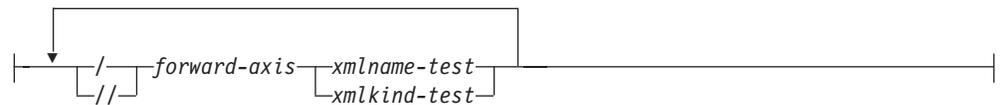
xmlpattern-clause:



namespace-declaration:



pattern-expression:



forward-axis:



blanks are significant. Therefore, values that would be duplicates in SQL but differ in trailing blanks are considered unique values in an index over XML data.

When UNIQUE is used, null values are treated as any other values. For example, if the key is a single column that may contain null values, that column may contain no more than one null value.

If the UNIQUE option is specified, and the table has a distribution key, the columns in the index key must be a superset of the distribution key. That is, the columns specified for a unique index key must include all the columns of the distribution key (SQLSTATE 42997).

Primary or unique keys cannot be subsets of dimensions (SQLSTATE 429BE).

If ON *nickname* is specified, UNIQUE should be specified only if the data for the index key contains unique values for every row of the data source table. The uniqueness will not be checked.

For an index over XML data, UNIQUE can be specified only if the specified *pattern-expression* specifies a single complete path and does not contain a descendant or descendant-or-self axis, *"/ /"*, an *xml-wildcard*, *node()*, or *processing-instruction()* (SQLSTATE 429BS).

INDEX *index-name*

Names the index or index specification. The name, including the implicit or explicit qualifier, must not identify an index or index specification that is described in the catalog, or an existing index on a declared temporary table (SQLSTATE 42704). The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

The implicit or explicit qualifier for indexes on declared global temporary tables must be SESSION (SQLSTATE 428EK).

ON *table-name* or *nickname*

The *table-name* identifies a table on which an index is to be created. The table must be a base table (not a view), a materialized query table described in the catalog, or a declared temporary table. The name of a declared temporary table must be qualified with SESSION. The *table-name* must not identify a catalog table (SQLSTATE 42832). If UNIQUE is specified and *table-name* is a typed table, it must not be a subtable (SQLSTATE 429B3).

nickname is the nickname on which an index specification is to be created. The *nickname* references either a data source table whose index is described by the index specification, or a data source view that is based on such a table. The *nickname* must be listed in the catalog.

column-name

For an index, *column-name* identifies a column that is to be part of the index key. For an index specification, *column-name* is the name by which the federated server references a column of a data source table.

Each *column-name* must be an unqualified name that identifies a column of the table. Up to 64 columns can be specified. If *table-name* is a typed table, up to 63 columns can be specified. If *table-name* is a subtable, at least one *column-name* must be introduced in the subtable; that is, not inherited from a supertable (SQLSTATE 428DS). No *column-name* can be repeated (SQLSTATE 42711).

The sum of the stored lengths of the specified columns must not be greater than the index key length limit for the page size. For key length limits, see "SQL limits". If *table-name* is a typed table, the index key length limit is further reduced by 4 bytes. Note that this length limit can be reduced even more by

system overhead, which varies according to the data type of the column and whether or not the column is nullable. For more information on overhead affecting this limit, see “Byte Counts” in “CREATE TABLE”.

Note that this length can be reduced by system overhead, which varies according to the data type of the column and whether it is nullable. For more information on overhead affecting this limit, see “Byte Counts” in “CREATE TABLE”.

No LOB column, DATALINK column, LONG VARCHAR column, LONG VARGRAPHIC column, or distinct type column based on a LOB, DATALINK, LONG VARCHAR, or LONG VARGRAPHIC can be used as part of an index, even if the length attribute of the column is small enough to fit within the index key length limit for the page size (SQLSTATE 54008). A structured type column can only be specified if the EXTEND USING clause is also specified (SQLSTATE 42962). If the EXTEND USING clause is specified, only one column can be specified, and the type of the column must be a structured type or a distinct type that is not based on a LOB, DATALINK, LONG VARCHAR, or LONG VARGRAPHIC (SQLSTATE 42997).

If an index has only one column, and that column has the XML data type, and the GENERATE KEY USING XMLPATTERN clause is also specified, the index is an index over XML data. A column with the XML data type can be specified only if the GENERATE KEY USING XMLPATTERN clause is also specified (SQLSTATE 42962). If the GENERATE KEY USING XMLPATTERN clause is specified, only one column can be specified, and the type of the column must be XML.

ASC

Specifies that index entries are to be kept in ascending order of the column values; this is the default setting. ASC cannot be specified for indexes that are defined with EXTEND USING (SQLSTATE 42601).

DESC

Specifies that index entries are to be kept in descending order of the column values. DESC cannot be specified for indexes that are defined with EXTEND USING, or if the index is an index over XML data (SQLSTATE 42601).

NOT PARTITIONED

Indicates that a single index should be created that spans all of the data partitions defined for the table. The *table-name* must identify a table defined with data partitions (SQLSTATE 53036).

IN *tablespace-name*

Specifies the table space in which the index is to be created. This clause is only supported for indexes on partitioned tables. You can specify this clause even if the INDEX IN clause was specified when the table was created. This will override that clause.

The table space specified by *tablespace-name* must be in the same database partition group as the data table spaces for the table and manage space in the same way as the other table spaces of the partitioned table (SQLSTATE 42838); it must be a table space on which the authorization ID of the statement holds the USE privilege.

If the IN clause is not specified, the index is created in the table space that was specified by the INDEX IN clause on the CREATE TABLE statement. If no INDEX IN clause was specified, the table space of the first visible or attached data partition of the table is used. This is the first partition in the list of data

partitions that are sorted on the basis of range specifications. If the IN clause is not specified, the authorization ID of the statement is not required to have the USE privilege on the default table space.

SPECIFICATION ONLY

Indicates that this statement will be used to create an index specification that applies to the data source table referenced by *nickname*. SPECIFICATION ONLY must be specified if *nickname* is specified (SQLSTATE 42601). It cannot be specified if *table-name* is specified (SQLSTATE 42601).

If the index specification applies to an index that is unique, DB2 does not verify that the column values in the remote table are unique. If the remote column values are not unique, queries against the nickname that include the index column might return incorrect data or errors.

This clause cannot be used when creating an index on a declared temporary table (SQLSTATE 42995).

INCLUDE

This keyword introduces a clause that specifies additional columns to be appended to the set of index key columns. Any columns included with this clause are not used to enforce uniqueness. These included columns might improve the performance of some queries through index only access. The columns must be distinct from the columns used to enforce uniqueness (SQLSTATE 42711). UNIQUE must be specified when INCLUDE is specified (SQLSTATE 42613). The limits for the number of columns and sum of the length attributes apply to all of the columns in the unique key and in the index.

This clause cannot be used with declared temporary tables (SQLSTATE 42995).

column-name

Identifies a column that is included in the index but not part of the unique index key. The same rules apply as defined for columns of the unique index key. The keywords ASC or DESC may be specified following the column-name but have no effect on the order.

INCLUDE cannot be specified for indexes that are defined with EXTEND USING, if *nickname* is specified, or if the index is an XML values index (SQLSTATE 42601).

xml-index-specification

Specifies how index keys are generated from XML documents that are stored in an XML column. *xml-index-specification* cannot be specified if there is more than one index column, or if the column does not have the XML data type.

This clause only applies to XML columns (SQLSTATE 429BS).

GENERATE KEY USING XMLPATTERN *xmlpattern-clause*

Specifies the parts of an XML document that are to be indexed. List data type nodes are not supported in the index. If a node is qualified by the *xmlpattern-clause* and an XML schema exists that specifies that the node is a list data type, then the list data type node cannot be indexed (SQLSTATE 23526 for CREATE INDEX statements, or SQLSTATE 23525 for INSERT and UPDATE statements).

xmlpattern-clause

Contains a pattern expression that identifies the nodes that are to be indexed. It consists of an optional *namespace-declaration* and a required *pattern-expression*.

namespace-declaration

If the pattern expression contains qualified names, a *namespace-declaration* must be specified to define namespace prefixes. A default namespace can be defined for unqualified names.

DECLARE NAMESPACE *namespace-prefix=namespace-uri*

Maps *namespace-prefix*, which is an NCName, to *namespace-uri*, which is a string literal. The *namespace-declaration* can contain multiple *namespace-prefix-to-namespace-uri* mappings. The *namespace-prefix* must be unique within the list of *namespace-declaration* (SQLSTATE 10503).

DECLARE DEFAULT ELEMENT NAMESPACE *namespace-uri*

Declares the default namespace URI for unqualified element names or types. If no default namespace is declared, unqualified names of elements and types are in no namespace. Only one default namespace can be declared (SQLSTATE 10502).

pattern-expression

Specifies the nodes in an XML document that are indexed. The *pattern-expression* can contain pattern-matching characters (*). It is similar to a path expression in XQuery, but supports a subset of the XQuery language that is supported by DB2.

/ (forward slash)

Separates path expression steps.

// (double forward slash)

This is the abbreviated syntax for */descendant-or-self::node()/*. You cannot use *// (double forward slash)* if you also specify UNIQUE.

forward-axis

child::

Specifies children of the context node. This is the default, if no other forward axis is specified.

@ Specifies attributes of the context node. This is the abbreviated syntax for *attribute::*.

attribute::

Specifies attributes of the context node.

descendant::

Specifies the descendants of the context node. You cannot use *descendant::* if you also specify UNIQUE.

self::

Specifies just the context node itself.

descendant-or-self::

Specifies the context node and the descendants of the context node. You cannot use *descendant-or-self::* if you also specify UNIQUE.

xmlname-test

Specifies the node name for the step in the path using a qualified XML name (*xml-qname*) or a wildcard (*xml-wildcard*).

xml-ncname

An XML name as defined by XML 1.0. It cannot include a colon character.

xml-qname

Specifies a qualified XML name (also known as a QName) that can have two possible forms:

- *xml-namespace:xml-ncname*, where the *xml-namespace* is an *xml-ncname* that identifies an in-scope namespace
- *xml-ncname*, which indicates that the default namespace should be applied as the implicit *xml-namespace*

xml-wildcard

Specifies an *xml-qname* as a wildcard that can have three possible forms:

- * (a single asterisk character) indicates any *xml-qname*
- *xml-namespace:** indicates any *xml-ncname* within the specified namespace
- **:xml-ncname* indicates a specific XML name in any in-scope namespace

You cannot use *xml-wildcard* if you also specify UNIQUE.

xmlkind-test

Use these options to specify what types of nodes you pattern match. The following options are available to you:

node()

Matches any node. You cannot use *node()* if you also specify UNIQUE.

text()

Matches any text node.

comment()

Matches any comment node.

processing-instruction()

Matches any processing instruction node. You cannot use *processing-instruction()* if you also specify UNIQUE.

xmltype-clause

AS data-type

Specifies the data type to which indexed values are converted before they are stored. Values are converted to the index XML data type that corresponds to the specified index SQL data type.

Table 24. Corresponding index data types

Index XML data type	Index SQL data type
xs:string	VARCHAR(<i>integer</i>), VARCHAR HASHED
xs:double	DOUBLE
xs:date	DATE
xs:dateTime	TIMESTAMP

For VARCHAR(*integer*) and VARCHAR HASHED, the value is converted to an xs:string value using the XQuery function fn:string. The length attribute of VARCHAR(*integer*) is applied as a

constraint to the resulting `xs:string` value. An index SQL data type of `VARCHAR HASHED` applies a hash algorithm to the resulting `xs:string` value to generate a hash code that is inserted into the index.

For indexes using the data types `DOUBLE`, `DATE`, and `TIMESTAMP`, the value is converted to the index XML data type using the XQuery cast expression. Invalid XML values for the target index XML data type are ignored and are not indexed. The value will be inserted into the table, but it will not be inserted into the index. No error or warning is raised since specifying these data types is not considered a constraint on the values. Note that the index can ignore only invalid XML values for the data type. Valid values must conform to the DB2 representation of the value for the index XML data type, or an error will be issued (SQLSTATE 23526, sqlcode -20306).

If the index is unique, the uniqueness of the value is enforced after the value is converted to the indexed type.

data-type

The following data type is supported:

SQL data type (*sql-data-type*)

sql-data-type

Supported SQL data types are:

VARCHAR(integer)

If this form of `VARCHAR` is specified, DB2 uses *integer* as a constraint. If document nodes that are to be indexed have values that are longer than *integer*, the documents are not inserted into the table if the index already exists. If the index does not exist, the index is not created. *integer* is a value between 1 and a page size-dependent maximum. Table 25 shows the maximum value for each page size.

Table 25. Maximum length of document nodes by page size

Page size	Maximum length of document node (bytes)
4KB	817
8KB	1841
16KB	3889
32KB	7985

XQuery semantics are used for string comparisons, where trailing blanks are significant. This differs from SQL semantics, where trailing blanks are insignificant during comparisons.

VARCHAR HASHED

Specify `VARCHAR HASHED` to handle indexing of arbitrary length character strings. The length of an indexed string has no limit. DB2 generates an eight-byte hash code over the entire string. Indexes

that use these hashed character strings can be used only for equality lookups. XQuery semantics are used for string equality comparisons, where trailing blanks are significant. This differs from SQL semantics, where trailing blanks are insignificant during comparisons. The hash on the string preserves XQuery semantics for equality and not SQL semantics.

DOUBLE

Specifies that the data type DOUBLE is used for indexing numeric values. Unbounded decimal types and 64 bit integers may lose precision when they are stored as a DOUBLE value. The values for DOUBLE may include the special numeric values *NaN*, *INF*, *-INF*, *+0*, and *-0*, even though the SQL data type DOUBLE itself does not support these values.

DATE

Specifies that the data type DATE is used for indexing XML values. Note that the XML schema data type for *xs:date* allows greater precision than the SQL data type. If an out-of-range value is encountered, an error is returned.

TIMESTAMP

Specifies that the data type TIMESTAMP is used for indexing XML values. Note that the XML schema data type for *xs:dateTime* allows greater precision than the SQL data type. If an out-of range value is encountered, an error is returned.

CLUSTER

Specifies that the index is the clustering index of the table. The cluster factor of a clustering index is maintained or improved dynamically as data is inserted into the associated table, by attempting to insert new rows physically close to the rows for which the key values of this index are in the same range. Only one clustering index may exist for a table so CLUSTER may not be specified if it was used in the definition of any existing index on the table (SQLSTATE 55012). A clustering index may not be created on a table that is defined to use append mode (SQLSTATE 428D8).

CLUSTER is disallowed if *nickname* is specified, or if the index is an index over XML data (SQLSTATE 42601). This clause cannot be used with declared temporary tables (SQLSTATE 42995) or range-clustered tables (SQLSTATE 429BG).

EXTEND USING *index-extension-name*

Names the *index-extension* used to manage this index. If this clause is specified, then there must be only one *column-name* specified and that column must be a structured type or a distinct type (SQLSTATE 42997). The *index-extension-name* must name an index extension described in the catalog (SQLSTATE 42704). For a distinct type, the column must exactly match the type of the corresponding source key parameter in the index extension. For a structured type column, the type of the corresponding source key parameter must be the same type or a supertype of the column type (SQLSTATE 428E0).

This clause cannot be used with declared temporary tables (SQLSTATE 42995).

constant-expression

Identifies values for any required arguments for the index extension. Each expression must be a constant value with a data type that exactly matches the defined data type of the corresponding index extension parameters, including length or precision, and scale (SQLSTATE 428E0). This clause must not exceed 32 768 bytes in length in the database code page (SQLSTATE 22001).

PCTFREE *integer*

Specifies what percentage of each index page to leave as free space when building the index. The first entry in a page is added without restriction. When additional entries are placed in an index page at least *integer* percent of free space is left on each page. The value of *integer* can range from 0 to 99. If a value greater than 10 is specified, only 10 percent free space will be left in non-leaf pages. The default is 10.

PCTFREE is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with declared temporary tables (SQLSTATE 42995).

LEVEL2 PCTFREE *integer*

Specifies what percentage of each index level 2 page to leave as free space when building the index. The value of *integer* can range from 0 to 99. If LEVEL2 PCTFREE is not set, a minimum of 10 or PCTFREE percent of free space is left on all non-leaf pages. If LEVEL2 PCTFREE is set, *integer* percent of free space is left on level 2 intermediate pages, and a minimum of 10 or *integer* percent of free space is left on level 3 and higher intermediate pages.

LEVEL2 PCTFREE is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with declared temporary tables (SQLSTATE 42995).

MINPCTUSED *integer*

Indicates whether index leaf pages are merged online, and the threshold for the minimum percentage of space used on an index leaf page. If, after a key is removed from an index leaf page, the percentage of space used on the page is at or below *integer* percent, an attempt is made to merge the remaining keys on this page with those of a neighboring page. If there is sufficient space on one of these pages, the merge is performed and one of the pages is deleted. The value of *integer* can be from 0 to 99. A value of 50 or below is recommended for performance reasons. Specifying this option will have an impact on update and delete performance. For type 2 indexes, merging is only done during update and delete operations when there is an exclusive table lock. If an exclusive table lock does not exist, keys are marked as pseudo deleted during update and delete operations, and no merging is done. Consider using the CLEANUP ONLY ALL option of REORG INDEXES to merge leaf pages instead of using the MINPCTUSED option of CREATE INDEX.

MINPCTUSED is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with declared temporary tables (SQLSTATE 42995).

DISALLOW REVERSE SCANS

Specifies that an index only supports forward scans or scanning of the index in the order that was defined at index creation time.

DISALLOW REVERSE SCANS cannot be specified together with *nickname* (SQLSTATE 42601).

ALLOW REVERSE SCANS

Specifies that an index can support both forward and reverse scans; that is, scanning of the index in the order that was defined at index creation time, and scanning in the opposite order.

ALLOW REVERSE SCANS cannot be specified together with *nickname* (SQLSTATE 42601).

PAGE SPLIT

Specifies an index split behavior. The default is SYMMETRIC.

SYMMETRIC

Specifies that pages are to be split roughly in the middle.

HIGH

Specifies an index page split behavior that uses the space on index pages efficiently when the values of the index keys being inserted follow a particular pattern. For a subset of index key values, the leftmost column or columns of the index must contain the same value, and the rightmost column or columns of the index must contain values that increase with each insertion. For details, see “Options on the CREATE INDEX statement”.

LOW

Specifies an index page split behavior that uses the space on index pages efficiently when the values of the index keys being inserted follow a particular pattern. For a subset of index key values, the leftmost column or columns of the index must contain the same value, and the rightmost column or columns of the index must contain values that decrease with each insertion. For details, see “Options on the CREATE INDEX statement”.

COLLECT STATISTICS

Specifies that basic index statistics are to be collected during index creation.

DETAILED

Specifies that extended index statistics (CLUSTERFACTOR and PAGE_FETCH_PAIRS) are also to be collected during index creation.

SAMPLED

Specifies that sampling can be used when compiling extended index statistics.

Rules:

- The CREATE INDEX statement will fail (SQLSTATE 01550) if attempting to create an index that matches an existing index.

Two index descriptions are considered duplicates if:

- the set of columns (both key and include columns) and their order in the index is the same as that of an existing index AND
- the ordering attributes are the same AND
- both the previously existing index and the one being created are non-unique OR the previously existing index is unique AND
- if both the previously existing index and the one being created are unique, the key columns of the index being created are the same or a superset of key columns of the previously existing index.

For indexes over XML data, the index descriptions are not considered duplicates if the index names are different, even if the indexed XML column, the XML patterns, and the data type, including its options, are identical.

- Unique indexes on system-maintained MQTs are not supported (SQLSTATE 42809).
- The COLLECT STATISTICS options are not supported if a nickname is specified (SQLSTATE 42601).

Notes:

- Indexes over XML data do not support concurrent write access while CREATE INDEX is executing.
- For relational indexes only: Concurrent read/write access to the table is permitted while an index is being created. Once the index has been built, changes that were made to the table during index creation time are forward-fitted to the new index. Write access to the table is then briefly blocked while index creation completes, after which the new index becomes available. To circumvent this default behavior, use the LOCK TABLE statement to explicitly lock the table before issuing a CREATE INDEX statement. (The table can be locked in either SHARE or EXCLUSIVE mode, depending on whether read access is to be allowed.)
- If the named table already contains data, CREATE INDEX creates the index entries for it. If the table does not yet contain data, CREATE INDEX creates a description of the index; the index entries are created when data is inserted into the table.
- Once the index is created and data is loaded into the table, it is advisable to issue the RUNSTATS command. The RUNSTATS command updates statistics collected on the database tables, columns, and indexes. These statistics are used to determine the optimal access path to the tables. By issuing the RUNSTATS command, the database manager can determine the characteristics of the new index. If data has been loaded before the CREATE INDEX statement is issued, it is recommended that the COLLECT STATISTICS option on the CREATE INDEX statement be used as an alternative to the RUNSTATS command.
- Creating an index with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- The optimizer can recommend indexes prior to creating the actual index.
- If an index specification is being defined for a data source table that has an index, the name of the index specification does not have to match the name of the index.
- The optimizer uses index specifications to improve access to the data source tables that the specifications apply to.
- **Compatibilities**
 - For compatibility with DB2 for OS/390®:
 - The following syntax is tolerated and ignored:
 - CLOSE
 - DEFINE
 - FREEPAGE
 - GBPCACHE
 - PIECESIZE
 - TYPE 2
 - using-block
 - The following syntax is accepted as the default behavior:
 - COPY NO
 - DEFER NO

Examples:

Example 1: Create an index named UNIQUE_NAM on the PROJECT table. The purpose of the index is to ensure that there are not two entries in the table with the same value for project name (PROJNAME). The index entries are to be in ascending order.

```
CREATE UNIQUE INDEX UNIQUE_NAM  
ON PROJECT(PROJNAME)
```

Example 2: Create an index named JOB_BY_DPT on the EMPLOYEE table. Arrange the index entries in ascending order by job title (JOB) within each department (WORKDEPT).

```
CREATE INDEX JOB_BY_DPT  
ON EMPLOYEE (WORKDEPT, JOB)
```

Example 3: The nickname EMPLOYEE references a data source table called CURRENT_EMP. After this nickname was created, an index was defined on CURRENT_EMP. The columns chosen for the index key were WORKDEPT and JOB. Create an index specification that describes this index. Through this specification, the optimizer will know that the index exists and what its key is. With this information, the optimizer can improve its strategy to access the table.

```
CREATE UNIQUE INDEX JOB_BY_DEPT  
ON EMPLOYEE (WORKDEPT, JOB)  
SPECIFICATION ONLY
```

Example 4: Create an extended index type named SPATIAL_INDEX on a structured type column location. The description in index extension GRID_EXTENSION is used to maintain SPATIAL_INDEX. The literal is given to GRID_EXTENSION to create the index grid size.

```
CREATE INDEX SPATIAL_INDEX ON CUSTOMER (LOCATION)  
EXTEND USING (GRID_EXTENSION (x'000100100010001000400010'))
```

Example 5: Create an index named IDX1 on a table named TAB1, and collect basic index statistics on index IDX1.

```
CREATE INDEX IDX1 ON TAB1 (col1) COLLECT STATISTICS
```

Example 6: Create an index named IDX2 on a table named TAB1, and collect detailed index statistics on index IDX2.

```
CREATE INDEX IDX2 ON TAB1 (col2) COLLECT DETAILED STATISTICS
```

Example 7: Create an index named IDX3 on a table named TAB1, and collect detailed index statistics on index IDX3 using sampling.

```
CREATE INDEX IDX3 ON TAB1 (col3) COLLECT SAMPLED DETAILED STATISTICS
```

Example 8: Create a unique index named A_IDX on a partitioned table named MYNUMBERDATA in table space IDX_TBSP.

```
CREATE UNIQUE INDEX A_IDX ON MYNUMBERDATA (A) IN IDX_TBSP
```

Example 9: Create a non-unique index named B_IDX on a partitioned table named MYNUMBERDATA in table space IDX_TBSP.

```
CREATE INDEX B_IDX ON MYNUMBERDATA (B)  
NOT PARTITIONED IN IDX_TBSP
```

Example 10: Create an index over XML data on a table named COMPANYINFO, which contains an XML column named COMPANYDOCS. The XML column COMPANYDOCS contains a large number of XML documents similar to the one below:

```

<company name="Company1">
  <emp id="31201" salary="60000" gender="Female">
    <name>
      <first>Laura</first>
      <last>Brown</last>
    </name>
    <dept id="M25">
      Finance
    </dept>
  </emp>
</company>

```

Users of the COMPANYINFO table often need to retrieve employee information using the employee ID. An index like the following one can make that retrieval more efficient.

```

CREATE INDEX EMPINDEX ON COMPANYINFO(COMPANYDOCS)
GENERATE KEY USING XMLPATTERN '/company/emp/@id'
AS SQL DOUBLE

```

Example 11: The following index is logically equivalent to the index created in the previous example, except that it uses unabbreviated syntax.

```

CREATE INDEX EMPINDEX ON COMPANYINFO(COMPANYDOCS)
GENERATE KEY USING XMLPATTERN '/child::company/child::emp/attribute::id'
AS SQL DOUBLE

```

Example 12: Create an index on a column named DOC, indexing only the book title as a VARCHAR(100). Because the book title should be unique across all books, the index must be unique.

```

CREATE UNIQUE INDEX MYDOCSIDX ON MYDOCS(DOC)
GENERATE KEY USING XMLPATTERN '/book/title'
AS SQL VARCHAR(100)

```

Example 13: Create an index on a column named DOC, indexing the chapter number as a DOUBLE. This example includes namespace declarations.

```

CREATE INDEX MYDOCSIDX ON MYDOCS(DOC)
GENERATE KEY USING XMLPATTERN
'declare namespace b="http://www.foobar.com/book/";
declare namespace c="http://acme.org/chapters";
/b:book/c:chapter/@number'
AS SQL DOUBLE

```

Related concepts:

- “Data type conversion for indexes over XML data” on page 102
- “Restrictions on indexes over XML data” on page 126
- “Understanding clustering index behavior on partitioned tables” in *Performance Guide*
- “Understanding index behavior on partitioned tables” in *Performance Guide*
- “XMLEXISTS predicate usage” on page 67
- “Options on the CREATE INDEX statement” in *Administration Guide: Implementation*

Related reference:

- “CREATE INDEX EXTENSION statement” in *SQL Reference, Volume 2*
- “CREATE TABLE statement” in *SQL Reference, Volume 2*
- “Interaction of triggers and constraints” in *SQL Reference, Volume 1*
- “SQL and XQuery limits” in *SQL Reference, Volume 1*

Related samples:

- “dbstat.sqb -- Reorganize table and run statistics (MF COBOL)”
- “TbGenCol.java -- How to use generated columns (JDBC)”

Sample queries against indexes over XML data

Sample queries that can use an index over XML data:

Queries with a wide variety of different predicates can exploit an index over XML data. Some examples of XQuery predicates matched with indexes they can use are shown below. Queries are followed by matching indexes.

Example 1. Issue a query for equality: Find the employee with ID 42366:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[@id='42366']
```

Alternatively, this query could also be issued using the full XQuery syntax:

```
XQUERY for $i in db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[@id='42366']
return $i
```

```
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@id' AS SQL VARCHAR(5)
```

Example 2. Query for a range: Find employees with a salary greater than 35000:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[@salary > 35000]
```

```
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '//@salary' AS SQL DOUBLE
```

Example 3. Issue a query which includes a disjunction (OR): Find employees that are in the Finance department or in the Marketing department:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[dept/text()='Finance' or
dept/text()='Marketing']
```

```
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/dept/text()' AS SQL
VARCHAR(30)
```

Example 4. Different queries can be satisfied by the same index:

Find the employee with the ID 31201:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[@id='31201']
```

Find departments with the ID K55

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp/dept[@id='K55']
```

```
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '//@id' AS SQL VARCHAR(25)
```

Example 5. Query predicates can contain paths: Find employees with last name *Murphy* who are in the Sales department:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[name/last='Murphy' and
dept/text()='Sales']
```

```
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/name/last' AS SQL
VARCHAR(100)

CREATE INDEX deptindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/dept/text()' AS SQL
VARCHAR(30)
```

Example 6. Exercise hierarchical containment during queries: A query can use indexes to perform ANDing at different levels in the document hierarchy. A query can also use the indexes to determine what children nodes belong to the same ancestor to do appropriate filtering.

Find companies with employees that have a salary equal to 60000 and find companies with female employees. In the sample XML fragments of the XML data indexing overview topic (see the Related concepts section), both Company1 and Company2 would qualify.

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company[emp/@salary=60000 and
emp/@gender='Female']
```

Find employees who have a salary equal to 60000 and who are female. Only Laura Brown from Company1 would qualify.

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[@salary=60000 and
@gender='Female']
```

```
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@salary' AS DOUBLE
CREATE INDEX genderindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@gender' AS SQL
VARCHAR(10)
```

Example 7. A query can use the descendant-or-self axis (//) and make use of indexes, provided that the query predicate is at least as restrictive as, or more restrictive than, the index pattern.

Find employees with department ID K55:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company//emp[.//dept//@id='K55' ]
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '//emp//@id' AS SQL VARCHAR(25)
```

Sample queries that cannot use an index over XML data:

There are some conditions when a query cannot use an index over XML data. Some examples of XQuery predicates that cannot make use of their intended indexes as shown are listed below.

Example 1. The data type requested by the query must match the indexed data type, before the query can use the index. In this example, the query requests the employee ID as a string, but the ID is indexed as a number:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')/company/emp[@id='31664']
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@id' AS SQL DOUBLE
```

Example 2. The XML pattern expression used to create the index may be more restrictive than the query predicate. In this example, the query cannot use the index, because the query retrieves both department ID and employee IDs but the index contains only employee IDs:

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')//@id
CREATE INDEX empindex on company(companydocs)
GENERATE KEY USING XMLPATTERN '/company/emp/@id' AS SQL VARCHAR(5)
```

The following query retrieves the employees who have the employee ID 31201 or the department ID K55. Since the ID may either be an employee ID or a department ID, but the index contains only department IDs, the index cannot be used as created.

```
db2-fn:xmlcolumn('COMPANY.COMPANYDOCS')//emp[.//@id='31201' or .//@id='K55']
CREATE INDEX empindex on company(companydocs)
    GENERATE KEY USING XMLPATTERN '//dept//@id' AS SQL VARCHAR(5)
```

Related concepts:

- “Guidelines for matching indexes with queries overview” on page 72
- “XML data indexing overview” on page 93

Related reference:

- “CREATE INDEX ” on page 108

Restrictions on indexes over XML data

The following restrictions apply to indexes over XML data (also known as indexes over XML columns):

Data type support

Every XML pattern expression specified in the CREATE INDEX statement must be associated with a data type. Only four SQL-based data types are supported: DATE, TIMESTAMP, VARCHAR, and DOUBLE.

Concurrency levels

Support for some concurrency levels is restricted when processing XML columns and associated indexes. The following table describes what concurrency levels are or are not supported:

Table 26. Supported concurrency levels for indexes over XML data

Function	Concurrency Level	Supported
CREATE INDEX on an XML column	Concurrent read access to the table	Yes
CREATE INDEX on an XML column	Concurrent write access to the table	No
CREATE INDEX on a non-XML column	Concurrent read and write access to the table	Yes
REORG INDEXES ALL FOR TABLE (with at least one XML column in the table)	Index clause: ALLOW [READ NO ACCESS]	Yes. An index on the XML column may or may not exist
REORG INDEXES ALL FOR TABLE (with at least one XML column in the table)	Index clause: ALLOW WRITE ACCESS	No. Not supported for all indexes, even if no index on an XML column exists
REORG INDEXES ALL FOR TABLE (with at least one XML column in the table and clean up only of pseudo deletes)	Index clause: ALLOW [READ WRITE] ACCESS CLEANUP ONLY	Yes
REORG TABLE (with at least one XML column in the table)	Table clause: ALLOW [READ NO] ACCESS	Yes. An index on the XML column may or may not exist
REORG TABLE INPLACE (no index over an XML column exists on the table)	Table clause: ALLOW [READ WRITE] ACCESS	Yes
REORG TABLE INPLACE (at least one index over an XML column exists on the table)	Table Clause: ALLOW [READ WRITE] ACCESS	No

XML list elements

List data type nodes cannot be indexed. If a node is qualified by the *xmlpattern-clause* and an XML schema exists that specifies that the node is a list data type, the list data type node cannot be indexed. Issuing a CREATE INDEX statement on a list data type node will return an error (SQLSTATE 23526, sqlcode -20306). Issuing INSERT and UPDATE statements will also return an error (SQLSTATE 23525, sqlcode -20305).

Creating indexes on XML columns is also subject to restrictions placed on native XML data store overall. Please refer to the Related reference section below.

Related concepts:

- “XML data indexing overview” on page 93

Related reference:

- “CREATE INDEX ” on page 108
- Chapter 9, “Restrictions on native XML data store,” on page 233

Recreation of indexes over XML data

An index over XML data is recreated under the following circumstances:

- During an offline REORG INDEX or REORG INDEXES command with the option ALLOW READ ACCESS or the option ALLOW NO ACCESS specified.
- During an offline REORG TABLE command.
- When you issue an IMPORT command with the REPLACE option specified.
- When a query, insert, delete, or update operation attempts to access a table or an index and detects that the index object is marked bad.

Note that all indexes associated with native XML data store functionality are contained in the same index object for a table as relational indexes. This includes any XML path indexes, XML regions indexes, and indexes over XML data that may exist. Single indexes are not recreated alone. If index recreation becomes necessary, all indexes in the index object are recreated together.

Related concepts:

- “Other database objects associated with XML columns” on page 107

Related reference:

- “IMPORT Command” in *Command Reference*
- “REORG INDEXES/TABLE command” in *Command Reference*

Troubleshooting

Common XML indexing issues

Cases where values are not indexed:

The index XML data type acts like a filter and is not a constraint since the user may have multiple indexes with different data types on the same XML column. XML values that do not have a valid lexical form for the target index XML data

type are ignored and not indexed. If the value cannot be converted to the index XML data type, then the value is inserted in the table, but it is not inserted into the index. No error or warning is raised since specifying the index XML data type is not considered a constraint on the values. Note that the index can ignore only invalid XML values for the data type. Valid values must conform to DB2's representation of the value for the data type, or an error will be issued.

For example, suppose the user creates two indexes on the same pattern but with different data types. The index *ALLID* uses the VARCHAR data type and indexes on all the IDs in the document (both department IDs and employee IDs). The index *EMPID* indexes only on the numeric employee IDs and uses the DOUBLE data type as a filter:

```
CREATE INDEX ALLID ON DEPARTMENT(DEPTDOCS)
  GENERATE KEY USING XMLPATTERN '//@id' AS SQL VARCHAR(10)

CREATE INDEX EMPID ON DEPARTMENT(DEPTDOCS)
  GENERATE KEY USING XMLPATTERN '//@id' AS SQL DOUBLE
```

The department ID value, *M25*, is a valid VARCHAR data type value and will be inserted into the index *ALLID*. However, *M25* cannot be converted to the DOUBLE data type, so that the value will not be inserted into *EMPID*, and no error or warning is raised. The value is inserted for the document stored in the table.

Although the value *M25* does not exist in the DOUBLE index *EMPID*, queries may still use the DOUBLE index to retrieve all the matching numeric values and no conversion errors will occur, because the document that contains *M25* will not be accessed.

However, if the query does not use the DOUBLE index *EMPID* and scans the document using the *//@id=25* predicate, then a conversion error will occur because the value *M25* matches the pattern and still exists in the document but it is not a numeric value.

Note that all values in the document are valid for the *xs:string* (SQL VARCHAR) data type. The case where values are not inserted into the index occurs only for the data types *xs:double* (SQL DOUBLE), *xs:date* (SQL DATE), and *xs:dateTime* (SQL TIMESTAMP).

Cases where a value is not indexed, without a specified XML schema:

If there is no specified XML schema, then an attempt is made to convert the value to the target index XML type.

If the value would be invalid XML data for the target index XML data type (for *xs:double*, *xs:date*, or *xs:dateTime*), then the value is not indexed, and no error or warning is issued.

All values represent valid XML data for indexes using the VARCHAR data type, since *fn:string* is used.

Cases where a value is not indexed when an XML schema is specified:

If an XML schema is specified, then the parser will validate the source value and the value will be converted to the schema data type. However, if the conversion from the schema data type to the index XML data type fails (for *xs:double*, *xs:date*, *xs:dateTime*), then the value is not indexed and no error or warning is issued.

Conversion rules follow the allowed type casting function table of XQuery, as supported by DB2 (see the Related reference section below).

Cases of document rejection or CREATE INDEX statement failure:

For the following types of indexing errors, the XML document will be rejected for INSERT or UPDATE statements (SQLSTATE 23525, sqlcode -20305). In the case of a CREATE INDEX statement on a populated table where the XML document already exists in the table, the CREATE INDEX statement will fail (SQLSTATE 23526, sqlcode -20306) and the document remains stored in the table.

VARCHAR(integer) length constraint errors

The length of a resulting index value from one or more XML pattern expressions exceeds the user-specified length constraint for the VARCHAR data type.

Unsupported list data type node error

One or more XML node values in an XML value is a list data type node that cannot be indexed by the specified index. List data type nodes are not supported by indexes over XML data.

Conversion errors

If the source value is a valid XML value which cannot be converted to DB2's representation for either the schema data type or for the index XML data type because of an internal DB2 limitations, an error is issued. The error must be issued to maintain consistent results. The results for the query should include the value that exceeds the supported limit.

Table 27. Some examples of internal DB2 limitations

XML data type	XML schema	DB2 range (min : max)
xs:date	No maximum limit for years Negative dates are supported	0001-01-01: 9999-12-31
xs:dateTime	No maximum limit for years Negative dates are supported Arbitrary precision is supported for fractional seconds	0001-01-01T00:00:00.000000Z: 9999-12-31T23:59:59.999999Z
xs:integer	No limit on minimum or maximum range	-9223372036854775808: 9223372036854775807

DB2 does not support the entire range of XML values. Value ranges that are unsupported include:

- Date or dateTime values with year > 9999 or < 0
- Date or dateTime values with fractional second precision > 6 digits
- Out-of-range numeric values

Summary tables for conversion to the index XML data type:

For the data to convert successfully to the target index XML data type, the source value must be lexically valid according to the schema data type/index XML data type and the value must be within DB2 limits for the schema data type/index XML data type.

When no XML schema is present:

Table 28. Summary table for conversion to the index XML data type when no xml schema is present

The value is valid according to the index XML data type (all values are valid for the xs:string data type)	The value is within DB2 limits for the index XML data type	Indexing result
No	not applicable	The value is ignored and not indexed.
Yes	Yes	The value is indexed.
Yes	No	Error: the value is outside of DB2 limits.

When an XML schema is present:

Table 29. Summary table for conversion to the index XML data type when an xml schema is present

The value is valid according to the schema data type	The value is within DB2 limits for the schema data type	The value is valid according to the index XML data type	The value is within DB2 limits for the index XML data type	Indexing result
No	not applicable	not applicable	not applicable	Error: The document is rejected during schema validation, whether an index exists or not
Yes	Yes	Yes	Yes	The value is indexed.
Yes	No	not applicable	not applicable	Error: the value is outside of DB2 limits.
Yes	Yes	No	not applicable	The value is ignored and not indexed.
Yes	Yes	Yes	No	Error: the value is outside of DB2 limits.

Related concepts:

- “Guidelines for matching indexes with queries overview” on page 72
- “XML data indexing overview” on page 93

Related reference:

- “Limits for XQuery data types” in *IBM DB2 XQuery Reference*
- “Type casting (DB2 XQuery)” in *IBM DB2 XQuery Reference*

Preference of database managed table spaces for native XML data store performance

For performance-sensitive applications, particularly those involving extensive INSERT activity, it is strongly recommended that you use database managed space (DMS). If you encounter query performance degradation with native XML data store and are using system managed space (SMS), you should consider switching to DMS.

The use of DMS functionality also allows you to take advantage of autonomic capabilities in DB2.

Related concepts:

- “DMS table spaces” in *Administration Guide: Planning*

Chapter 5. XML schema repository (XSR)

XML schema, DTD, and external entity management using the XML schema repository (XSR)

The XML schema repository (XSR) is a repository for all XML artifacts used to validate and process XML instance documents stored in XML columns.

XML instance documents normally contain a reference to a Uniform Resource Identifier (URI) that points to an associated XML schema, DTD or other external entity. This URI is required to validate the instance document. The DB2 database system manages dependencies on such externally referenced XML artifacts with the XSR without requiring changes to the URI location reference.

Without this mechanism to store associated XML schemas, DTDs or other external entities, an external resource may not be accessible when needed by the database, or may be subject to change without also triggering the necessary changes to an already validated and annotated XML document stored within the database. The XSR also removes the additional overhead required to locate external documents, along with the possible performance impact.

Each database contains an XML schema repository that resides in the database catalog and comprises catalog tables, catalog views and some system defined stored procedures to enter data into these catalog tables.

Related concepts:

- “XML input and output overview” on page 3
- “XSR object registration” on page 134

Related reference:

- “SYSCAT.XSROBJECTAUTH catalog view” in *SQL Reference, Volume 1*
- “SYSCAT.XSROBJECTCOMPONENTS catalog view” in *SQL Reference, Volume 1*
- “SYSCAT.XSROBJECTDEP catalog view” in *SQL Reference, Volume 1*
- “SYSCAT.XSROBJECTHIERARCHIES catalog view” in *SQL Reference, Volume 1*
- “SYSCAT.XSROBJECTS catalog view” in *SQL Reference, Volume 1*
- “applheapsz - Application heap size configuration parameter” in *Performance Guide*

XSR objects

The XML schema repository (XSR) supports the creation of a copy of the information contained in an XML schema, DTD or external entity as an XSR object. This information is used to validate and process XML instance documents stored in XML columns.

New XSR objects must be explicitly added to the XSR before use through the registration process, which identifies the XML schema, DTD or external entity. XSR objects can be registered from Java(TM) applications, stored procedures, or the command line processor.

The most widely used XSR object is the XML schema. Each XML schema in the XSR can consist of one or more XML schema documents. Where an XML schema consists of multiple documents, the document used to begin the registration process is the primary XML schema document. Where an XML schema consists of only one document, that document is the primary XML schema document.

Related concepts:

- “XSR object registration” on page 134

XSR object registration

Before an XML schema, DTD, or external entity can be used for validation and annotation, it must be registered with the XML schema repository (XSR). Registration with the XSR creates an XSR object.

In order to register most XML schemas, the application heap size configuration parameter (applheapsz) needs to be increased. To register very complex XML schemas on a Windows 32-bit operating system, the agent stack size configuration parameter (agent_stack_sz) may also need to be increased. Please refer to the related links below on how to change either of these configuration parameters.

For XML schemas, XSR object registration involves the following steps:

1. Register the XML schema document in the XML schema repository.
2. Specify additional XML schema documents to be included with the XSR object. This step is required only if your XML schema consists of more than one schema document.
3. Complete the registration process with the XML schema repository.

For DTDs and external entities, XSR object registration with the XML schema repository is a single step process.

The XSR object registration steps can be performed from either of the following:

- Java(TM) applications
- Stored procedures
- Command line processor

In the description of these methods below, the following example of an XML schema made up of two XML schema documents is used: "PO.xsd" and "address.xsd", both stored locally in C:\TEMP. The user wants to register this schema under the SQL two-part name of "user1.POschema". The XML Schema has a properties file associated with it called schemaProp.xml. This properties file is also stored locally, in the same C:\TEMP directory. The two XML schema documents do not have properties associated with them. The user defines the URI by which this schema is externally known as "http://myPOschema/PO".

Privileges:

Any user with SYSADM or DBADM authority can register an XSR object. For all other users, the privileges are based on the SQL schema that is supplied during the registration process. If the SQL schema does not exist, then IMPLICIT_SCHEMA authority on the database is required to register the schema. If the SQL schema exists, then the user registering the schema needs CREATEIN privilege on the SQL schema.

For XML schemas, the user to initiate the XSR object registration process (through the XSR_REGISTER stored procedure, for example) must also be the user to specify additional XML schema documents (if applicable) and to complete the registration process.

The USAGE privilege for an XSR object is automatically granted to the creator of the XSR object.

Related concepts:

- “Registering XSR objects through stored procedures” on page 135
- “Registering XSR objects through the command line processor” on page 136

Related reference:

- “agent_stack_sz - Agent stack size configuration parameter” in *Performance Guide*
- “applheapsz - Application heap size configuration parameter” in *Performance Guide*

Registering XSR objects through stored procedures

When a database is created, the stored procedures used to register XML schemas are also created. To register XML schemas through the stored procedure method, simply invoke the stored procedures with the CALL statement.

Note: An XML schema document is not checked for correctness when registering or adding documents. Document checks are performed only when you complete the XML schema registration.

1. Register the primary XML schema document by calling the SYSPROC.XSR_REGISTER stored procedure:

```
CALL SYSPROC.XSR_REGISTER ('user1', 'POschema', 'http://myPOschema/PO',
                           :content_host_var, NULL)
```

2. Before completing the registration, add any additional XML schema documents to be included with the primary XML schema. For our example, this step is not optional, since the XML schema consists of two XML schema documents, both of which must be registered. Use the XSR_ADDSCHEMADOC stored procedure to add additional XML schema documents. In the example below, we add the schema constructs for addresses to the XSR object:

```
CALL SYSPROC.XSR_ADDSCHEMADOC ('user1', 'POschema', 'http://myPOschema/address',
                               :content_host_var, NULL)
```

3. Complete registration by calling the SYSPROC.XSR_COMPLETE stored procedure. In the example below, the last parameter indicates that the XML schema will not be used for decomposition (a value of 1 would indicate that it will be used for decomposition):

```
CALL SYSPROC.XSR_COMPLETE ('user1', 'POschema', :schemaproperty_host_var, 0)
```

Privileges:

Any user with SYSADM or DBADM authority can register an XML schema. For all other users, the privileges are based on the SQL schema that is supplied during the registration process. If the SQL schema does not exist, then IMPLICIT_SCHEMA authority on the database is required to register the schema. If the SQL schema exists, then the user registering the schema needs CREATEIN privilege on the SQL schema.

The USAGE privilege for an XSR object is automatically granted to the creator of the XSR object.

Related concepts:

- “XSR_ADDSCHEMADOC procedure” on page 142
- “XSR_COMPLETE procedure” on page 143
- “XSR_REGISTER procedure” on page 141

Registering XSR objects through the command line processor

To register XML schemas through the command line processor, the following steps are required:

Note: An XML schema document is not checked for correctness when registering or adding documents. Document checks are performed only when you complete schema registration.

1. Register the primary XML schema document by issuing the REGISTER XMLSCHEMA command:

```
REGISTER XMLSCHEMA 'http://myPOschema/PO'  
FROM 'file://c:/TEMP/PO.xsd'  
AS user1.POschema
```
2. Before completing the registration, you may optionally add additional XML schema documents to be included with the primary XML schema. Use the ADD XMLSCHEMA DOCUMENT command to add additional XML schema documents. In the example below, we add the schema constructs for addresses to storage:

```
ADD XMLSCHEMA DOCUMENT TO user1.POschema  
  ADD 'http://myPOschema/address'  
  FROM 'file://c:/TEMP/address.xsd'
```
3. Complete registration by issuing the COMPLETE XMLSCHEMA command:

```
COMPLETE XMLSCHEMA user1.POschema  
WITH 'file://c:/TEMP/schemaProp.xml'
```

Privileges:

Any user with SYSADM or DBADM authority can register an XML schema. For all other users, the privileges are based on the SQL schema that is supplied during the registration process. If the SQL schema does not exist, then IMPLICIT_SCHEMA authority on the database is required to register the schema. If the SQL schema exists, then the user registering the schema needs CREATEIN privilege on the SQL schema.

The USE privilege for XSR objects is automatically granted to the creator of the XSR object.

Related reference:

- “ADD XMLSCHEMA DOCUMENT ” on page 149
- “COMPLETE XMLSCHEMA ” on page 150
- “REGISTER XMLSCHEMA ” on page 147

Java support for XML schema registration and removal

DB2 provides the `SYSPROC.XSR_REGISTER`, `SYSPROC.XSR_ADDSCHEMADOC`, `SYSPROC.XSR_COMPLETE`, and `SYSPROC.XSR_REMOVE` stored procedures that let you register and remove XML schemas and their components.

The IBM DB2 Driver for JDBC and SQLJ provides methods that let you perform the same functions from a Java application program. Those methods are:

DB2Connection.registerDB2XMLSchema

Registers an XML schema in DB2, using one or more XML schema documents. There are two forms of this method: one form for XML schema documents that are input from an `InputStream` objects, and one form for XML schema documents that are in a `String`s.

DB2Connection.deregisterDB2XMLObject

Removes an XML schema definition from DB2.

Before you can invoke these methods, the underlying stored procedures must be installed on the DB2 database server.

Example:Registration of an XML schema: The following example demonstrates the use of `registerDB2XmlSchema` to register an XML schema in DB2 using a single XML schema document (`customer.xsd`) that is read from an input stream. The SQL schema name for the registered schema is `SYSXSR`. The `xmlSchemaLocations` value is null, so DB2 will not find this XML schema on an invocation of `DSN_XMLVALIDATE` that supplies a non-null XML schema location value. No additional properties are registered.

```

public static void registerSchema(
    Connection con,
    String schemaName)
    throws SQLException {
    // Define the registerDB2XmlSchema parameters
    String[] xmlSchemaNameQualifiers = new String[1];
    String[] xmlSchemaNames = new String[1];
    String[] xmlSchemaLocations = new String[1];
    InputStream[] xmlSchemaDocuments = new InputStream[1];
    int[] xmlSchemaDocumentsLengths = new int[1];
    java.io.InputStream[] xmlSchemaDocumentsProperties = new InputStream[1];
    int[] xmlSchemaDocumentsPropertiesLengths = new int[1];
    InputStream xmlSchemaProperties;
    int xmlSchemaPropertiesLength;
    //Set the parameter values
    xmlSchemaLocations[0] = "";
    FileInputStream fi = null;
    xmlSchemaNameQualifiers[0] = "SYSXSR";
    xmlSchemaNames[0] = schemaName;
    try {
        fi = new FileInputStream("customer.xsd");
        xmlSchemaDocuments[0] = new BufferedInputStream(fi);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    try {
        xmlSchemaDocumentsLengths[0] = (int) fi.getChannel().size();
        System.out.println(xmlSchemaDocumentsLengths[0]);
    } catch (IOException e1) {
        e1.printStackTrace();
    }
    xmlSchemaDocumentsProperties[0] = null;
    xmlSchemaDocumentsPropertiesLengths[0] = 0;
    xmlSchemaProperties = null;
    xmlSchemaPropertiesLength = 0;
    DB2Connection ds = (DB2Connection) con;
    // Invoke registerDB2XmlSchema
    ds.registerDB2XmlSchema(
        xmlSchemaNameQualifiers,
        xmlSchemaNames,
        xmlSchemaLocations,
        xmlSchemaDocuments,
        xmlSchemaDocumentsLengths,
        xmlSchemaDocumentsProperties,
        xmlSchemaDocumentsPropertiesLengths,
        xmlSchemaProperties,
        xmlSchemaPropertiesLength,
        false);
}

```

Figure 6. Example of registration of an XML schema with DB2 using an XML document from an input stream

Example:Removal of an XML schema: The following example demonstrates the use of `deregisterDB2XmlObject` to remove an XML schema from DB2. The SQL schema name for the registered schema is SYSXSR.

```

public static void deregisterSchema(
    Connection con,
    String schemaName)
    throws SQLException {
    // Define and assign values to the deregisterDB2XmlObject parameters
    String xmlSchemaNameQualifier = "SYSXSR";
    String xmlSchemaName = schemaName;
    DB2Connection ds = (DB2Connection) con;
    // Invoke deregisterDB2XmlObject
    ds.deregisterDB2XmlObject(
        xmlSchemaNameQualifier,
        xmlSchemaName);
}

```

Figure 7. Example of removal of an XML schema from DB2

Related concepts:

- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133
- “XSR object registration” on page 134

Altering registered XSR objects

The XML schema repository is used to manage dependencies of XML documents on XML schemas, DTDs or other external entities. Each of these XML schemas, DTDs or external entities first must be registered as a new XSR object in the XML schema repository.

Once registered in the XML schema repository, XSR objects can be altered to enable or disable decomposition, dropped, or associated with a comment. Additionally, usage privileges can be granted or revoked for registered XSR objects.

Related reference:

- “ALTER XSROBJECT statement” in *SQL Reference, Volume 2*
- “COMMENT statement” in *SQL Reference, Volume 2*
- “DROP statement” in *SQL Reference, Volume 2*
- “GRANT (XSR object privileges) statement” in *SQL Reference, Volume 2*
- “REVOKE (XSR object privileges) statement” in *SQL Reference, Volume 2*

Examples of extracting XML schema information

Listing XML schemas registered with the XSR

The following examples show how XML schemas that have been fully registered with the XML schema repository can be queried for through SQL statements. Registration must have been completed before an XML schema is fully registered.

Example 1: List all registered XML schemas

This example returns the SQL schema and the SQL identifier of all XML schemas registered with the XSR.

```

SELECT XSROBJECTNAME, XSROBJECTSCHEMA
   FROM SYSCAT.XSROBJECTS
  WHERE TYPE='S' AND STATUS='C'

```

Example 2: Return the target namespace and the schema location

This example returns the uniform resource identifiers (URIs) of the target namespaces and schema locations for all registered XML schemas (*targetNamespace* and *schemaLocation*).

```
SELECT TARGETNAMESPACE, SCHEMALOCATION
FROM SYSCAT.XSROBJECTS
WHERE TYPE='S' AND STATUS='C'
```

Example 3: Return the object information document

This example returns the object information document for all registered schemas (*schemaInfo*). This XML document is generated during schema registration and describes each XML schema document that is part of an XML schema registered with the XSR.

```
SELECT OBJECTINFO
FROM SYSCAT.XSROBJECTS
WHERE TYPE='S' AND STATUS='C'
```

Related concepts:

- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133
- “XSR objects” on page 133

Retrieving all components of an XML schema registered with the XSR

The following example shows how all the component XML schema documents that make up a registered XML schema can be retrieved from the XML schema repository.

Example 1: Return the component XML schema documents of a registered XML schema, along with the target namespace and schema location (*targetNamespace* and *schemaLocation*):

```
SELECT COMPONENT, TARGETNAMESPACE, SCHEMALOCATION
FROM SYSCAT.XSROBJECTCOMPONENTS
WHERE XSROBJECTSCHEMA = ? AND XSROBJECTNAME = ?
```

Related concepts:

- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133
- “XSR objects” on page 133

Retrieving the XML schema of an XML document

The following example shows how the XML schema associated with an XML document can be retrieved from the XML schema repository.

Example 1: Retrieve the XML schema of an XML document:

```
SELECT DOC, XMLXSROBJECTID(DOC)
FROM T
```

Related concepts:

- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133
- “XSR objects” on page 133

Stored procedures

XSR_REGISTER procedure

```

▶▶ XSR_REGISTER (—rschema—, —name—, —schemalocation—, —content—, —
—docproperty—)

```

The schema is SYSPROC.

The XSR_REGISTER procedure is the first stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR).

Authorization:

The user ID of the caller of the procedure must have at least one of the following:

- SYSADM or DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema

An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name

An input and output argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation

An input argument of type VARCHAR (1000), which can have a NULL value, that indicates the schema location of the primary XML schema document. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the `xsi:schemaLocation` attribute.

content

An input parameter of type BLOB (30M) that contains the content of the primary XML schema document. This argument cannot have a NULL value; an XML schema document must be supplied.

docproperty

An input parameter of type BLOB (5M) that indicates the properties for the primary XML schema document. This parameter can have a NULL value; otherwise, the value is an XML document.

Example:

```
CALL SYSPROC.XSR_REGISTER(  
  'user1',  
  'POschema',  
  'http://myPOschema/PO.xsd',  
  :content_host_var,  
  :docproperty_host_var)
```

XSR_ADDSCHEMADOC procedure

```
►► XSR_ADDSCHEMADOC ( (rschema, name, schemalocation, content, docproperty) )
```

The schema is SYSPROC.

Each XML schema in the XML schema repository (XSR) can consist of one or more XML schema documents. Where an XML schema consists of multiple documents, the XSR_ADDSCHEMADOC stored procedure is used to add every XML schema other than the primary XML schema document.

Authorization:

The user ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

rschema

An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name

An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER stored procedure, and XML schema registration cannot yet be completed. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation

An input argument of type VARCHAR (1000), which can have a NULL value, that indicates the schema location of the primary XML schema document to which the XML schema document is being added. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

content

An input parameter of type BLOB (30M) that contains the content of the XML schema document being added. This argument cannot have a NULL value; an XML schema document must be supplied.

docproperty

An input parameter of type BLOB (5M) that indicates the properties for the XML schema document being added. This parameter can have a NULL value; otherwise, the value is an XML document.

Example:

```
CALL SYSPROC.XSR_ADDSCHEMADOC(  
  'user1',  
  'POschema',  
  'http://myPOschema/address.xsd',  
  :content_host_var,  
  0)
```

XSR_COMPLETE procedure

►►XSR_COMPLETE—(—*rschema*—,—*name*—,—*schemaproperties*—,——————►
►—*isusedfordecomposition*—)—————►◄

The schema is SYSPROC.

The XSR_COMPLETE procedure is the final stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR). An XML schema is not available for validation until the schema registration completes through a call to this stored procedure.

Authorization::

The user ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

rschema

An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name

An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema, for which a completion check is to be performed, is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER stored procedure, and XML schema registration cannot yet be completed. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemaproperties

An input argument of type BLOB (5M) that specifies properties, if any, associated with the XML schema. The value for this argument is either NULL, if there are no associated properties, or an XML document representing the properties for the XML schema.

isusedfordecomposition

An input parameter of type integer that indicates if an XML schema is to be used for decomposition. If an XML schema is to be used for decomposition, this value should be set to 1; otherwise, it should be set to zero.

Example:

```
CALL SYSPROC.XSR_COMPLETE(  
  'user1',  
  'POschema',  
  :schemaproperty_host_var,  
  0)
```

XSR_DTD procedure

►► XSR_DTD (—*rschema*—, —*name*—, —*systemid*—, —*publicid*—, —*content*—) ◀◀

The schema is SYSPROC.

The XSR_DTD procedure registers a document type declaration (DTD) with the XML schema repository (XSR).

Authorization:

The user ID of the caller of the procedure must have at least one of the following:

- SYSADM or DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema

An input argument of type VARCHAR (128) that specifies the SQL schema for the DTD. The SQL schema is one part of the SQL identifier used to identify this DTD in the XSR. (The other part of the SQL identifier is supplied by the *name* argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT_SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name

An input and output argument of type VARCHAR (128) that specifies the name of the DTD. The complete SQL identifier for the DTD is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

systemid

An input parameter of type VARCHAR (1000) that specifies the system identifier of the DTD. The system ID of the DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a NULL value. The system ID can be specified together with a public ID.

publicid

An input parameter of type VARCHAR (1000) that specifies the public identifier of the DTD. The public ID of a DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a NULL value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

content

An input parameter of type BLOB (30M) that contains the content of the DTD document. This argument cannot have a NULL value.

Example: Register the DTD identified by the system ID *http://www.test.com/person.dtd* and public ID *http://www.test.com/person*:

```
CALL SYSPROC.XSR_DTD ( 'MYDEPT' ,  
  'PERSONDTD' ,  
  'http://www.test.com/person.dtd' ,  
  'http://www.test.com/person' ,  
  :content_host_variable  
)
```

Related concepts:

- “Registering XSR objects through stored procedures” on page 135
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133

XSR_EXTENTITY procedure

►► XSR_EXTENTITY (—*rschema*—, —*name*—, —*systemid*—, —*publicid*—, —————►
►—*content*—)—————►►

The schema is SYSPROC.

The XSR_EXTENTITY procedure registers an external entity with the XML schema repository (XSR).

Authorization:

The user ID of the caller of the procedure must have at least one of the following:

- SYSADM or DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema

An input argument of type VARCHAR (128) that specifies the SQL schema for the external entity. The SQL schema is one part of the SQL identifier used to identify this external entity in the XSR. (The other part of the SQL identifier is supplied by the *name* argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name

An input and output argument of type VARCHAR (128) that specifies the name of the external entity. The complete SQL identifier for the external entity is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

systemid

An input parameter of type VARCHAR (1000) that specifies the system identifier of the external entity. The system ID of the external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a NULL value. The system ID can be specified together with a public ID.

publicid

An input parameter of type VARCHAR (1000) that specifies the public identifier of the external entity. The public ID of a external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a NULL value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

content

An input parameter of type BLOB (30M) that contains the content of the external entity document. This argument cannot have a NULL value.

Example: Register the external entites identified by the system identifiers <http://www.test.com/food/chocolate.txt> and <http://www.test.com/food/cookie.txt>:

```
CALL SYSPROC.XSR_EXTENTITY ( 'FOOD' ,
    'CHOCLATE' ,
    'http://www.test.com/food/chocolate.txt' ,
    NULL ,
    :content_of_chocolate.txt_as_a_host_variable
)

CALL SYSPROC.XSR_EXTENTITY ( 'FOOD' ,
    'COOKIE' ,
```

```
'http://www.test.com/food/cookie.txt' ,
NULL ,
:content_of_cookie.txt_as_a_host_variable
)
```

Related concepts:

- “Registering XSR objects through stored procedures” on page 135
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133

Commands

REGISTER XMLSCHEMA

Registers an XML schema with the XML schema repository (XSR).

Authorization:

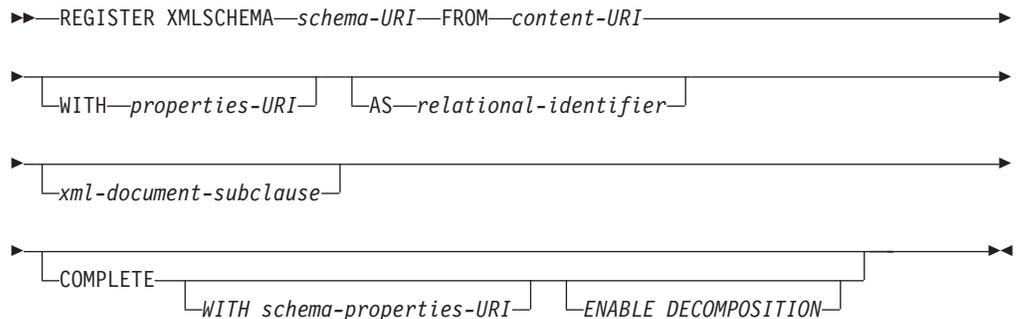
One of the following:

- SYSADM or DBADM
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist
- CREATEIN privilege if the SQL schema exists

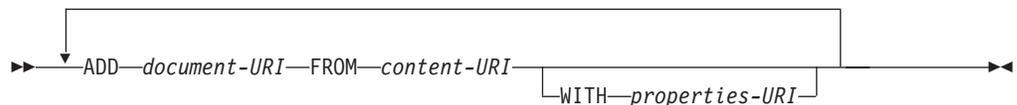
Required connection:

Database

Command syntax:



xml-document-subclause:



Description:

schema-URI

Specifies the URI, as referenced by XML instance documents, of the XML schema being registered.

FROM *content-URI*

Specifies the URI where the XML schema document is located. Only a local file specified by a file scheme URI is supported.

WITH *properties-URI*

Specifies the URI of a properties document for the XML schema. Only a local file specified by a file scheme URI is supported.

AS *relational-identifier*

Specifies a name that can be used to refer to the XML schema being registered. The relational name can be specified as a two-part SQL identifier, consisting of the SQL schema and the XML schema name, having the following format: SQLschema.name. The default relational schema, as defined in the CURRENT SCHEMA special register, is used if no schema is specified. If no name is provided, a unique value is generated.

COMPLETE

Indicates that there are no more XML schema documents to be added. If specified, the schema is validated and marked as usable if no errors are found.

WITH *schema-properties-URI*

Specifies the URI of a properties document for the XML schema. Only a local file specified by a file scheme URI is supported.

ENABLE DECOMPOSITION

Specifies that this schema is to be used for decomposing XML documents.

ADD *document-URI*

Specifies the URI of an XML schema document to be added to this schema, as the document would be referenced from another XML document.

FROM *content-URI*

Specifies the URI where the XML schema document is located. Only a local file specified by a file scheme URI is supported.

WITH *properties-URI*

Specifies the URI of a properties document for the XML schema. Only a local file specified by a file scheme URI is supported.

Example:

```
REGISTER XMLSCHEMA 'http://myPOschema/PO.xsd'  
FROM 'file:///c:/TEMP/PO.xsd'  
WITH 'file:///c:/TEMP/schemaProp.xml'  
AS user1.POschema
```

Usage notes:

- Before an XML schema document can be referenced and be available for validation and annotation, it must first be registered with the XSR. This command performs the first step of the XML schema registration process, by registering the primary XML schema document. The final step of the XML schema registration process requires that the COMPLETE XMLSCHEMA command run successfully for the XML schema. Alternatively, if there are no other XML schema documents to be included, issue the REGISTER XMLSCHEMA command with the COMPLETE keyword to complete registration in one step.
- When registering an XML schema in the database, a larger application heap (APPLHEAPSZ) may be required depending on the size of the XML schema. The recommended size is 1024 but larger schemas will require additional memory.

Related reference:

- “ADD XMLSCHEMA DOCUMENT ” on page 149
- “COMPLETE XMLSCHEMA ” on page 150

ADD XMLSCHEMA DOCUMENT

Adds one or more XML schema documents to an existing but incomplete XML schema before completing registration.

Authorization:

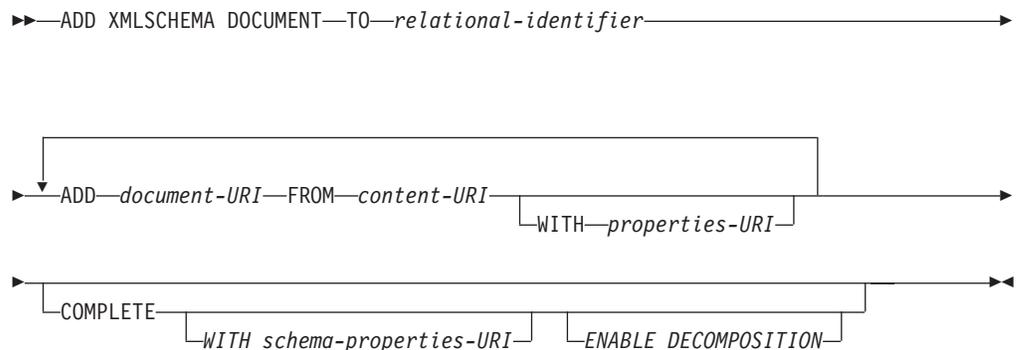
The following authority is required:

- The user ID must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

Required connection:

Database

Command syntax:



Description:

TO *relational-identifier*

Specifies the relational name of a registered but incomplete XML schema to which additional schema documents are added.

ADD *document-URI*

Specifies the uniform resource identifier (URI) of an XML schema document to be added to this schema, as the document would be referenced from another XML document.

FROM *content-URI*

Specifies the URI where the XML schema document is located. Only a file scheme URI is supported.

WITH *properties-URI*

Specifies the URI of a properties document for the XML schema. Only a file scheme URI is supported.

COMPLETE

Indicates that there are no more XML schema documents to be added. If specified, the schema is validated and marked as usable if no errors are found.

WITH *schema-properties-URI*

Specifies the URI of a properties document for the XML schema. Only a file scheme URI is supported.

ENABLE DECOMPOSITION

Specifies that this schema is to be used for decomposing XML documents.

Example:

```
ADD XMLSCHEMA DOCUMENT TO JOHNDOE.PRODSHEMA
  ADD 'http://myPOschema/address.xsd'
  FROM 'file:///c:/TEMP/address.xsd'
```

Related reference:

- “COMPLETE XMLSCHEMA ” on page 150
- “REGISTER XMLSCHEMA ” on page 147

COMPLETE XMLSCHEMA

This command completes the process of registering an XML schema in the XML schema repository (XSR).

Authorization:

- The user ID must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

Required connection:

Database

Command syntax:

```
▶▶ COMPLETE XMLSCHEMA relational-identifier
                                     └─ WITH schema-properties-URI ─┘
▶ ───────────────────────────────────────────────────────────────────────────▶
└─ ENABLE DECOMPOSITION ─┘
```

Description:*relational-identifier*

Specifies the relational name of an XML schema previously registered with the REGISTER XMLSCHEMA command. The relational name can be specified as a two-part SQL identifier, consisting of the SQL schema and the XML schema name, having the following format: *SQLschema.name*. The default SQL schema, as defined in the CURRENT SCHEMA special register, is used if no schema is specified.

WITH *schema-properties-URI*

Specifies the uniform resource identifier (URI) of a properties document for the XML schema. Only a local file, specified by a file scheme URI, is supported. A schema property document can only be specified during the completion stage of XML schema registration.

ENABLE DECOMPOSITION

Indicates that the schema can be used for decomposing XML instance documents.

Example:

```
COMPLETE XMLSCHEMA user1.P0schema WITH 'file:///c:/TEMP/schemaProp.xml'
```

Usage notes:

An XML schema cannot be referenced or used for validation or annotation until the XML schema registration process has been completed. This command completes the XML schema registration process for an XML schema that was begun with the REGISTER XMLSCHEMA command.

Related reference:

- “ADD XMLSCHEMA DOCUMENT ” on page 149
- “REGISTER XMLSCHEMA ” on page 147

REGISTER XSROBJECT

Registers an XML object in the database catalogs. Supported objects are DTDs and external entities.

Authorization:

One of the following:

- SYSADM or DBADM
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist
- CREATEIN privilege if the SQL schema exists

Required connection:

Database

Command syntax:

```
►►—REGISTER XSROBJECT—system-ID—PUBLIC—public-id—FROM—content-URI—►  
  
►—AS—relational-identifier—DTD—EXTERNAL ENTITY—►
```

Command parameters:

system-id

Specifies the system ID that is specified in the XML object declaration.

PUBLIC *public-id*

Specifies an optional PUBLIC ID in the XML object declaration.

FROM *content-URI*

Specifies the URI where the content of an XML schema document is located. Only a local file specified by a file scheme URI is supported.

AS *relational-identifier*

Specifies a name that can be used to refer to the XML object being

registered. The relational name can be specified as a two-part SQL identifier consisting of the relational schema and name separated by a period, for example "JOHNDOE.EMPLOYEEEDTD". If no relational schema is specified, the default relational schema defined in the special register CURRENT SCHEMA is used. If no name is specified, one is generated automatically.

DTD Specifies that the object being registered is a Data Type Definition document (DTD).

EXTERNAL ENTITY

Specifies that the object being registered is an external entity.

Examples:

1. Given this sample XML document which references an external entity:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE copyright [
  <!ELEMENT copyright (#PCDATA)>
  <!ENTITY c SYSTEM "http://www.xmlwriter.net/copyright.xml">
]>
<copyright>&c;</copyright>
```

Before this document can be successfully inserted into an XML column, the external entity needs to be registered. The following command registers an entity where the entity content is stored locally in C:\TEMP:

```
REGISTER XSROBJECT 'http://www.xmlwriter.net/copyright.xml'
FROM 'c:\temp\copyright.xml' EXTERNAL ENTITY
```

2. Given this XML document fragment which references a DTD:

```
<!--inform the XML processor
that an external DTD is referenced-->
<?xml version="1.0" standalone="no" ?>

<!--define the location of the
external DTD using a relative URL address-->
<!DOCTYPE document SYSTEM "http://www.xmlwriter.net/subjects.dtd">

<document>
  <title>Subjects available in Mechanical Engineering.</title>
  <subjectID>2.303</subjectID>
  <subjectname>Fluid Mechanics</subjectname>
  ...
```

Before this document can be successfully inserted into an XML column, the DTD needs to be registered. The following command registers a DTD where the DTD definition is stored locally in C:\TEMP and the relational identifier to be associated with the DTD is "TEST.SUBJECTS":

```
REGISTER XSROBJECT 'http://www.xmlwriter.net/subjects.dtd'
FROM 'file:///c:/temp/subjects.dtd' AS TEST.SUBJECTS DTD
```

3. Given this sample XML document which references a public external entity:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE copyright [
  <!ELEMENT copyright (#PCDATA)>
  <!ENTITY c PUBLIC "-//W3C//TEXT copyright//EN"
    "http://www.w3.org/xmlspec/copyright.xml">
]>
<copyright>&c;</copyright>
```

Before this document can be successfully inserted into an XML column, the public external entity needs to be registered. The following command registers an entity where the entity content is stored locally in C:\TEMP:

```
REGISTER XSROBJECT 'http://www.w3.org/xmlspec/copyright.xml'
PUBLIC "-//W3C//TEXT copyright//EN' FROM 'file:///c:/temp/copyright.xml'
EXTERNAL ENTITY
```

Related concepts:

- “XSR object registration” on page 134
- “XSR objects” on page 133

Related reference:

- “REGISTER XMLSCHEMA ” on page 147

Chapter 6. Data movement utility support

XML data movement overview

With the introduction of an XML column type, support for XML data has been added to the import and export utilities.

Importing XML data:

The import utility can be used to insert XML documents into a regular relational table. Only well-formed XML documents can be imported.

Use the XML FROM option of the IMPORT command to specify the location of the XML documents to import. The XMLVALIDATE option specifies how imported documents should be validated. You can select to have the imported XML data validated against a schema specified with the IMPORT command, against a schema identified by a schema location hint inside of the source XML document, or by the schema identified by the XML Data Specifier in the main data file. You can also use the XMLPARSE option to specify how whitespace should be handled when the XML document is imported. The xmlchar and xmlgraphic file type modifiers allow you to specify the encoding characteristics for the imported XML data.

Exporting XML data:

Data may be exported from tables that include one or more columns with an XML data type. Exported XML data is stored in files separate from the main data file containing the exported relational data. Information about each exported XML document is represented in the main exported data file by an XML data specifier (XDS). The XDS is a string that specifies the name of the system file in which the XML document is stored, the exact location and length of the XML document inside of this file, and the XML schema used to validate the XML document.

You can use the XMLFILE, XML TO, and XMLSAVESHEMA parameters of the EXPORT command to specify details about how exported XML documents are stored. The xmlinsefiles, xmlnodeclaration, xmlchar, and xmlgraphic file type modifiers allow you to specify further details about the storage location and the encoding of the exported XML data.

Related concepts:

- “Exporting XML data” on page 159
- “Importing XML data” on page 162
- “Native XML data store overview” on page 1

Related reference:

- “EXPORT command” in *Command Reference*
- “IMPORT Command” in *Command Reference*

Important considerations for XML data movement

Following are a number of factors to keep in mind when importing or exporting XML data:

- Exported XML data is always stored separately from the main data file containing exported relational data.
- By default, the export utility writes XML data in Unicode. You can use the XMLCHAR file type modifier to have XML data written in the character code page. The XMLGRAPHIC file type modifier specifies that XML data is written in the graphic code page, which is UTF-16 regardless of the application code page.
- For the import utility, unless the XML document to import contains a declaration tag that includes an encoding attribute, this document is assumed to be in Unicode. You can use the XMLCHAR file type modifier to indicate that XML documents to import are encoded in the character code page, while the XMLGRAPHIC file type modifier indicates that XML documents to import are encoded in UTF-16.
- For the import utility, rows which contain documents that are not well-formed will be rejected.
- If the XMLVALIDATE option is specified for the import utility, documents which successfully validate against their matching schema will be annotated with the schema information as they are inserted into a table. Rows containing documents that fail to validate against their matching schema will be rejected.
- You can use the export utility with an XQuery specification to export XQuery Data Model (QDM) instances that are not well-formed XML documents. However, exported XML documents that are not well-formed cannot be imported directly into an XML column, since columns defined with the XML data type can contain only complete XML documents.

Related concepts:

- “Exporting XML data” on page 159
- “Importing XML data” on page 162
- “XML data movement overview” on page 155

XQuery data model

XML data can be accessed in a database table either by use of the XQuery functions available in SQL, or by invoking XQuery directly. An instance of the XQuery data model (QDM) can be a well-formed XML document, a sequence of nodes, a sequence of atomic values, or any combination of nodes and atomic values.

Individual QDM instances can be written to one or more XML files by means of the EXPORT command.

Related concepts:

- “XML data movement overview” on page 155
- “XML data type” on page 17

LOB and XML file behavior with regard to import and export

LOB and XML files have certain shared behaviors with regard to the import and export utilities.

When exporting data, if one or more LOB paths are specified with the LOBS TO option, the export utility will cycle between the paths to write each successful LOB value to the appropriate LOB file. Similarly, if one or more XML paths are specified with the XML TO option, the export utility will cycle between the paths to write each successive QDM (XQuery Data Model) instance to the appropriate XML file. By default, LOB values and QDM instances are written to the same path to which the exported relational data is written. Unless the LOBSINSEPFILLES or XMLINSEPFILLES file type modifier is set, both LOB files and XML files can have multiple values concatenated to the same file.

The LOBFILE option provides a means to specify the base name of the LOB files generated by the export utility. Similarly, the XMLFILE option provides a means to specify the base name of the XML files generated by the export utility. The default LOB file base name is the name of the exported data file, with the extension .lob. The default XML file base name is the name of the exported data file, with the extension .xml. The full name of the exported LOB file or XML file therefore consists of the base name, followed by a number extension that is padded to three digits, and the extension .lob or .xml.

When importing data, a LOB Location Specifier (LLS) is compatible with an XML target column, and an XML Data Specifier (XDS) is compatible with a LOB target column. If the LOBS FROM option is not specified, the LOB files to import are assumed to reside in the same path as the input relational data file. Similarly, if the XML FROM option is not specified, the XML files to import are assumed to reside in the same path as the input relational data file.

Example 1:

For the following EXPORT command:

```
EXPORT TO /mypath/t1export.del OF DEL MODIFIED BY LOBSINFILE
SELECT * FROM USER.T1
```

All LOB values are written to the file "/mypath/t1export.del.001.lob", and all QDM instances are written to the file "/mypath/t1export.del.001.xml".

Example 2:

For the following EXPORT command:

```
EXPORT TO /mypath/t1export.del OF DEL LOBS TO /lob1,/lob2
MODIFIED BY LOBSINFILE SELECT * FROM USER.T1
```

The first LOB value will be written to the file "/lob1/t1export.del.001.lob", the second will be written to the file "/lob2/t1export.del.002.lob", the third will be appended to "/lob1/t1export.del.001.lob", the fourth will be appended to "/lob2/t1export.del.002.lob", and so on.

Example 3:

For the following EXPORT command:

```
EXPORT TO /mypath/t1export.del OF DEL XML TO /xml1,/xml2 XMLFILE xmlbase
MODIFIED BY XMLINSEPFILS SELECT * FROM USER.T1
```

The first QDM instance will be written to the file `"/xml1/xmlbase.001.xml"`, the second will be written to the file `"/xml2/xmlbase.002.xml"`, the third will be written to `"/xml1/xmlbase.003.xml"`, the fourth will be written to `"/xml2/xmlbase.004.xml"`, and so on.

Example 4:

For a table `"mytable"` that contains a single XML column, and the following IMPORT command:

```
IMPORT FROM myfile.del of del LOBS FROM /lobpath XML FROM /xmlpath
MODIFIED BY LOBSINFILE XMLCHAR replace into mytable
```

If `"myfile.del"` contains the following data:

```
mylobfile.001.lob.123.456/
```

The import utility will try to import an XML document from the file `"/lobpath/mylobfile.001.lob"`, starting at file offset 123, with its length being 456 bytes.

The file `"mylobfile.001.lob"` is assumed to be in the LOB path, as opposed to the XML path, since the value is referred to by a LOB Location Specifier (LLS) instead of an XML Data Specifier (XDS).

The document is assumed to be encoded in the character codepage, since the XMLCHAR file type modifier is specified.

Related concepts:

- "XML data type" on page 17
- "Export Overview" in *Data Movement Utilities Guide and Reference*
- "Exporting XML data" on page 159
- "Import Overview" in *Data Movement Utilities Guide and Reference*
- "Importing XML data" on page 162

Related tasks:

- "Exporting data" in *Data Movement Utilities Guide and Reference*
- "Importing data" in *Data Movement Utilities Guide and Reference*

XML data specifier

XML data involved in the export and import utilities must be stored in files separate from the main data file. The XML data, however, is represented in the main data file with an XML data specifier (XDS). The XDS is a string represented as an XML tag named `"XDS"`, which has attributes that describe information about the actual XML data in the column; such information includes the name of the file that contains the actual XML data, and the offset and length of the XML data within that file. The attributes of the XDS are described below.

FIL The name of the file that contains the XML data.

OFF The byte offset of the XML data in the file named by the FIL attribute, where the offset begins from 0.

- LEN** The length in bytes of the XML data in the file named by the FIL attribute.
- SCH** The fully qualified SQL identifier of the XML schema that is used to validate this XML document. The schema and name components of the SQL identifier are stored as the "OBJECTSCHEMA" and "OBJECTNAME" values, respectively, of the row in the SYSCAT.XSROBJECTS catalog table that corresponds to this XML schema.

The XDS is interpreted as a character field in the data file and is subject to the file format's parsing behavior for character columns. For the delimited ASCII file format (DEL), for example, if the character delimiter is present in the XDS, it must be doubled. The special characters (<, >, &, ', ") within the attribute values must always be escaped. Consider a FIL attribute with the value: abc&"def".del. To include this in a delimited ASCII file, where the character delimiter is the " character, this XDS would be included as follows: <XDS FIL="abc&"def" &"def".del" /> where the " characters are doubled and special characters are escaped.

Example:

The following is an example of an XDS as it would appear in a delimited ASCII data file.

```
"<XDS FIL = "xmldocs.xml.001" OFF="100" LEN="300" />"
```

This entry indicates that the XML data is stored in the file xmldocs.xml.001 beginning at byte offset 100 with a length of 300 bytes. (Because this XDS is within an ASCII file delimited with double quotation marks, the double quotation marks within the XDS tag itself must be doubled.)

Related concepts:

- "Export Overview" in *Data Movement Utilities Guide and Reference*
- "Exporting XML data" on page 159
- "Import Overview" in *Data Movement Utilities Guide and Reference*
- "Importing XML data" on page 162

Related tasks:

- "Exporting data" in *Data Movement Utilities Guide and Reference*
- "Importing data" in *Data Movement Utilities Guide and Reference*

Exporting XML data

When exporting XML data, the resulting QDM (XQuery Data Model) instances are written to a file or files separate from the main data file containing exported relational data. This is true even if neither the XMLFILE nor the XML TO option is specified. By default, exported QDM instances are all concatenated to the same XML file. You can use the XMLINSEPFILES file type modifier to specify that each QDM instance be written to a separate file.

The destination paths and base names of the exported XML files can be specified with the XML TO and XMLFILE options. By default, exported XML files are written to the path of the exported data file. The default base name for exported XML files is the name of the exported data file, with an appending 3-digit sequence number, and the .xml extension.

Examples:

For the following examples, imagine a table USER.T1 containing four columns and two rows:

C1 INTEGER
C2 XML
C3 VARCHAR(10)
C4 XML

Table 30. USER.T1

C1	C2	C3	C4
2	<?xml version="1.0" encoding="UTF-8" ?><note time="12:00:00"><to>You</to><from> Me</from><heading>note1</heading><body>Hello World!</body></note>	'char1'	<?xml version="1.0" encoding="UTF-8" ?><note time="13:00:00"><to>Him</to><from> Her</from><heading>note2</heading><body>Hello World!</body></note>
4	NULL	'char2'	<?xml version="1.0" encoding="UTF-8" ?><note time="14:00:00"><to>Us</to><from> Them</from><heading>note3</heading><body>Hello World!</body></note>

Example 1:

The following command exports the contents of USER.T1 in Delimited ASCII (DEL) format to the file "/mypath/t1export.del". Because the XML TO and XMLFILE options are not specified, the XML documents contained in columns C2 and C4 are written to the same path as the main exported file "/mypath". The base name for these files is "t1export.del.xml". The XMLSAVESHEMA option indicates that XML schema information is saved during the export procedure.

```
EXPORT TO /mypath/t1export.del OF DEL XMLSAVESHEMA SELECT * FROM USER.T1
```

The exported file "/mypath/t1export.del" contains:

```
2,"<XDS FIL='t1export.del.001.xml' OFF='0' LEN='144' />","char1",  
"<XDS FIL='t1export.del.001.xml' OFF='144' LEN='145' />"  
4,,"char2","<XDS FIL='t1export.del.001.xml' OFF='289'  
LEN='145' SCH='S1.SCHEMA_A' />"
```

The exported XML file "/mypath/t1export.del.001.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="12:00:00"><to>You</to>  
<from>Me</from><heading>note1</heading><body>Hello World!</body>  
</note><?xml version="1.0" encoding="UTF-8" ?><note time="13:00:00"><to>Him  
</to><from>Her</from><heading>note2</heading><body>Hello World!  
</body></note><?xml version="1.0" encoding="UTF-8" ?><note time="14:00:00">  
<to>Us</to><from>Them</from><heading>note3</heading><body>  
Hello World!</body></note>
```

Example 2:

The following command exports the contents of USER.T1 in DEL format to the file "t1export.del". XML documents contained in columns C2 and C4 are written to the path "/home/user/xmlpath". The XML files are named with the base name

"xmldocs", with multiple exported XML documents written to the same XML file. The XMLSAVESCHEMA option indicates that XML schema information is saved during the export procedure.

```
EXPORT TO /mypath/t1export.del OF DEL XML TO /home/user/xmlpath
XMLFILE xmldocs XMLSAVESCHEMA SELECT * FROM USER.T1
```

The exported DEL file "/home/user/t1export.del" contains:

```
2,"<XDS FIL='xmldocs.001.xml' OFF='0' LEN='144' />","char1",
"<XDS FIL='xmldocs.001.xml' OFF='144' LEN='145' />"
4,,"char2","<XDS FIL='xmldocs.001.xml' OFF='289'
LEN='145' SCH='S1.SCHEMA_A' />"
```

The exported XML file "/home/user/xmlpath/xmldocs.001.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="12:00:00"><to>You</to>
<from>Me</from><heading>note1</heading><body>Hello World!</body>
</note><?xml version="1.0" encoding="UTF-8" ?><note time="13:00:00">
<to>Him</to><from>Her</from><heading>note2</heading><body>
Hello World!</body></note><?xml version="1.0" encoding="UTF-8" ?>
<note time="14:00:00"><to>Us</to><from>Them</from><heading>
note3</heading><body>Hello World!</body></note>
```

Example 3:

The following command is similar to Example 2, except that each exported XML document is written to a separate XML file.

```
EXPORT TO /mypath/t1export.del OF DEL XML TO /home/user/xmlpath
XMLFILE xmldocs MODIFIED BY XMLINSEPFILS XMLSAVESCHEMA
SELECT * FROM USER.T1
```

The exported file "/mypath/t1export.del" contains:

```
2,"<XDS FIL='xmldocs.001.xml' />","char1","<XDS FIL='xmldocs.002.xml' />"
4,,"char2","<XDS FIL='xmldocs.004.xml' SCH='S1.SCHEMA_A' />"
```

The exported XML file "/home/user/xmlpath/xmldocs.001.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="12:00:00"><to>You</to>
<from>Me</from><heading>note1</heading><body>Hello World!</body>
</note>
```

The exported XML file "/home/user/xmlpath/xmldocs.002.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="13:00:00"><to>Him</to>
<from>Her</from><heading>note2</heading><body>Hello World!</body>
</note>
```

The exported XML file "/home/user/xmlpath/xmldocs.004.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="14:00:00"><to>Us</to>
<from>Them</from><heading>note3</heading><body>Hello World!</body>
</note>
```

Example 4:

The following command writes the result of an XQuery to an XML file.

```
EXPORT TO /mypath/t1export.del OF DEL XML TO /home/user/xmlpath
XMLFILE xmldocs MODIFIED BY XMLNODEDECLARATION select
xmlquery( '$m/note/from/text()' passing by ref c4 as "m" returning sequence)
from USER.T1
```

The exported DEL file "/mypath/t1export.del" contains:

```
"<XDS FIL='xmldocs.001.xml' OFF='0' LEN='3' />"
"<XDS FIL='xmldocs.001.xml' OFF='3' LEN='4' />"
```

The exported XML file `"/home/user/xmlpath/xmldocs.001.xml"` contains:

```
HerThem
```

Note: The result of this particular XQuery does not produce well-formed XML documents. Therefore, the file exported above could not be directly imported into an XML column.

Related concepts:

- “Native XML data store overview” on page 1
- “Export Overview” in *Data Movement Utilities Guide and Reference*
- “Importing XML data” on page 162
- “XML data specifier” on page 158

Related reference:

- “LOB and XML file behavior with regard to import and export” on page 157

Importing XML data

When importing data into an XML table column, you can use the XML FROM option to specify the paths of the input XML data file or files. For example, For an XML file `"/home/user/xmlpath/xmldocs.001.xml"` that had previously been exported, the following command could be used to import the data back into the table.

```
IMPORT FROM tlexport.del OF DEL XML FROM /home/user/xmlpath INSERT INTO USER.T1
```

Validating Inserted Documents Against Schemas:

The XMLVALIDATE option allows XML documents to be validated against XML schemas as they are imported. In the following example, incoming XML documents are validated against schema information that was saved when the XML documents were exported:

```
IMPORT FROM tlexport.del OF DEL XML FROM /home/user/xmlpath XMLVALIDATE
USING XDS INSERT INTO USER.T1
```

Specifying Parse Options:

You can use the XMLPARSE option to specify whether whitespace in the imported XML documents is preserved or stripped. In the following example, all imported XML documents are validated against XML schema information that was saved when the XML documents were exported, and these documents are parsed with whitespace preserved.

```
IMPORT FROM tlexport.del OF DEL XML FROM /home/user/xmlpath XMLPARSE PRESERVE
WHITESPACE XMLVALIDATE USING XDS INSERT INTO USER.T1
```

Related concepts:

- “Exporting XML data” on page 159
- “Native XML data store overview” on page 1

Related reference:

- “LOB and XML file behavior with regard to import and export” on page 157

- “IMPORT Command” in *Command Reference*

Chapter 7. Application programming and routines support

Application programming language support for XML

You can write applications to store XML data in DB2 database tables, retrieve data from tables, or call stored procedures or user-defined functions with XML parameters. XML data can be stored only in Unicode databases.

You can use any of the following languages to write your applications:

- C or C++ (embedded SQL or DB2 CLI)
- COBOL
- Java (JDBC or SQLJ)
- C# and Visual Basic (DB2 .NET Data Provider)
- PHP

An application program can retrieve an entire document or a fragment of a document from an XML column. However, you can store only an entire document in an XML column.

Stored procedures and user-defined functions can pass XML values in input or output parameters. To call a stored procedure or user-defined function that has XML or XML AS CLOB parameters, execute a CALL statement with compatible data types.

When an application provides an XML value to a DB2 database server, the database server converts the data from the XML serialized string 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 the XML serialized string format. In addition, the database server might need to convert the output data from UTF-8 to the application encoding.

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.

When you retrieve an XML sequence, you have several choices:

- Execute an XQuery expression directly.
To execute an XQuery expression in an application, you prepend the string 'XQUERY' to the XQuery expression, and dynamically execute the resulting string.

When you execute an XQuery expression directly, the DB2 database server returns the sequence that is the result of the XQuery statement as a result table. Each row in the result table is an item in the sequence.

- Within an SQL FETCH or single-row SELECT INTO operation, call the XMLQUERY or XMLTABLE built-in functions, passing an XQuery expression as an argument.

This technique can be used with static or dynamic SQL and any application programming language. XMLQUERY is a scalar function that returns the entire sequence in an application variable. XMLTABLE is a table function that returns each item in the sequence as a row of the result table. The columns in the result table are values from the retrieved sequence item.

Parameter markers and host variables:

Parameter markers or host variables cannot be specified anywhere in an XQuery expression, including within the SQL specified in an XQuery expression. For example, the XQuery function db2-fn:sqlquery allows you to specify an SQL fullselect with an XQuery expression to extract the detailed description for a product:

```
xquery declare default element namespace "http://posample.org";
db2-fn:sqlquery("select description from product where pid='100-103-01'"
/product/description/details/text()
```

You cannot specify a parameter marker or host variable in the XQuery expression, even within the fullselect. The following expression is incorrect and unsupported (it returns SQLSTATE 42610, sqlcode -418):

```
xquery declare default element namespace "http://posample.org";
db2-fn:sqlquery("select description from product where pid=?"
/product/description/details/text()
```

In order to pass application values to XQuery expressions, use the SQL/XML functions XMLQUERY and XMLTABLE. The PASSING clause of these functions allows you to use application values during the evaluation of the XQuery expression.

The following query shows how the previous incorrect query can be rewritten using SQL/XML to achieve an equivalent result:

```
SELECT XMLQUERY ('declare default element namespace "http://posample.org";
$descdoc/product/description/details/text()' passing description as "descdoc")
FROM product
WHERE pid=?
```

Related concepts:

- “XML data handling in CLI applications - Overview” on page 167
- “DB2 .NET Data Provider” in *Developing ADO.NET and OLE DB Applications*
- “Data retrieval with XQuery” on page 70
- “Native XML data store overview” on page 1
- “XML data encoding” on page 217
- “XML data in JDBC applications” on page 177
- “XML data querying” on page 45
- “XML data type” on page 17
- “XML parsing” on page 37
- “XML serialization” on page 86
- “XML and XQuery support in SQL procedures” on page 189
- “XML data type support in external routines” on page 194
- “Introduction to PHP application development for DB2” on page 187

Related tasks:

- “Declaring XML host variables in embedded SQL applications” on page 171
- “Executing XQuery expressions in PHP (ibm_db2)” on page 188

Related reference:

- “Native XML data store roadmap” in *DB2 Information Center*
- “FETCH statement” in *SQL Reference, Volume 2*
- “Native XML - Application Development samples” on page 328
- Chapter 9, “Restrictions on native XML data store,” on page 233
- “XMLQUERY scalar function” in *SQL Reference, Volume 1*
- “XMLTABLE table function” in *SQL Reference, Volume 1*

CLI**XML data handling in CLI applications - Overview**

DB2 CLI applications can retrieve and store XML data using the SQL_XML data type. This data type corresponds to the native XML data type of the DB2 database, which is used to define columns that store well-formed XML documents. The SQL_XML type can be bound to the following C types: SQL_C_BINARY, SQL_C_CHAR, SQL_C_WCHAR, and SQL_C_DBCHAR. Using the default SQL_C_BINARY type, however, instead of character types, is recommended to avoid possible data loss or corruption resulting from code conversion when character types are used.

To store XML data in an XML column, bind a binary (SQL_C_BINARY) or character (SQL_C_CHAR, SQL_C_WCHAR, or SQL_C_DBCHAR) buffer that contains the XML value to the SQL_XML SQL type and execute the INSERT or UPDATE SQL statements. To retrieve XML data from the database, bind the result set to a binary (SQL_C_BINARY) or character (SQL_C_CHAR, SQL_C_WCHAR, or SQL_C_DBCHAR) type. Character types should be used with caution because of encoding issues.

When an XML value is retrieved into an application data buffer, the DB2 server performs an implicit serialization on the XML value to convert it from its stored hierarchical form to the serialized string form. For character typed buffers, the XML value is implicitly serialized to the application character code page associated with the character type.

By default, an XML declaration is included in the output serialized string. This default behavior can be changed by setting the SQL_ATTR_XML_DECLARATION statement or connection attribute, or by setting the XMLDeclaration CLI/ODBC configuration keyword in the db2cli.ini file.

XQuery expressions and SQL/XML functions can be issued and executed in DB2 CLI applications. SQL/XML functions are issued and executed like any other SQL statements. XQuery expressions must either be prepended with the case-insensitive keyword "XQUERY", or the SQL_ATTR_XQUERY_STATEMENT statement attribute must be set for the statement handle associated with the XQuery expression.

Related concepts:

- “Application programming language support for XML” on page 165

- “XML column inserts and updates in CLI applications” on page 168
- “XML data encoding” on page 217
- “XML data retrieval in CLI applications” on page 169
- “XML data type” on page 17

Related reference:

- “Connection attributes (CLI) list” in *Call Level Interface Guide and Reference, Volume 2*
- “Statement attributes (CLI) list” in *Call Level Interface Guide and Reference, Volume 2*
- “XMLDeclaration CLI/ODBC configuration keyword” in *Call Level Interface Guide and Reference, Volume 1*

XML column inserts and updates in CLI applications

When you update or insert data into XML columns of a table, the input data must be in the serialized string format.

For XML data, when you use `SQLBindParameter()` to bind parameter markers to input data buffers, you can specify the data type of the input data buffer as `SQL_C_BINARY`, `SQL_C_CHAR`, `SQL_C_DBCHAR` or `SQL_C_WCHAR`.

When you bind a data buffer that contains XML data as `SQL_C_BINARY`, DB2 CLI 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 when character types are used.

Important: If the XML data is encoded in an encoding scheme and CCSID other than the application code page 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`, DB2 CLI processes the XML data as externally encoded data. DB2 CLI determines the encoding of the data as follows:

- If the C type is `SQL_C_WCHAR`, DB2 CLI assumes that the data is encoded as UCS-2.
- If the C type is `SQL_C_CHAR` or `SQL_C_DBCHAR`, DB2 CLI assumes that the data is encoded in the application code page encoding scheme.

If you want the database server to implicitly parse the data before storing it in an XML column, the parameter marker data type in `SQLBindParameter()` should be specified as `SQL_XML`.

Implicit parsing is recommended, because explicit parsing of a character type with `XMLPARSE` can introduce encoding issues.

The following example shows how to update XML data in an XML column using the recommended `SQL_C_BINARY` type.

```
char xmlBuffer[10240];
integer length;

// Assume a table named dept has been created with the following statement:
// CREATE TABLE dept (id CHAR(8), deptdoc XML)
```

```

// xmlBuffer contains an internally encoded XML document that is to replace
// the existing XML document
length = strlen (xmlBuffer);
SQLPrepare (hStmt, "UPDATE dept SET deptdoc = ? WHERE id = '001'", SQL_NTS);
SQLBindParameter (hStmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_XML, 0, 0,
                 xmlBuffer, 10240, &length);
SQLExecute (hStmt);

```

Related concepts:

- “Application programming language support for XML” on page 165
- “Creation of tables with XML columns” on page 31
- “Updates of XML columns” on page 34
- “XML data encoding” on page 217
- “XML data type” on page 17
- “XML parsing” on page 37
- “XML data handling in CLI applications - Overview” on page 167

Related reference:

- “SQLBindParameter function (CLI) - Bind a parameter marker to a buffer or LOB locator” in *Call Level Interface Guide and Reference, Volume 2*
- “SQLExecute function (CLI) - Execute a statement” in *Call Level Interface Guide and Reference, Volume 2*
- “SQLPrepare function (CLI) - Prepare a statement” in *Call Level Interface Guide and Reference, Volume 2*
- “XMLPARSE scalar function” in *SQL Reference, Volume 1*
- “Data conversions supported in CLI” in *Call Level Interface Guide and Reference, Volume 1*

XML data retrieval in CLI applications

When you select data from XML columns in a table, the output data is in the serialized string 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 as `SQL_C_BINARY`, `SQL_C_CHAR`, `SQL_C_DBCHAR` or `SQL_C_WCHAR`. When retrieving a result set from an XML column, it is recommended that you bind your application variable to the `SQL_C_BINARY` type. Binding to character types can result in possible data loss resulting from code page conversion. Data loss can occur when characters in the source code page cannot be represented in the target code page. Binding your variable to the `SQL_C_BINARY` type avoids these issues.

XML data is returned to the application as internally encoded data. DB2 CLI determines the encoding of the data as follows:

- If the C type is `SQL_C_BINARY`, DB2 CLI returns the data in the UTF-8 encoding scheme.
- If the C type is `SQL_C_CHAR` or `SQL_C_DBCHAR`, DB2 CLI returns the data in the application code page encoding scheme.
- If the C type is `SQL_C_WCHAR`, DB2 CLI returns the data in the UCS-2 encoding scheme.

The database server performs an implicit serialization of the data before returning it to the application. You can explicitly serialize the XML data to a specific data type by calling the XMLSERIALIZE function. Implicit serialization is recommended, however, because explicitly serializing to character types with XMLSERIALIZE can introduce encoding issues.

The following example shows how to retrieve XML data from an XML column into a binary application variable.

```
char xmlBuffer[10240];
// xmlBuffer is used to hold the retrieved XML document
integer length;

// Assume a table named dept has been created with the following statement:
// CREATE TABLE dept (id CHAR(8), deptdoc XML)

length = sizeof (xmlBuffer);
SQLExecute (hStmt, "SELECT deptdoc FROM dept WHERE id='001'", SQL_NTS);
SQLBindCol (hStmt, 1, SQL_C_BINARY, xmlBuffer, &length, NULL);
SQLFetch (hStmt);
SQLCloseCursor (hStmt);
// xmlBuffer now contains a valid XML document encoded in UTF-8
```

Related concepts:

- “XML data handling in CLI applications - Overview” on page 167
- “Application programming language support for XML” on page 165
- “XML data encoding” on page 217
- “XML data querying” on page 45
- “XML data type” on page 17
- “XML serialization” on page 86

Related reference:

- “Data conversions supported in CLI” in *Call Level Interface Guide and Reference, Volume 1*
- “SQLBindCol function (CLI) - Bind a column to an application variable or LOB locator” in *Call Level Interface Guide and Reference, Volume 2*
- “SQLCloseCursor function (CLI) - Close cursor and discard pending results” in *Call Level Interface Guide and Reference, Volume 2*
- “SQLExecute function (CLI) - Execute a statement” in *Call Level Interface Guide and Reference, Volume 2*
- “SQLFetch function (CLI) - Fetch next row” in *Call Level Interface Guide and Reference, Volume 2*
- “XMLSERIALIZE scalar function” in *SQL Reference, Volume 1*

Changing of default XML type handling in CLI applications

DB2 CLI supports CLI/ODBC configuration keywords that provide compatibility for applications that do not expect the default types returned when describing or specifying SQL_C_DEFAULT for XML columns and parameter markers. Older CLI and ODBC applications might not recognize or expect the default SQL_XML type when describing XML columns or parameters. Some CLI or ODBC applications might also expect a default type other than SQL_C_BINARY for XML columns and parameter markers. To provide compatibility for these types of applications, DB2 CLI supports the MapXMLDescribe and MapXMLCDefault keywords.

MapXMLDescribe specifies which SQL data type is returned when XML columns or parameter markers are described.

MapXMLCDefault specifies the C type that is used when SQL_C_DEFAULT is specified for XML columns and parameter markers in DB2 CLI functions.

Related concepts:

- “Data types and data conversion in CLI applications” in *Call Level Interface Guide and Reference, Volume 1*
- “XML data handling in CLI applications - Overview” on page 167
- “XML data type” on page 17

Related reference:

- “CLI and ODBC function summary” in *Call Level Interface Guide and Reference, Volume 2*
- “MapXMLCDefault CLI/ODBC configuration keyword” in *Call Level Interface Guide and Reference, Volume 1*
- “MapXMLDescribe CLI/ODBC configuration keyword” in *Call Level Interface Guide and Reference, Volume 1*

Embedded SQL

Declaring XML host variables in embedded SQL applications

To exchange XML data between the database server and an embedded SQL application, you need to declare host variables in your application source code.

DB2 V9.1 introduces an XML data type that stores XML data in a structured set of nodes in a tree format. Columns with this XML data type are described as an SQL_TYP_XML column SQLTYPE, and applications can bind various language-specific data types for input to and output from these columns or parameters. XML columns can be accessed directly using SQL, the SQL/XML extensions, or XQuery. The XML data type applies to more than just columns. Functions can have XML value arguments and produce XML values as well. Similarly, stored procedures can take XML values as both input and output parameters. Finally, XQuery expressions produce XML values regardless of whether or not they access XML columns.

XML data is character in nature and has an encoding that specifies the character set used. The encoding of XML data can be determined externally, derived from the base application type containing the serialized string representation of the XML document. It can also be determined internally, which requires interpretation of the data. For Unicode encoded documents, a byte order mark (BOM), consisting of a Unicode character code at the beginning of a data stream is recommended. The BOM is used as a signature that defines the byte order and Unicode encoding form.

Existing character and binary types, which include CHAR, VARCHAR, CLOB, and BLOB may be used in addition to XML host variables for fetching and inserting data. However, they will not be subject to implicit XML parsing, as XML host variables would. Instead, an explicit XMLPARSE function with default whitespace stripping is injected and applied.

Restrictions:

XML and XQuery restrictions on developing embedded SQL applications

Procedure:

To declare XML host variables in embedded SQL applications:

In the declaration section of the application, declare the XML host variables as LOB data types:

- SQL TYPE IS XML AS CLOB(n) <hostvar_name>

where <hostvar_name> is a CLOB host variable that contains XML data encoded in the mixed codepage of the application.

- SQL TYPE IS XML AS DBCLOB(n) <hostvar_name>

where <hostvar_name> is a DBCLOB host variable that contains XML data encoded in the application graphic codepage.

- SQL TYPE IS XML AS BLOB(n) <hostvar_name>

where <hostvar_name> is a BLOB host variable that contains XML data internally encoded¹.

- SQL TYPE IS XML AS CLOB_FILE <hostvar_name>

where <hostvar_name> is a CLOB file that contains XML data encoded in the application mixed codepage.

- SQL TYPE IS XML AS DBCLOB_FILE <hostvar_name>

where <hostvar_name> is a DBCLOB file that contains XML data encoded in the application graphic codepage.

- SQL TYPE IS XML AS BLOB_FILE <hostvar_name>

where <hostvar_name> is a BLOB file that contains XML data internally encoded¹.

Notes:

1. Refer to the algorithm for determining encoding with XML 1.0 specifications (<http://www.w3.org/TR/REC-xml/#sec-guessing-no-ext-info>).

Related concepts:

- “Data types that map to SQL data types in embedded SQL applications” in *Developing Embedded SQL Applications*
- “Example: Referencing XML host variables in embedded SQL applications” on page 173
- “Background information on XML internal encoding” on page 217
- “XML data encoding” on page 217

Related tasks:

- “Executing XQuery expressions in embedded SQL applications” on page 174

Related reference:

- “Recommendations for developing embedded SQL applications with XML and XQuery” on page 176

Example: Referencing XML host variables in embedded SQL applications

The following sample applications demonstrate how to reference XML host variables in C and COBOL.

Example: Embedded SQL C application::

The following code example has been formatted for clarity:

```
EXEC SQL BEGIN DECLARE;
    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;

// as XML AS CLOB
// The XML value written to xmlBuf will be prefixed by an XML declaration
// similar to: <?xml version = "1.0" encoding = "ISO-8859-1" ?>
// Note: The encoding name will depend upon the application codepage
EXEC SQL SELECT xmlCol INTO :xmlBuf
    FROM myTable
    WHERE id = '001';
EXEC SQL UPDATE myTable
    SET xmlCol = :xmlBuf
    WHERE id = '001';

// as XML AS BLOB
// The XML value written to xmlblob will be prefixed by an XML declaration
// similar to: <?xml version = "1.0" encoding = "UTF-8"?>
EXEC SQL SELECT xmlCol INTO :xmlblob
    FROM myTable
    WHERE id = '001';
EXEC SQL UPDATE myTable
    SET xmlCol = :xmlblob
    WHERE id = '001';

// as CLOB
// The output will be encoded in the application character codepage,
// but will not contain an XML declaration
EXEC SQL SELECT XMLSERIALIZE (xmlCol AS CLOB(10K)) INTO :clobBuf
    FROM myTable
    WHERE id = '001';
EXEC SQL UPDATE myTable
    SET xmlCol = XMLPARSE (:clobBuf PRESERVE WHITESPACE)
    WHERE id = '001';
```

Example: Embedded SQL COBOL application::

The following code example has been formatted for clarity:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 xmlBuf USAGE IS SQL TYPE IS XML as CLOB(5K).
    01 clobBuf USAGE IS SQL TYPE IS CLOB(5K).
    01 xmlblob USAGE IS SQL TYPE IS BLOB(5K).
EXEC SQL END DECLARE SECTION END-EXEC.

* as XML
EXEC SQL SELECT xmlCol INTO :xmlBuf
    FROM myTable
    WHERE id = '001'.
EXEC SQL UPDATE myTable
    SET xmlCol = :xmlBuf
    WHERE id = '001'.

* as BLOB
EXEC SQL SELECT xmlCol INTO :xmlblob
    FROM myTable
```

```

WHERE id = '001'.
EXEC SQL UPDATE myTable
SET xmlCol = :xmlblob
WHERE id = '001'.

* as CLOB
EXEC SQL SELECT XMLSERIALIZE(xmlCol AS CLOB(10K)) INTO :clobBuf
FROM myTable
WHERE id= '001'.
EXEC SQL UPDATE myTable
SET xmlCol = XMLPARSE(:clobBuf) PRESERVE WHITESPACE
WHERE id = '001'.

```

Related tasks:

- “Declaring XML host variables in embedded SQL applications” on page 171
- “Executing XQuery expressions in embedded SQL applications” on page 174

Related reference:

- “Recommendations for developing embedded SQL applications with XML and XQuery” on page 176

Executing XQuery expressions in embedded SQL applications

You can store XML data in your tables and use embedded SQL applications to access the XML columns using XQuery expressions. To access XML data, use XML host variables instead of casting the data to character or binary data types. If you do not make use of XML host variables, the best alternative for accessing XML data is with FOR BIT DATA or BLOB data types to avoid codepage conversion.

Prerequisites:

- Declare XML host variables within your embedded SQL applications.

Restrictions:

- An XML type must be used to retrieve XML values in a static SQL SELECT INTO statement.
- If a CHAR, VARCHAR, CLOB, or BLOB host variable is used for input where an XML value is expected, the value will be subject to an XMLPARSE function operation with default whitespace (STRIP) handling. Otherwise, an XML host variable is required.

Procedure:

To execute XQuery expressions in embedded SQL application directly, prepend the expression with the "XQUERY" keyword. For static SQL use the XMLQUERY function. When the XMLQUERY function is called, the XQuery expression is not prefixed by "XQUERY".

Example 1: Executing XQuery expressions directly in C and C++ dynamic SQL by prepending the "XQUERY" keyword:

In C and C++ applications, XQuery expressions can be executed in the following way:

```

EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char stmt[16384];
SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
EXEC SQL END DECLARE SECTION;

```

```

sprintf( stmt, "XQUERY (10, xs:integer(1) to xs:integer(4))" );

EXEC SQL PREPARE s1 FROM :stmt;
EXEC SQL DECLARE c1 CURSOR FOR s1;
EXEC SQL OPEN c1;

while( sqlca.sqlcode == SQL_RC_OK )
{
    EXEC SQL FETCH c1 INTO :xmlblob;
    /* Display results */
}

EXEC SQL CLOSE c1;
EXEC SQL COMMIT;

```

Example 2: Executing XQuery expressions in static SQL using the XMLQUERY function:

SQL statements containing the XMLQUERY function can be prepared statically, as follows:

```

EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE C1 CURSOR FOR SELECT XMLQUERY( '(10, xs:integer(1) to
xs:integer(4))' RETURNING SEQUENCE BY REF) from SYSIBM.SYSDUMMY1;

EXEC SQL OPEN c1;

while( sqlca.sqlcode == SQL_RC_OK )
{
    EXEC SQL FETCH c1 INTO :xmlblob;
    /* Display results */
}

EXEC SQL CLOSE c1;
EXEC SQL COMMIT;

```

Example 3: Executing XQuery expressions in COBOL embedded SQL applications:

In COBOL applications, XQuery expressions can be executed in the following way:

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 stmt pic x(80).
    01 xmlBuff USAGE IS SQL TYPE IS XML AS BLOB (10K).
EXEC SQL END DECLARE SECTION END-EXEC.

MOVE "XQUERY (10, xs:integer(1) to xs:integer(4))" TO stmt.
EXEC SQL PREPARE s1 FROM :stmt END-EXEC.
EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.
EXEC SQL OPEN c1 USING :host-var END-EXEC.

*Call the FETCH and UPDATE loop.
Perform Fetch-Loop through End-Fetch-Loop
    until SQLCODE does not equal 0.

EXEC SQL CLOSE c1 END-EXEC.
EXEC SQL COMMIT END-EXEC.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :xmlBuff END-EXEC.

```

```

        if SQLCODE not equal 0
            go to End-Fetch-Loop.
    * Display results
End-Fetch-Loop. exit.

```

Related concepts:

- “Binary storage of variable values using the FOR BIT DATA clause in C and C++ embedded SQL applications” in *Developing Embedded SQL Applications*
- “Binary storage of variable values using the FOR BIT DATA clause in COBOL embedded SQL applications” in *Developing Embedded SQL Applications*
- “Data types that map to SQL data types in embedded SQL applications” in *Developing Embedded SQL Applications*
- “Example: Referencing XML host variables in embedded SQL applications” on page 173
- “XML serialization” on page 86

Related tasks:

- “Declaring XML host variables in embedded SQL applications” on page 171

Related reference:

- “Recommendations for developing embedded SQL applications with XML and XQuery” on page 176

Recommendations for developing embedded SQL applications with XML and XQuery

The following recommendations and restrictions apply to using XML and XQuery within embedded SQL applications.

- Applications must access all XML data in the serialized string format.
 - You must represent all data, including numeric and date time data, in its serialized string format.
- Externalized XML data is limited to 2 GB.
- All cursors containing XML data are non-blocking (each fetch operation produces a database server request).
- Whenever character host variables contain serialized XML data, the application code page is assumed to be used as the encoding of the data and must match any internal encoding that exists in the data.
- You must specify a LOB data type as the base type for an XML host variable.
- The following apply to static SQL:
 - Character and binary host variables cannot be used to retrieve XML values from a SELECT INTO operation.
 - Where an XML data type is expected for input, the use of CHAR, VARCHAR, CLOB, and BLOB host variables will be subject to an XMLPARSE operation with default whitespace handling characteristics ('STRIP WHITESPACE'). Any other non-XML host variable type will be rejected.
 - There is no support for static XQuery expressions; attempts to precompile an XQuery expression will fail with an error. You can only execute XQuery expressions through the XMLQUERY function.
- An XQuery expression can be dynamically executed by pre-pending the expression with the string "XQUERY".

Related concepts:

- “Errors and warnings from precompilation of embedded SQL applications” in *Developing Embedded SQL Applications*
- “Example: Referencing XML host variables in embedded SQL applications” on page 173

Related tasks:

- “Declaring XML host variables in embedded SQL applications” on page 171
- “Executing XQuery expressions in embedded SQL applications” on page 174

Identifying XML values in an SQLDA

To indicate that a base type holds XML data, the `sqlname` field of the SQLVAR must be updated as follows:

- `sqlname.length` must be 8
- The first two bytes of `sqlname.data` must be `X'0000'`
- The third and fourth bytes of `sqlname.data` should be `X'0000'`
- The fifth byte of `sqlname.data` must be `X'01'` (referred to as the XML subtype indicator only when the first two conditions are met)
- The remaining bytes should be `X'000000'`

If the XML subtype indicator is set in an SQLVAR whose SQLTYPE is non-LOB, an SQL0804 error (rc=115) will be returned at runtime.

Note: `SQL_TYP_XML` can only be returned from the DESCRIBE statement. This type cannot be used for any other requests. The application must modify the SQLDA to contain a valid character or binary type, and set the `sqlname` field appropriately to indicate that the data is XML.

Related concepts:

- “Host Variables in embedded SQL applications” in *Developing Embedded SQL Applications*

Related tasks:

- “Referencing host variables in embedded SQL applications” in *Developing Embedded SQL Applications*
- “Declaring host variables in embedded SQL applications” in *Developing Embedded SQL Applications*
- “Declaring the SQLDA structure in a dynamically executed SQL program” in *Developing Embedded SQL Applications*

Java

JDBC

XML data in JDBC 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.

In applications, XML data is in the serialized string format.

In JDBC applications, you can:

- Store an entire XML document in an XML column using `setXXX` methods.
- Retrieve an entire XML document from an XML column using `getXXX` methods.
- Retrieve a sequence from a document in an XML column by using the SQL `XMLQUERY` function to retrieve the sequence into a serialized XML string in the database, and then using `getXXX` methods to retrieve the data into an application variable.
- Retrieve a sequence from a document in an XML column by using an `XQuery` expression, prepended with the string 'XQUERY', to retrieve the elements of the sequence into a result table in the database, in which each row of the result table represents an item in the sequence. Then use `getXXX` methods to retrieve the data into application variables.
- Retrieve a sequence from a document in an XML column as a user-defined table by using the SQL `XMLTABLE` function to define the result table and retrieve it. Then use `getXXX` methods to retrieve the data from the result table into application variables.

Java has no XML data type, and invocations of metadata methods, such as `ResultSetMetaData.getColumnTypeName` return a type of `java.sql.Types.OTHER` for an XML column type.

Related concepts:

- “XML column updates in JDBC applications” on page 178
- “XML data retrieval in JDBC applications” on page 180

Related reference:

- “Data types that map to SQL data types in JDBC applications” in *Developing Java Applications*

XML column updates in JDBC applications

When you update or insert data into XML columns of a DB2 table, the input data must be in the serialized string format. Table 31 lists the methods and corresponding input data types that you can use to put data in XML columns.

Table 31. Methods and data types for updating XML columns

Method	Input data type
<code>PreparedStatement.setAsciiStream</code>	<code>InputStream</code>
<code>PreparedStatement.setBinaryStream</code>	<code>InputStream</code>
<code>PreparedStatement.setBlob</code>	<code>Blob</code>
<code>PreparedStatement.setBytes</code>	<code>byte[]</code>
<code>PreparedStatement.setCharacterStream</code>	<code>Reader</code>
<code>PreparedStatement.setClob</code>	<code>Clob</code>
<code>PreparedStatement.setObject</code>	<code>byte[]</code> , <code>Blob</code> , <code>Clob</code> , <code>DB2Xml</code> , <code>InputStream</code> , <code>Reader</code> , <code>String</code>
<code>PreparedStatement.setString</code>	<code>String</code>

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.

External encoding for Java applications is always Unicode encoding.

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. The database server handles incompatibilities between internal and external encoding as follows:

- If the database server is DB2 Database for Linux, UNIX, and Windows, the database server generates an error if the external and internal encoding are incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the database server ignores the internal encoding.
- If the database server is DB2 for z/OS, the database server ignores the internal encoding.

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.

Example: The following example demonstrates inserting data from a file into an XML column. The data is inserted as binary data, so the database server honors the internal encoding.

```
public void insertBinStream()
{
    PreparedStatement insertStmt = null;
    String sqls = null;
    int cid = 0;
    ResultSet rs=null;
    Statement stmt=null;
    try {
        sqls = "INSERT INTO CUSTOMER (CID, INFO) VALUES (?, ?)";
        insertStmt = conn.prepareStatement(sqls);
        insertStmt.setInt(1, cid);
        File file = new File(fn);
        insertStmt.setBinaryStream(2,
            new FileInputStream(file), (int)file.length());
        if (insertStmt.executeUpdate() != 1) {
            System.out.println("insertBinStream: No record inserted.");
        }
    }
    catch (IOException ioe) {
        ioe.printStackTrace();
    }
    catch (SQLException sqle) {
        System.out.println("insertBinStream: SQL Exception: " +
            sqle.getMessage());
        System.out.println("insertBinStream: SQL State: " +
            sqle.getSQLState());
        System.out.println("insertBinStream: SQL Error Code: " +
            sqle.getErrorCode());
    }
}
```

Related concepts:

- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221

Related reference:

- “Data types that map to SQL data types in JDBC applications” in *Developing Java Applications*

XML data retrieval in JDBC applications

When you retrieve data from XML columns of a DB2 table, the output data is in the serialized string format. This is true whether you retrieve the entire contents of an XML column or a sequence from the column.

You can use one of the following techniques to retrieve XML data:

- Use a `ResultSet.getXXX` method other than `ResultSet.getObject` to retrieve the data into a compatible data type.
- Use the `ResultSet.getObject` method to retrieve the data, and then cast it to the `DB2Xml` type and assign it to a `DB2Xml` object. Then use a `DB2Xml.getDB2XXX` or `DB2Xml.getDB2XmlXXX` method to retrieve the data into a compatible output data type.

`DB2Xml.getDB2XmlXXX` methods add XML declarations with encoding specifications to the output data. `DB2Xml.getDB2XXX` methods do not add XML declarations with encoding specifications to the output data.

Table 32 lists the `ResultSet` methods and corresponding output data types for retrieving XML data.

Table 32. *ResultSet methods and data types for retrieving XML data*

Method	Output data type
<code>ResultSet.getAsciiStream</code>	<code>InputStream</code>
<code>ResultSet.getBinaryStream</code>	<code>InputStream</code>
<code>ResultSet.getBytes</code>	<code>byte[]</code>
<code>ResultSet.getCharacterStream</code>	<code>Reader</code>
<code>ResultSet.getObject</code>	<code>DB2Xml</code>
<code>ResultSet.getString</code>	<code>String</code>

Table 33 lists the methods and corresponding output data types for retrieving data from a `DB2Xml` object, as well as the type of encoding in the XML declaration that the driver adds to the output data.

Table 33. *DB2Xml methods, data types, and added encoding specifications*

Method	Output data type	Type of XML internal encoding declaration added
<code>DB2Xml.getDB2AsciiStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2BinaryStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2Bytes</code>	<code>byte[]</code>	None
<code>DB2Xml.getDB2CharacterStream</code>	<code>Reader</code>	None
<code>DB2Xml.getDB2String</code>	<code>String</code>	None
<code>DB2Xml.getDB2XmlAsciiStream</code>	<code>InputStream</code>	US-ASCII
<code>DB2Xml.getDB2XmlBinaryStream</code>	<code>InputStream</code>	Specified by <code>getDB2XmlBinaryStream targetEncoding</code> parameter
<code>DB2Xml.getDB2XmlBytes</code>	<code>byte[]</code>	Specified by <code>DB2Xml.getDB2XmlBytes targetEncoding</code> parameter
<code>DB2Xml.getDB2XmlCharacterStream</code>	<code>Reader</code>	ISO-10646-UCS-2
<code>DB2Xml.getDB2XmlString</code>	<code>String</code>	ISO-10646-UCS-2

If the application executes the XMLSERIALIZE function on the data that is to be returned, after execution of the function, the data has the data type that is specified in the XMLSERIALIZE function, not the XML data type. Therefore, the driver handles the data as the specified type and ignores any internal encoding declarations.

Example: The following example demonstrates retrieving data from an XML column into a String variable.

```
public void fetchToString()
{
    System.out.println(">> fetchToString: Get XML data " +
        "using getString");
    PreparedStatement selectStmt = null;
    String sqls = null, stringDoc = null;
    ResultSet rs = null;

    try{
        sqls = "SELECT info FROM customer WHERE cid = " + cid;
        selectStmt = conn.prepareStatement(sqls);
        rs = selectStmt.executeQuery();

        // get metadata. Column type for XML column is java.sql.Types.OTHER
        ResultSetMetaData meta = rs.getMetaData();
        String colType = meta.getColumnTypeName(1);
        System.out.println("fetchToString: Column type = " + colType);

        while (rs.next()) {
            stringDoc = rs.getString(1);
            System.out.println("Document contents:");
            System.out.println(stringDoc);
        }
    } catch (SQLException sqle) {
        System.out.println("fetchToString: SQL Exception: " +
            sqle.getMessage());
        System.out.println("fetchToString: SQL State: " +
            sqle.getSQLState());
        System.out.println("fetchToString: SQL Error Code: " +
            sqle.getErrorCode());
    }
}
```

Example: The following example demonstrates retrieving data from an XML column into a DB2Xml object, and then using the DB2Xml.getDB2XmlString method to retrieve the data into a string with an added XML declaration with an ISO-10646-UCS-2 encoding specification.

```
public void fetchToDB2Xml()
{
    System.out.println(">> fetchToDB2Xml: Get XML data as a DB2XML object " +
        "using getObject");
    PreparedStatement selectStmt = null;
    String sqls = null, stringDoc = null;
    ResultSet rs = null;

    try{
        sqls = "SELECT info FROM customer WHERE cid = " + cid;
        selectStmt = conn.prepareStatement(sqls);
        rs = selectStmt.executeQuery();

        // get metadata. Column type for XML column is java.sql.Types.OTHER
        ResultSetMetaData meta = rs.getMetaData();
        String colType = meta.getColumnTypeName(1);
        System.out.println("fetchToObject: Column type = " + colType);
        if (rs.next() == false) {
            System.out.println("fetchToObject: "
```

```

        "Cannot read document with cid " + cid);
    }
    else {
        // Retrieve the XML data with getObject, and cast the object
        // as a DB2Xml object. Then write it to a string with
        // explicit internal ISO-10646-UCS-2 encoding.
        com.ibm.db2.jcc.DB2Xml xml =
            (com.ibm.db2.jcc.DB2Xml) rs.getObject(1);
        System.out.println (xml.getDB2XmlString());
    }
    rs.close();
}
catch (SQLException sqle) {
    System.out.println("fetchToString: SQL Exception: " +
        sqle.getMessage());
    System.out.println("fetchToString: SQL State: " +
        sqle.getSQLState());
    System.out.println("fetchToString: SQL Error Code: " +
        sqle.getErrorCode());
}
}
}

```

Related concepts:

- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221

Related reference:

- “Data types that map to SQL data types in JDBC applications” in *Developing Java Applications*

Invocation of routines with XML parameters in Java applications

SQL or external stored procedures and external user-defined functions can include XML parameters. For SQL procedures, those parameters in the stored procedure definition have the XML type. For external stored procedures and user-defined functions, XML parameters in the routine definition have the XML AS CLOB type. When you call a stored procedure or user-defined function that has XML parameters, you need to use a compatible data type in the invoking statement.

To call a routine with XML input parameters from a JDBC program, use parameters of the `com.ibm.db2.jcc.DB2Xml` type. To register XML output parameters, register the parameters as the `com.ibm.db2.jcc.DB2Types.XML` type.

Example: JDBC program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```

com.ibm.db2.jcc.DB2Xml in_xml = xmlvar;
com.ibm.db2.jcc.DB2Xml out_xml = null;
com.ibm.db2.jcc.DB2Xml inout_xml = xmlvar;
                                // Declare an input, output, and
                                // input/output XML parameter

Connection con;
CallableStatement cstmt;
ResultSet rs;
...
stmt = con.prepareCall("CALL SP_xml(?,?,?)");
                                // Create a CallableStatement object
cstmt.setObject (1, in_xml);      // Set input parameter
cstmt.registerOutParameter (2, com.ibm.db2.jcc.DB2Types.XML);
                                // Register out and input parameters
cstmt.registerOutParameter (3, com.ibm.db2.jcc.DB2Types.XML);
cstmt.executeUpdate();           // Call the stored procedure
System.out.println("Parameter values from SP_xml call: ");

```

```

System.out.println("Output parameter value ");
printBytes(out_xml.getDB2String());
                // Use the DB2-only method getBytes to
                // convert the value to bytes for printing
System.out.println("Input/output parameter value ");
printBytes(inout_xml.getDB2String());

```

To call a routine with XML parameters from an SQLJ program, use parameters of the `com.ibm.db2.jcc.DB2Xml` type.

Example: SQLJ program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```

com.ibm.db2.jcc.DB2Xml in_xml = xmlvar;
com.ibm.db2.jcc.DB2Xml out_xml = null;
com.ibm.db2.jcc.DB2Xml inout_xml = xmlvar;
                // Declare an input, output, and
                // input/output XML parameter
...
#sql [myConnCtx] {CALL SP_xml(:IN in_xml,
                            :OUT out_xml,
                            :INOUT inout_xml)};
                // Call the stored procedure
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value ");
printBytes(out_xml.getDB2String());
                // Use the DB2-only method getBytes to
                // convert the value to bytes for printing
System.out.println("Input/output parameter value ");
printBytes(inout_xml.getDB2String());

```

Related tasks:

- “Calling stored procedures in an SQLJ application” in *Developing Java Applications*
- “Calling stored procedures using CallableStatement methods” in *Developing Java Applications*

Related reference:

- “Data types that map to SQL data types in JDBC applications” in *Developing Java Applications*
- “DB2Xml interface” in *Developing Java Applications*

SQLJ

XML data in SQLJ 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.

In applications, XML data is in the serialized string format.

In SQLJ 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 single-row SELECT statements or iterators.
- Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function to retrieve the sequence into a serialized XML string in the

database, and then using using single-row SELECT statements or iterators to retrieve the data into an application variable.

- Retrieve a sequence from a document in an XML column by using an XQuery expression, prepended with the string 'XQUERY', to retrieve the elements of the sequence into a result table in the database, in which each row of the result table represents an item in the sequence. Then use using single-row SELECT statements or iterators to retrieve the data into application variables.
- Retrieve a sequence from a document in an XML column as a user-defined table by using the SQL XMLTABLE function to define the result table and retrieve it. Then use using single-row SELECT statements or iterators to retrieve the data from the result table into application variables.

Java has no XML data type, and invocations of metadata methods, such as `ResultSetMetaData.getColumnTypeName` return a type of `java.sql.Types.OTHER` for an XML column type.

Related concepts:

- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “XML column updates in SQLJ applications” on page 184
- “XML data retrieval in SQLJ applications” on page 186

XML column updates in SQLJ applications

When you update or insert data into XML columns of a DB2 table, the input data must be in the serialized string format. The host expression data types that you can use to update XML columns are:

- `com.ibm.db2.jcc.DB2Xml`
- `String`
- `byte`
- `Blob`
- `Clob`
- `sqlj.runtime.AsciiStream`
- `sqlj.runtime.BinaryStream`
- `sqlj.runtime.CharacterStream`

For stream types, you need to use an `sqlj.runtime.typeStream` host expression, rather than a `java.io.typeInputStream` host expression so that you can pass the length of the stream to the JDBC driver.

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. The external encoding is the default encoding for the JVM.

External encoding for Java applications is always Unicode encoding.

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. The database server handles incompatibilities between internal and external encoding as follows:

- If the database server is DB2 Database for Linux, UNIX, and Windows, the database server generates an error if the external and internal encoding are

incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the database server ignores the internal encoding.

- If the database server is DB2 for z/OS, the database server ignores internal encoding.

Data in XML columns is stored in UTF-8 encoding.

Example: Suppose that you use the following statement to insert data from String host expression `xmlString` into an XML column in a table on a DB2 database server. `xmlString` is a character type, so its external encoding is used, whether or not it has an internal encoding specification.

```
#sql [ctx] {INSERT INTO CUSTACC VALUES (1, :xmlString)};
```

Example: Suppose that you copy the data from `xmlString` into a byte array with CP500 encoding. The data contains an XML declaration with an encoding declaration for Cp500. Then you insert the data from the `byte[]` host expression into an XML column in a table on a DB2 database server.

```
byte[] xmlBytes = xmlString.getBytes("CP500");  
#sql[ctx] {INSERT INTO CUSTACC VALUES (4, :xmlBytes)};
```

A byte string is considered to be internally encoded data. The data is converted from its internal encoding scheme to UTF-8, if necessary, and stored in its hierarchical format on the DB2 database server.

Example: Suppose that you copy the data from `xmlString` into a byte array with US-ASCII encoding. Then you construct an `sqlj.runtime.AsciiStream` host expression, and insert data from the `sqlj.runtime.AsciiStream` host expression into an XML column in a table on a DB2 database server.

```
byte[] b = xmlString.getBytes("US-ASCII");  
java.io.ByteArrayInputStream xmlAsciiInputStream =  
    new java.io.ByteArrayInputStream(b);  
sqlj.runtime.AsciiStream sqljXmlAsciiStream =  
    new sqlj.runtime.AsciiStream(xmlAsciiInputStream, b.length);  
#sql[ctx] {INSERT INTO CUSTACC VALUES (4, :sqljXmlAsciiStream)};
```

`sqljXmlAsciiStream` is a stream type, so its internal encoding is used. The data is converted from its internal encoding to UTF-8 encoding and stored in its hierarchical form on the database server.

Example: `sqlj.runtime.CharacterStream` host expression: Suppose that you construct an `sqlj.runtime.CharacterStream` host expression, and insert data from the `sqlj.runtime.CharacterStream` host expression into an XML column in a table on a DB2 database server.

```
java.io.StringReader xmlReader =  
    new java.io.StringReader(xmlString);  
sqlj.runtime.CharacterStream sqljXmlCharacterStream =  
    new sqlj.runtime.CharacterStream(xmlReader, xmlString.length());  
#sql [ctx] {INSERT INTO CUSTACC VALUES (4, :sqljXmlCharacterStream)};
```

`sqljXmlCharacterStream` is a character type, so its external encoding is used, whether or not it has an internal encoding specification.

Example: Suppose that you retrieve a document from an XML column into a `com.ibm.db2.jcc.DB2Xml` host expression, and insert the data into an XML column in a table on a DB2 database server.

```

java.sql.ResultSet rs = s.executeQuery ("SELECT * FROM CUSTACC");
rs.next();
com.ibm.db2.jcc.DB2Xml xmlObject = (com.ibm.db2.jcc.DB2Xml)rs.getObject(2);
#sql [ctx] {INSERT INTO CUSTACC VALUES (6, :xmlObject)};

```

After you retrieve the data it is still in UTF-8 encoding, so when you insert the data into another XML column, no conversion occurs.

XML data retrieval in SQLJ applications

When you retrieve data from XML columns of a DB2 table, the output data is in the serialized string format.

The host expression or iterator data types that you can use to retrieve data from XML columns are:

- com.ibm.db2.jcc.DB2Xml
- String
- byte[]
- sqlj.runtime.AsciiStream
- sqlj.runtime.BinaryStream
- sqlj.runtime.CharacterStream

If the application does not call the XMLSERIALIZE function before data retrieval, the data is converted from UTF-8 to the external application encoding for the character data types, or the internal encoding for the binary data types. No XML declaration is added. If the host expression is an object of the com.ibm.db2.jcc.DB2Xml type, you need to call an additional method to retrieve the data from this object. The method that you call determines the encoding of the output data and whether an XML declaration with an encoding specification is added. Table 34 lists the methods that you can call to retrieve data from an com.ibm.db2.jcc.DB2Xml object, and the corresponding output data types and type of encoding in the XML declarations.

Table 34. DB2Xml methods, data types, and added encoding specifications

Method	Output data type	Type of XML internal encoding declaration added
DB2Xml.getDB2AsciiStream	InputStream	None
DB2Xml.getDB2BinaryStream	InputStream	None
DB2Xml.getDB2Bytes	byte[]	None
DB2Xml.getDB2CharacterStream	Reader	None
DB2Xml.getDB2String	String	None
DB2Xml.getDB2XmlAsciiStream	InputStream	US-ASCII
DB2Xml.getDB2XmlBinaryStream	InputStream	Specified by getDB2XmlBinaryStream <i>targetEncoding</i> parameter
DB2Xml.getDB2XmlBytes	byte[]	Specified by DB2Xml.getDB2XmlBytes <i>targetEncoding</i> parameter
DB2Xml.getDB2XmlCharacterStream	Reader	ISO-10646-UCS-2
DB2Xml.getDB2XmlString	String	ISO-10646-UCS-2

If the application executes the XMLSERIALIZE function on the data that is to be returned, after execution of the function, the data has the data type that is specified in the XMLSERIALIZE function, not the XML data type. Therefore, the driver handles the data as the specified type and ignores any internal encoding declarations.

Example: Suppose that you retrieve data from an XML column into a String host expression.

```
#sql iterator XmlStringIter (int, String);
#sql [ctx] siter = {SELECT C1, CADOC from CUSTACC};
#sql {FETCH :siter INTO :row, :outString};
```

The String type is a character type, so the data is converted from UTF-8 to the external encoding, which is the default JVM encoding, and returned without any XML declaration.

Example: Suppose that you retrieve data from an XML column into a byte[] host expression.

```
#sql iterator XmlByteArrayIter (int, byte[]);
XmlByteArrayIter biter = null;
#sql [ctx] biter = {SELECT c1, CADOC from CUSTACC};
#sql {FETCH :biter INTO :row, :outBytes};
```

The byte[] type is a binary type, so the data is converted from UTF-8 to the internal encoding, and returned without any XML declaration.

Example: Suppose that you retrieve a document from an XML column into a com.ibm.db2.jcc.DB2Xml host expression, but you need the data in a byte string with an XML declaration that includes an internal encoding specification for UTF-8.

```
#sql iterator DB2XmlIter (int, com.ibm.db2.jcc.DB2Xml);
DB2XmlIter db2xmliter = null;
com.ibm.db2.jcc.DB2Xml outDB2Xml = null;
#sql [ctx] db2xmliter = {SELECT c1, CADOC from CUSTACC};
#sql {FETCH :db2xmliter INTO :row, :outDB2Xml};
byte[] byteArray = outDB2XML.getDB2XmlBytes("UTF-8");
```

The FETCH statement retrieves the data into the DB2Xml object in UTF-8 encoding. The getDB2XmlBytes method with the UTF-8 argument adds an XML declaration with a UTF-8 encoding specification and stores the data in a byte array.

PHP

Introduction to PHP application development for DB2

PHP: Hypertext Preprocessor (PHP) is an interpreted programming language primarily intended for the development of Web applications. The first version of PHP was created by Rasmus Lerdorf and contributed under an open source license in 1995. PHP was initially a very simple HTML templating engine, but over time the developers of PHP added database access functionality, rewrote the interpreter, introduced object-oriented support, and improved performance. Today, PHP has become a popular language for Web application development because of its focus on practical solutions and support for the most commonly required functionality in Web applications.

For the easiest install and configuration experience on Linux[®], UNIX[®], or Windows operating systems, you can download and install Zend Core for IBM for use in production systems. Paid support for Zend Core for IBM is available from Zend. On Windows, precompiled binary versions of PHP are available for download from <http://php.net/>. Most Linux distributions include a precompiled version of PHP. On UNIX operating systems that do not include a precompiled version of PHP, you can compile your own version of PHP.

PHP is a modular language that enables you to customize the available functionality through the use of extensions. These extensions can simplify tasks such as reading, writing, and manipulating XML, creating SOAP clients and servers, and encrypting communications between server and browser. The most popular extensions for PHP, however, provide read and write access to databases so that you can easily create a dynamic database-driven Web site. IBM supports access to DB2 databases from PHP applications through two extensions offering distinct sets of features:

- `ibm_db2` is an extension written, maintained, and supported by IBM for access to DB2 databases. The `ibm_db2` extension offers a procedural application programming interface (API) that, in addition to the normal create, read, update, and write database operations, also offers extensive access to the database metadata. You can compile the `ibm_db2` extension with either PHP 4 or PHP 5.
- `PDO_ODBC` is a driver for the PHP Data Objects (PDO) extension that offers access to DB2 databases through the standard object-oriented database interface introduced in PHP 5.1. Despite its name, you can compile the `PDO_ODBC` extension directly against the DB2 libraries to avoid the communications overhead and potential interference of an ODBC driver manager.

A third extension, Unified ODBC, has historically offered access to DB2 database systems. It is not recommended that you write new applications with this extension because `ibm_db2` and `PDO_ODBC` both offer significant performance and stability benefits over Unified ODBC. The `ibm_db2` extension API makes porting an application that was previously written for Unified ODBC almost as easy as globally changing the `odbc_` function name to `db2_` throughout the source code of your application.

Related tasks:

- “Setting up the PHP environment on Linux or UNIX” in *Developing Perl and PHP Applications*
- “Setting up the PHP environment on Windows” in *Developing Perl and PHP Applications*

Executing XQuery expressions in PHP (`ibm_db2`)

After connecting to a DB2 database, your PHP script is ready to issue XQuery expressions. The `db2_exec()` and `db2_execute()` functions execute SQL statements, through which you can pass your XQuery expressions. A typical use of `db2_exec()` is to set the default schema for your application in a common include file or base class.

Prerequisites:

You must set up the PHP environment on your system and enable the `ibm_db2` extension.

Restrictions:

To avoid the security threat of injection attacks, `db2_exec()` should only be used to execute SQL statements composed of static strings. Interpolation of PHP variables representing user input into the XQuery expression can expose your application to injection attacks.

Procedure:

1. Call `db2_exec()` with the following arguments:

- a. The connection resource;
- b. A string containing the SQL statement, including the XQuery expression. The XQuery expression needs to be wrapped in a XMLQUERY clause in the SQL statement.
- c. (Optional): an array containing statement options

DB2_ATTR_CASE

For compatibility with database systems that do not follow the SQL standard, this option sets the case in which column names will be returned to the application. By default, the case is set to *DB2_CASE_NATURAL*, which returns column names as they are returned by DB2. You can set this parameter to *DB2_CASE_LOWER* to force column names to lower case, or to *DB2_CASE_UPPER* to force column names to upper case.

DB2_ATTR_CURSOR

This option sets the type of cursor that `ibm_db2` returns for result sets. By default, `ibm_db2` returns a forward-only cursor (*DB2_FORWARD_ONLY*) which returns the next row in a result set for every call to `db2_fetch_array()`, `db2_fetch_assoc()`, `db2_fetch_both()`, `db2_fetch_object()`, or `db2_fetch_row()`. You can set this parameter to *DB2_SCROLLABLE* to request a scrollable cursor so that the `ibm_db2` fetch functions accept a second argument specifying the absolute position of the row that you want to access within the result set .

2. Check the value returned by `db2_exec()`:
 - If the value is `FALSE`, the SQL statement failed. You can retrieve diagnostic information through the `db2_stmt_error()` and `db2_stmt_errormsg()` functions.
 - If the value is not `FALSE`, the SQL statement succeeded and returned a statement resource that can be used in subsequent function calls related to this query.

Example:

```
<?php
$xmlquery = '$doc/customerinfo/phone';
$stmt = db2_exec($conn, "select xmlquery('$xmlquery'
PASSING INFO AS \"doc\") from customer");?>
```

Routines

SQL procedures

XML and XQuery support in SQL procedures

SQL procedures support parameters and variables of data type XML. They can be used in SQL statements in the same way as variables of any other data type. In addition, variables of data type XML can be passed as parameters to XQuery expressions in `XMLEXISTS`, `XMLQUERY` and `XMLTABLE` expressions.

SQL procedures make it easy to add procedural logic around statements that produce or use XML values and store those values temporarily in variables.

The following example shows the declaration, use, and assignment of XML parameters and variables in an SQL procedure:

```

CREATE TABLE T1(C1 XML) %

CREATE PROCEDURE proc1(IN parm1 XML, IN parm2 VARCHAR(32000))
LANGUAGE SQL
BEGIN
    DECLARE var1 XML;

    /* check if the value of XML parameter parm1
       contains an item with a value less than 200 */
    IF(XMLEXISTS('$x/ITEM[value < 200]' passing by ref parm1 as "x"))THEN

        /* if it does, insert the value of parm1 into table T1 */
        INSERT INTO T1 VALUES(parm1);

    END IF;

    /* parse parameter parm2's value and assign it to a variable */
    SET var1 = XMLPARSE(document parm2 preserve whitespace);

    /* insert variable var1 into table T1
       INSERT INTO T1 VALUES(var1);

END %

```

In the example above there is a table T1 with an XML column. The SQL procedure accepts two parameters of data type XML named parm1 and parm2. Within the SQL procedure an XML variable is declared named var1.

The logic of the SQL procedure checks if the value of XML parameter parm1 contains an item with a value less than 200. If it does, the XML value is directly inserted into column C1 in table T1.

Then the value of parameter parm2 is parsed using the XMLPARSE function and assigned to XML variable var1. The XML variable value is then also inserted into column C1 in table T1.

The ability to implement control flow logic around XQuery operations makes it easy to develop complex algorithms that query and access XML data stored in a database.

Related concepts:

- “XML serialization” on page 86
- “XQuery” in *IBM DB2 XQuery Reference*
- “Atomic blocks and scope of variables in SQL procedures (BEGIN, END)” in *Developing SQL and External Routines*
- “Compound statements and scope of variables in SQL procedures” in *Developing SQL and External Routines*
- “Cursors in SQL procedures” in *Developing SQL and External Routines*
- “Effect of commits and rollbacks on XML parameter and variable values in SQL procedures” on page 193
- “Parameters in SQL procedures” in *Developing SQL and External Routines*
- “Parts of SQL procedures” in *Developing SQL and External Routines*
- “SQL PL logic elements in the SQL-procedure body” in *Developing SQL and External Routines*
- “SQLCODE and SQLSTATE variables in SQL procedures” in *Developing SQL and External Routines*

- “Structure of SQL procedures” in *Developing SQL and External Routines*
- “Variables in SQL procedures (DECLARE, SET statements)” in *Developing SQL and External Routines*
- “Cursors for XQuery expressions in SQL procedures” on page 191

Related tasks:

- “Designing SQL procedures” in *Developing SQL and External Routines*

Related reference:

- “CREATE PROCEDURE (SQL) statement” in *SQL Reference, Volume 2*

Cursors for XQuery expressions in SQL procedures

SQL Procedures support the definition of cursors on XQuery expressions. A cursor on an XQuery expression allows you to iterate over the elements of the XQuery sequence returned by the expression.

Unlike cursors defined on SQL statements, which can be defined either statically or dynamically, cursors on XQuery expressions can only be defined dynamically. To declare a cursor dynamically, it is necessary to declare a variable of type CHAR or VARCHAR to contain the XQuery expression that will define the cursor result set. The XQuery expression must be prepared before the cursor can be opened and the result set resolved.

An example of an SQL procedure that dynamically declares a cursor for an XQuery expression, opens the cursor, and fetches XML data is shown here:

```
CREATE PROCEDURE xmlProc(IN inCust XML, OUT resXML XML)
SPECIFIC xmlProc
LANGUAGE SQL
BEGIN
    DECLARE SQLSTATE CHAR(5);
    DECLARE stmt_text VARCHAR (1024);
    DECLARE customer XML;
    DECLARE cityXml XML;
    DECLARE city VARCHAR (100);
    DECLARE stmt STATEMENT;
    DECLARE cur1 CURSOR FOR stmt;

    -- Get the city of the input customer
    SET cityXml = XMLQUERY('$cust/customerinfo//city' passing inCust as "cust");
    SET city = XMLCAST(cityXml as VARCHAR(100));

    -- Iterate over all the customers from the city using an XQUERY cursor
    -- and collect the customer name values into the output XML value

    SET stmt_text = 'XQUERY for $cust
                    in db2-fn:xmlcolumn("CUSTOMER.INFO")
                    /*:customerinfo/*:addr[*:city= "' || city || '" ]
                    return <Customer>{$cust/../@Cid}{$cust/../*:name}</Customer>';

    -- Use the name of the city for the input customer data as a prefix
    SET resXML = cityXml;

    PREPARE stmt FROM stmt_text;
    OPEN cur1;

    FETCH cur1 INTO customer;
    WHILE (SQLSTATE = '00000') DO
        SET resXML = XMLCONCAT(resXML, customer);
        FETCH cur1 INTO customer;
    END WHILE;
```

```

set resXML = XMLQUERY('<result> {$res} </result>'
                      passing resXML as "res");

```

END

This SQL procedure collects the IDs and names of customers defined in a table name CUSTOMER that are located in the same city as the customer for which XML data is provided as an input parameter.

The SQL procedure above can be called by executing the CALL statement as follows:

```

CALL xmlProc(xmlparse(document '<customerinfo Cid="5002">
                             <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">905-566-7258</phone>
                             </customerinfo>' PRESERVE WHITESPACE),?,?)

```

If this SQL procedure is created and run against the SAMPLE database, it returns XML data for two customers.

SQL procedures do not support cursors for static XQuery expressions. However, there is a simple workaround for this limitation. Although parameter markers are not supported in dynamic XQuery expressions, a cursor definition that might seem easy to define with a static XQuery expression, can generally be easily rewritten as a cursor with dynamic XQuery expression.

For example, consider the following unsupported cursor definition:

```

DECLARE cur1 CURSOR FOR
XQUERY for $fname in db2-fn:sqlquery
("SELECT person
 FROM team
 WHERE docID < 10")//fullname where $fname/first = "Joe";

```

This cursor definition can be reformulated and declared in SQL procedures as follows:

```

DECLARE stmt_text VARCHAR (1024);
DECLARE stmt STATEMENT;
DECLARE cur1 CURSOR FOR stmt;

SET stmt_text = ' for $fname in db2-fn:sqlquery
("SELECT doc
 FROM T1
 WHERE DOCID < 10")//fullname where $fname/first = "Joe";

PREPARE stmt FROM stmt_text;

```

The cursor declared above defines a result set for an XQuery that itself contains an SQL query. The defined result set will contain the full names of the persons with the first name Joe by looking through the XML documents that define a person within a table containing the members of a team. Functionally, the SQL portion selects the XML documents in column person from a table team that have an ID greater than 10. The XQuery portion then selects the fullname values in the XML documents where the value first is Joe.

Related concepts:

- “Atomic blocks and scope of variables in SQL procedures (BEGIN, END)” in *Developing SQL and External Routines*
- “Compound statements and scope of variables in SQL procedures” in *Developing SQL and External Routines*
- “Cursors in SQL procedures” in *Developing SQL and External Routines*
- “Variables in SQL procedures (DECLARE, SET statements)” in *Developing SQL and External Routines*
- “Effect of commits and rollbacks on XML parameter and variable values in SQL procedures” on page 193
- “XML and XQuery support in SQL procedures” on page 189
- “Parameters in SQL procedures” in *Developing SQL and External Routines*
- “Parts of SQL procedures” in *Developing SQL and External Routines*
- “SQL PL logic elements in the SQL-procedure body” in *Developing SQL and External Routines*
- “SQLCODE and SQLSTATE variables in SQL procedures” in *Developing SQL and External Routines*
- “Structure of SQL procedures” in *Developing SQL and External Routines*

Related tasks:

- “Designing SQL procedures” in *Developing SQL and External Routines*

Related reference:

- “CREATE PROCEDURE (SQL) statement” in *SQL Reference, Volume 2*

Effect of commits and rollbacks on XML parameter and variable values in SQL procedures

Commits and rollbacks within SQL procedures affect the values of parameters and variables of data type XML. During the execution of SQL procedures, upon a commit or rollback operation, the values assigned to XML parameters and XML variables will no longer be available.

Attempts to reference an SQL variable or SQL parameter of data type XML after a commit or rollback operation will cause an error (SQL1354N, 560CE) to be raised.

To successfully reference XML parameters and variables after a commit or rollback operation occurs, new values must first be assigned to them.

Consider the availability of XML parameter and variable values when adding ROLLBACK and COMMIT statements to SQL procedures.

Related concepts:

- “Atomic blocks and scope of variables in SQL procedures (BEGIN, END)” in *Developing SQL and External Routines*
- “Compound statements and scope of variables in SQL procedures” in *Developing SQL and External Routines*
- “Cursors in SQL procedures” in *Developing SQL and External Routines*
- “XML and XQuery support in SQL procedures” on page 189
- “Parameters in SQL procedures” in *Developing SQL and External Routines*
- “Parts of SQL procedures” in *Developing SQL and External Routines*

- “SQLCODE and SQLSTATE variables in SQL procedures” in *Developing SQL and External Routines*
- “Structure of SQL procedures” in *Developing SQL and External Routines*
- “Variables in SQL procedures (DECLARE, SET statements)” in *Developing SQL and External Routines*
- “Cursors for XQuery expressions in SQL procedures” on page 191

Related tasks:

- “Designing SQL procedures” in *Developing SQL and External Routines*

External routines

XML data type support in external routines

External procedures and functions written in the following programming languages support parameters and variables of data type XML:

- C
- C++
- COBOL
- Java
- .NET CLR languages

External OLE and OLEDB routines do not support parameters of data type XML.

XML data type values are represented in external routine code in the same way as CLOB data types.

When declaring external routine parameters of data type XML, the CREATE PROCEDURE and CREATE FUNCTION statements that will be used to create the routines in the database must specify that the XML data type is to be stored as a CLOB data type. The size of the CLOB value should be close to the size of the XML document represented by the XML parameter.

The CREATE PROCEDURE statement below shows a CREATE PROCEDURE statement for an external procedure implemented in the C programming language with an XML parameter named parm1:

```
CREATE PROCEDURE myproc(IN parm1 XML AS CLOB(2M), IN parm2 VARCHAR(32000))
LANGUAGE C
FENCED
PARAMETER STYLE SQL
EXTERNAL NAME 'mylib!myproc';
```

Similar considerations apply when creating external UDFs, as shown in the example below:

```
CREATE FUNCTION myfunc (IN parm1 XML AS CLOB(2M))
RETURNS SMALLINT
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NOT FENCED
NULL CALL
NO SQL
NO EXTERNAL ACTION
EXTERNAL NAME 'mylib!myfunc'
```

XML values are serialized before they are passed to external routines.

Within external routine code, XML parameter and variable values are accessed, set, and modified in the same way as in database applications.

Related concepts:

- “Overview of external routines” in *Developing SQL and External Routines*
- “Encoding considerations for passing XML data in routine parameters” on page 220
- “Parameters and variables of data type XML in SQL functions” in *Developing SQL and External Routines*
- “XML and XQuery support in SQL procedures” on page 189

Related tasks:

- “Specifying a column data type” in *Developing SQL and External Routines*

Related reference:

- “Supported SQL data types for the DB2 .NET Data Provider” in *Developing SQL and External Routines*
- “Supported SQL data types in DB2GENERAL routines” in *Developing SQL and External Routines*
- “Supported SQL data types in Java routines” in *Developing SQL and External Routines*

Specification of a driver for Java routines

Java routine development and invocation requires that a JDBC or SQLJ driver be specified. Java routines can use one of the following two drivers:

- IBM DB2 Driver for JDBC and SQLJ
- DB2 Type 2 Driver

The specification of a driver for Java routines is done by setting the `DB2_USE_DB2JCCT2_JROUTINE` environment variable.

By default the `DB2_USE_DB2JCCT2_JROUTINE` environment variable is not set and the IBM DB2 Type 2 Driver is used. This driver is the default so as to ensure the continued functioning of Java routines created in a DB2 UDB Version 8 instance upon an upgrade to DB2 Version 9, however it is strongly recommended that the IBM DB2 Driver for JDBC and SQLJ be used.

The IBM DB2 Driver is preferred because it is a more robust driver, the DB2 Type 2 Driver is deprecated, and because it is a prerequisite if Java routines contain:

- Parameters of data type XML
- Variables of data type XML
- References to XML data
- References to XML functions
- Any other native-XML feature

To upgrade to using the DB2 Driver for JDBC and SQLJ, set the `DB2_USE_DB2JCCT2_JROUTINE` DB2 environment variable to the value, YES. This can be done by issuing the following command from a DB2 Command Window:

```
db2set DB2_USE_DB2JCCT2_JROUTINE=YES
```

After issuing this command, you must stop and restart the DB2 instance in order for the change to take effect.

Note: Upgrading to the DB2 Driver for JDBC and SQLJ from the DB2 Type 2 Driver might introduce changed behaviors in existing Java routines. Refer to the documentation on the differences in behaviors between these two drivers to identify potential changes and to determine how to manage or disable them.

Related concepts:

- “Java routines” in *Developing SQL and External Routines*

Related tasks:

- “Migrating Java applications that use DB2 JDBC Type 2 or 3 driver” in *Migration Guide*
- “Migrating Java applications that use IBM DB2 Driver for JDBC and SQLJ” in *Migration Guide*
- “Migrating Java routines” in *Migration Guide*

Related reference:

- “JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers” in *Developing Java Applications*

Example: XML and XQuery support in Java (JDBC) procedure

Once the basics of Java procedures, programming in Java using the JDBC application programming interface (API), and XQuery are understood, you can start creating and using Java procedures that query XML data.

This example of a Java procedure illustrates:

- the CREATE PROCEDURE statement for a parameter stle JAVA procedure
- the source code for a parameter style JAVA procedure
- input and output parameters of data type XML
- use of an XML input parameter in a query
- assignment of the result of an XQuery, an XML value, to an output parameter
- assignment of the result of an SQL statement, an XML value, to an output parameter

Prerequisites:

Before working with this Java procedure example you might want to read the following topics:

- Java routines
- Routines
- Building Java routine code

The examples below makes use of a table named `xmltable` defined and containing data as follows:

```
CREATE TABLE xmltable
(
  num INTEGER,
  xdata XML
)@
```

```

INSERT INTO xmltable VALUES
(1, XMLPARSE(DOCUMENT '<doc>
    <type>car</type>
    <make>Pontiac</make>
    <model>Sunfire</model>
  </doc>' PRESERVE WHITESPACE)),
(2, XMLPARSE(DOCUMENT '<doc>
    <type>car</type>
    <make>Mazda</make>
    <model>Miata</model>
  </doc>' PRESERVE WHITESPACE)),
(3, XMLPARSE(DOCUMENT '<doc>
    <type>person</type>
    <name>Mary</name>
    <town>Vancouver</town>
    <street>Waterside</street>
  </doc>' PRESERVE WHITESPACE)),
(4, XMLPARSE(DOCUMENT '<doc>
    <type>person</type>
    <name>Mark</name>
    <town>Edmonton</town>
    <street>Oak</street>
  </doc>' PRESERVE WHITESPACE)),
(5, XMLPARSE(DOCUMENT '<doc>
    <type>animal</type>
    <name>dog</name>
  </doc>' PRESERVE WHITESPACE)),
(6, NULL),
(7, XMLPARSE(DOCUMENT '<doc>
    <type>car</type>
    <make>Ford</make>
    <model>Taurus</model>
  </doc>' PRESERVE WHITESPACE)),
(8, XMLPARSE(DOCUMENT '<doc>
    <type>person</type>
    <name>Kim</name>
    <town>Toronto</town>
    <street>Elm</street>
  </doc>' PRESERVE WHITESPACE)),
(9, XMLPARSE(DOCUMENT '<doc>
    <type>person</type>
    <name>Bob</name>
    <town>Toronto</town>
    <street>Oak</street>
  </doc>' PRESERVE WHITESPACE)),
(10, XMLPARSE(DOCUMENT '<doc>
    <type>animal</type>
    <name>bird</name>
  </doc>' PRESERVE WHITESPACE)))@

```

Procedure:

Use the following example as references when making your own Java procedures:

- Table 35 on page 198
- Table 36 on page 198

The Java external code file:

The example shows a Java procedure implementation. The example consists of two parts: the CREATE PROCEDURE statement and the external Java code implementation of the procedure from which the associated Java class can be built.

The Java source file that contains the procedure implementations of the following examples is named `stpclass.java` included in a JAR file named `myJAR`. The file has

the following format:

Table 35. Java external code file format

```
using System;
import java.lang.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import com.ibm.db2.jcc.DB2Xml;

public class stpclass
{
    ...
    // Java procedure implementations
    ...
}
```

The Java class file imports are indicated at the top of the file. The `com.ibm.db2.jcc.DB2Xml` import is required if any of the procedures in the file contain parameters or variables of type XML will be used.

It is important to note the name of the class file and JAR name that contains a given procedure implementation. These names are important, because the EXTERNAL clause of the CREATE PROCEDURE statement for each procedure must specify this information so that DB2 can locate the class at run time.

Example 1: Parameter style JAVA procedure with XML parameters:

This example shows the following:

- CREATE PROCEDURE statement for a parameter style JAVA procedure
- Java code for a parameter style JAVA procedure with XML parameters

This procedure takes an input parameter, `inXML`, inserts a row including that value into a table, queries XML data using both an SQL statement and an XQuery expression, and sets two output parameters, `outXML1`, and `outXML2`.

Table 36. Code to create a parameter style JAVA procedure with XML parameters

```
CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER,
                           IN inXML XML as CLOB (1K),
                           OUT out1XML XML as CLOB (1K),
                           OUT out2XML XML as CLOB (1K)
                           )

DYNAMIC RESULT SETS 0
DETERMINISTIC
LANGUAGE JAVA
PARAMETER STYLE JAVA
MODIFIES SQL DATA
FENCED
THREADSAFE
DYNAMIC RESULT SETS 0
PROGRAM TYPE SUB
NO DBINFO
EXTERNAL NAME 'myJar:stpclass.xmlProc1'
```

Table 36. Code to create a parameter style JAVA procedure with XML parameters (continued)

```

//*****
// Stored Procedure: XMLPROC1
//
// Purpose:  Inserts XML data into XML column; queries and returns XML data
//
// Parameters:
//
// IN:      inNum -- the sequence of XML data to be insert in xmldata table
//          inXML -- XML data to be inserted
// OUT:    out1XML -- XML data to be returned
//          out2XML -- XML data to be returned
//
//*****
public void xmlProc1(int inNum,
                    DB2Xml inXML ,
                    DB2Xml[] out1XML,
                    DB2Xml[] out2XML
                    )
throws Exception
{
    Connection con = DriverManager.getConnection("jdbc:default:connection");

    // Insert data including the XML parameter value into a table
    String query = "INSERT INTO xmltable (num, inXML ) VALUES ( ?, ? )" ;
    String xmlString = inXML.getDB2String() ;

    stmt = con.prepareStatement(query);
    stmt.setInt(1, inNum);
    stmt.setString (2, xmlString );
    stmt.executeUpdate();
    stmt.close();

    // Query and retrieve a single XML value from a table using SQL
    query = "SELECT xdata from xmltable WHERE num = ? " ;

    stmt = con.prepareStatement(query);
    stmt.setInt(1, inNum);
    ResultSet rs = stmt.executeQuery();

    if ( rs.next() )
    { out1Xml[0] = (DB2Xml) rs.getObject(1); }

    rs.close() ;
    stmt.close();

    // Query and retrieve a single XML value from a table using XQuery
    query = "XQUERY for $x in db2-fn:xmlcolumn(\"XMLTABLE.XDATA\")/doc
            where $x/make = 'Mazda'
            return <carInfo>{$x/make}{$x/model}</carInfo>";

    stmt = con.createStatement();

    rs = stmt.executeQuery( query );

    if ( rs.next() )
    { out2Xml[0] = (DB2Xml) rs.getObject(1) ; }

    rs.close();
    stmt.close();
    con.close();

    return ;
}

```

Related concepts:

- “Java routines” in *Developing SQL and External Routines*
- “Routines” in *Developing SQL and External Routines*

Related tasks:

- “Building Java routine code” in *Developing SQL and External Routines*

Example: XML and XQuery support in C# .NET CLR procedure

Once the basics of procedures, the essentials of .NET common language runtime routines, XQuery and XML are understood, you can start creating and using CLR procedures with XML features.

The example below demonstrates a C# .NET CLR procedure with parameters of type XML as well as how to update and query XML data.

Prerequisites:

Before working with the CLR procedure example you might want to read the following concept topics:

- Common language runtime (CLR) routines
- Creating CLR routines
- Benefits of using routines
- Building common language runtime (CLR) .NET routines

The examples below makes use of a table named `xmltable` that is defined as follows:

```
CREATE TABLE xmltable
(
  num INTEGER,
  xdata XML
)

INSERT INTO xmltable VALUES
(1, XMLPARSE(DOCUMENT '<doc>
                <type>car</type>
                <make>Pontiac</make>
                <model>Sunfire</model>
                </doc>' PRESERVE WHITESPACE)),
(2, XMLPARSE(DOCUMENT '<doc>
                <type>car</type>
                <make>Mazda</make>
                <model>Miata</model>
                </doc>' PRESERVE WHITESPACE)),
(3, XMLPARSE(DOCUMENT '<doc>
                <type>person</type>
                <name>Mary</name>
                <town>Vancouver</town>
                <street>Waterside</street>
                </doc>' PRESERVE WHITESPACE)),
(4, XMLPARSE(DOCUMENT '<doc>
                <type>person</type>
                <name>Mark</name>
                <town>Edmonton</town>
                <street>Oak</street>
                </doc>' PRESERVE WHITESPACE)),
(5, XMLPARSE(DOCUMENT '<doc>
                <type>animal</type>
                <name>dog</name>
                </doc>' PRESERVE WHITESPACE)),
(6, NULL),
```

```

(7, XMLPARSE(DOCUMENT '<doc>
    <type>car</type>
    <make>Ford</make>
    <model>Taurus</model>
  </doc>' PRESERVE WHITESPACE)),
(8, XMLPARSE(DOCUMENT '<doc>
    <type>person</type>
    <name>Kim</name>
    <town>Toronto</town>
    <street>Elm</street>
  </doc>' PRESERVE WHITESPACE)),
(9, XMLPARSE(DOCUMENT '<doc>
    <type>person</type>
    <name>Bob</name>
    <town>Toronto</town>
    <street>Oak</street>
  </doc>' PRESERVE WHITESPACE)),
(10, XMLPARSE(DOCUMENT '<doc>
    <type>animal</type>
    <name>bird</name>
  </doc>' PRESERVE WHITESPACE))@

```

Procedure:

Use the following examples as references when making your own C# CLR procedures:

- “The C# external code file”
- “Example 1: C# parameter style GENERAL procedure with XML features” on page 202

The C# external code file:

The example consists of two parts: the CREATE PROCEDURE statement and the external C# code implementation of the procedure from which the associated assembly can be built.

The C# source file that contains the procedure implementations of the following examples is named `gwenProc.cs` and has the following format:

Table 37. C# external code file format

```

using System;
using System.IO;
using System.Data;
using IBM.Data.DB2;
using IBM.Data.DB2Types;

namespace bizLogic
{
    class empOps
    {
        ...
        // C# procedures
        ...
    }
}

```

The file inclusions are indicated at the top of the file. The `IBM.Data.DB2` inclusion is required if any of the procedures in the file contain SQL. The `IBM.Data.DB2Types` inclusion is required if any of the procedures in the file contains parameters or variables of type XML. There is a namespace declaration in this file and a class `empOps` that contains the procedures. The use of namespaces is optional. If a

namespace is used, the namespace must appear in the assembly path name provided in the EXTERNAL clause of the CREATE PROCEDURE statement.

It is important to note the name of the file, the namespace, and the name of the class, that contains a given procedure implementation. These names are important, because the EXTERNAL clause of the CREATE PROCEDURE statement for each procedure must specify this information so that DB2 can locate the assembly and class of the CLR procedure.

Example 1: C# parameter style GENERAL procedure with XML features:

This example shows the following:

- CREATE PROCEDURE statement for a parameter style GENERAL procedure
- C# code for a parameter style GENERAL procedure with XML parameters

This procedure takes two parameters, an integer inNum and inXML. These values are inserted into the table xml table. Then an XML value is retrieved using XQuery. Another XML value is retrieved using SQL. The retrieved XML values are assigned to two output parameters, outXML1 and outXML2. No result sets are returned.

Table 38. Code to create a C# parameter style GENERAL procedure

```
CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER,
                           IN inXML XML as CLOB (1K),
                           OUT inXML XML as CLOB (1K),
                           OUT inXML XML as CLOB (1K)
                           )
LANGUAGE CLR
PARAMETER STYLE GENERAL
DYNAMIC RESULT SETS 0
FENCED
THREADSAFE
DETERMINISTIC
NO DBINFO
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!xmlProc1' ;

//*****
// Stored Procedure: xmlProc1
//
// Purpose:  insert XML data into XML column
//
// Parameters:
//
// IN:      inNum -- the sequence of XML data to be insert in xmldata table
//          inXML -- XML data to be inserted
// OUT:     outXML1 -- XML data returned - value retrieved using XQuery
//          outXML2 -- XML data returned - value retrieved using SQL
//*****
```

Table 38. Code to create a C# parameter style GENERAL procedure (continued)

```

public static void xmlProc1 (      int      inNum, DB2Xml  inXML,
                                out DB2Xml  outXML1, out DB2Xml  outXML2 )
{
    // Create new command object from connection context
    DB2Parameter parm;
    DB2Command cmd;
    DB2DataReader reader = null;
    outXML1 = DB2Xml.Null;
    outXML2 = DB2Xml.Null;

    // Insert input XML parameter value into a table
    cmd = DB2Context.GetCommand();
    cmd.CommandText = "INSERT INTO "
        + "xmltable( num , xdata ) "
        + "VALUES( ?, ? )";

    parm = cmd.Parameters.Add("@num", DB2Type.Integer );
    parm.Direction = ParameterDirection.Input;
    cmd.Parameters["@num"].Value = inNum;
    parm = cmd.Parameters.Add("@data", DB2Type.Xml);
    parm.Direction = ParameterDirection.Input;
    cmd.Parameters["@data"].Value = inXML ;
    cmd.ExecuteNonQuery();
    cmd.Close();

    // Retrieve XML value using XQuery
    // and assign value to an XML output parameter
    cmd = DB2Context.GetCommand();
    cmd.CommandText = "XQUERY for $x " +
        "in db2-fn:xmlcolumn(\"XMLTABLE.XDATA\")/doc "+
        "where $x/make = 'Mazda' " +
        "return <carInfo>{$x/make}{$x/model}</carInfo>";
    reader = cmd.ExecuteReader();
    reader.CacheData= true;

    if (reader.Read())
    { outXML1 = reader.GetDB2Xml(0); }
    else
    { outXML1 = DB2Xml.Null; }

    reader.Close();
    cmd.Close();

    // Retrieve XML value using SQL
    // and assign value to an XML output parameter value
    cmd = DB2Context.GetCommand();
    cmd.CommandText = "SELECT xdata "
        + "FROM xmltable "
        + "WHERE num = ?";

    parm = cmd.Parameters.Add("@num", DB2Type.Integer );
    parm.Direction = ParameterDirection.Input;
    cmd.Parameters["@num"].Value = inNum;
    reader = cmd.ExecuteReader();
    reader.CacheData= true;

    if (reader.Read())
    { outXML2 = reader.GetDB2Xml(0); }
    else
    { outXML = DB2Xml.Null; }

    reader.Close() ;
    cmd.Close();

    return;
}

```

Related concepts:

- “.NET common language runtime (CLR) routines” in *Developing SQL and External Routines*
- “Benefits of using routines” in *Developing SQL and External Routines*

Related tasks:

- “Building Common Language Runtime (CLR) .NET routines” in *Developing ADO.NET and OLE DB Applications*
- “Creating .NET CLR routines from DB2 Command Windows” in *Developing SQL and External Routines*

Example: XML and XQuery support in C procedure

Once the basics of procedures, the essentials of C routines, XQuery and XML are understood, you can start creating and using C procedures with XML features.

The example below demonstrates a C procedure with parameters of type XML as well as how to update and query XML data.

Prerequisites:

Before working with the C procedure example you might want to read the following concept topics:

- C routines
- Creating C routines
- Benefits of using routines
- Building C routines

The examples below makes use of a table named `xmltable` that is defined as follows:

```
CREATE TABLE xmltable
(
  num INTEGER,
  xdata XML
)

INSERT INTO xmltable VALUES
(1, XMLPARSE(DOCUMENT '<doc>
                <type>car</type>
                <make>Pontiac</make>
                <model>Sunfire</model>
                </doc>' PRESERVE WHITESPACE)),
(2, XMLPARSE(DOCUMENT '<doc>
                <type>car</type>
                <make>Mazda</make>
                <model>Miata</model>
                </doc>' PRESERVE WHITESPACE)),
(3, XMLPARSE(DOCUMENT '<doc>
                <type>person</type>
                <name>Mary</name>
                <town>Vancouver</town>
                <street>Waterside</street>
                </doc>' PRESERVE WHITESPACE)),
(4, XMLPARSE(DOCUMENT '<doc>
                <type>person</type>
                <name>Mark</name>
                <town>Edmonton</town>
                <street>Oak</street>
                </doc>' PRESERVE WHITESPACE)),
```

```

(5, XMLPARSE(DOCUMENT '<doc>
                    <type>animal</type>
                    <name>dog</name>
                    </doc>' PRESERVE WHITESPACE)),
(6, NULL),
(7, XMLPARSE(DOCUMENT '<doc>
                    <type>car</type>
                    <make>Ford</make>
                    <model>Taurus</model>
                    </doc>' PRESERVE WHITESPACE)),
(8, XMLPARSE(DOCUMENT '<doc>
                    <type>person</type>
                    <name>Kim</name>
                    <town>Toronto</town>
                    <street>Elm</street>
                    </doc>' PRESERVE WHITESPACE)),
(9, XMLPARSE(DOCUMENT '<doc>
                    <type>person</type>
                    <name>Bob</name>
                    <town>Toronto</town>
                    <street>Oak</street>
                    </doc>' PRESERVE WHITESPACE)),
(10, XMLPARSE(DOCUMENT '<doc>
                    <type>animal</type>
                    <name>bird</name>
                    </doc>' PRESERVE WHITESPACE))

```

Procedure:

Use the following examples as references when making your own C procedures:

- “The C external code file”
- “Example 1: C parameter style SQL procedure with XML features” on page 206

The C external code file:

The example consists of two parts: the CREATE PROCEDURE statement and the external C code implementation of the procedure from which the associated assembly can be built.

The C source file that contains the procedure implementations of the following examples is named gwenProc.SQC and has the following format:

Table 39. C external code file format

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include <sqlca.h>
#include <sqludf.h>
#include <sql.h>
#include <memory.h>

// C procedures
    ...

```

The file inclusions are indicated at the top of the file. There are no extra include files required for XML support in embedded SQL routines.

It is important to note the name of the file and the name of the function that corresponds to the procedure implementation. These names are important, because the EXTERNAL clause of the CREATE PROCEDURE statement for each procedure must specify this information so that the DB2 database manager can locate the library and entry point that corresponds to the C procedure.

Example 1: C parameter style SQL procedure with XML features:

This example shows the following:

- CREATE PROCEDURE statement for a parameter style SQL procedure
- C code for a parameter style SQL procedure with XML parameters

This procedure receives two input parameters. The first input parameter is named inNum and is of type INTEGER. The second input parameters is named inXML and is of type XML. The values of the input parameters are used to insert a row into the table xml table. Then an XML value is retrieved using an SQL statement. Another XML value is retrieved using an XQuery expression. The retrieved XML values are respectively assigned to two output parameters, out1XML and out2XML. No result sets are returned.

Table 40. Code to create a C parameter style SQL procedure

```
CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER,
                           IN inXML XML as CLOB (1K),
                           OUT inXML XML as CLOB (1K),
                           OUT inXML XML as CLOB (1K)
                           )
LANGUAGE C
PARAMETER STYLE SQL
DYNAMIC RESULT SETS 0
FENCED
THREADSAFE
DETERMINISTIC
NO DBINFO
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc!xmlProc1' ;

//*****
// Stored Procedure: xmlProc1
//
// Purpose:  insert XML data into XML column
//
// Parameters:
//
// IN:      inNum -- the sequence of XML data to be insert in xmldata table
//          inXML -- XML data to be inserted
// OUT:     out1XML -- XML data returned - value retrieved using XQuery
//          out2XML -- XML data returned - value retrieved using SQL
//*****
```

Table 40. Code to create a C parameter style SQL procedure (continued)

```

#ifdef __cplusplus
extern "C"
#endif
SQL_API_RC SQL_API_FN testSecA1(sqlint32* inNum,
                                SQLUDF_CLOB* inXML,
                                SQLUDF_CLOB* out1XML,
                                SQLUDF_CLOB* out2XML,
                                SQLUDF_NULLIND *inNum_ind,
                                SQLUDF_NULLIND *inXML_ind,
                                SQLUDF_NULLIND *out1XML_ind,
                                SQLUDF_NULLIND *out2XML_ind,
                                SQLUDF_TRAIL_ARGS)
{
    char *str;
    FILE *file;

    EXEC SQL INCLUDE SQLCA;

    EXEC SQL BEGIN DECLARE SECTION;
        sqlint32 hvNum1;
        SQL TYPE IS XML AS CLOB(200) hvXML1;
        SQL TYPE IS XML AS CLOB(200) hvXML2;
        SQL TYPE IS XML AS CLOB(200) hvXML3;
    EXEC SQL END DECLARE SECTION;

    /* Check null indicators for input parameters */
    if ((*inNum_ind < 0) || (*inXML_ind < 0)) {
        strcpy(sqludf_sqlstate, "38100");
        strcpy(sqludf_msgtext, "Received null input");
        return 0;
    }

    /* Copy input parameters to host variables */
    hvNum1 = *inNum;
    hvXML1.length = inXML->length;
    strncpy(hvXML1.data, inXML->data, inXML->length);

    /* Execute SQL statement */
    EXEC SQL
        INSERT INTO xmltable (num, xdata) VALUES (:hvNum1, :hvXML1);

    /* Execute SQL statement */
    EXEC SQL
        SELECT xdata INTO :hvXML2
        FROM xmltable
        WHERE num = :hvNum1;

    sprintf(stmt5, "SELECT XMLQUERY('for $x in $xmldata/doc
                                return <carInfo>{$x/model}</carInfo>'
                                passing by ref xmltable.xdata
                                as \"xmldata\" returning sequence)
        FROM xmltable WHERE num = ?");

    EXEC SQL PREPARE selstmt5 FROM :stmt5 ;
    EXEC SQL DECLARE c5 CURSOR FOR selstmt5;
    EXEC SQL OPEN c5 using :hvNum1;
    EXEC SQL FETCH c5 INTO :hvXML3;

    exit:

    /* Set output return code */
    *outReturnCode = sqlca.sqlcode;
    *outReturnCode_ind = 0;

    return 0;
}

```

Performance of routines

The performance of routines is impacted by a variety of factors including the type and implementation of the routine, the number of SQL statements within the routine, the degree of complexity of the SQL in the routine, the number of parameters to the routine, the efficiency of the logic within the routine implementation, the error handling within the routines and more. Because users often choose to implement routines to improve the performance of applications, it is important to get the most out of routine performance.

The following table outlines some of the general factors that impact routine performance and gives recommendations on how to improve routine performance by altering each factor. For further details on performance factors that impact specific routine types, refer to the performance and tuning topics for the specific routine type.

Table 41. Performance considerations and routine performance recommendations

Performance consideration	Performance recommendation
Routine type: procedure, function, method	<ul style="list-style-type: none"> Procedures, functions, and methods serve different purposes and are referenced in different places. Their functional differences make it difficult to compare their performance directly. In general procedures can sometimes be rewritten as functions (particularly if they return a scalar value and only query data) and enjoy slight performance improvements, however these benefits are generally a result of simplifying the SQL required to implement the SQL logic. User-defined functions with complex initializations can make use of scratchpads to store any values required in the first invocation so that they can be used in subsequent invocations.
Routine implementation: system-defined or user-defined	<ul style="list-style-type: none"> For equivalent logic, built-in routines perform the best, followed by system-defined routines, because they enjoy a closer relationship with the database engine than do user-defined routines. User-defined routines can perform very well if they are well coded and follow best practices.

Table 41. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Routine implementation: SQL or external routine implementation	<ul style="list-style-type: none"> • In general, SQL routines perform comparably or better than external routines, because SQL routines always run in NOT FENCED mode, a mode which provides a performance benefit over most external routines. • SQL procedures generally perform better than logically equivalent external procedures. • For simple logic, SQL function performance will be comparable to that of an equivalent external function. • For complex logic, such as math algorithms and string manipulation functions that require little SQL, it is better to use an external routine in a low level programming language such as C since there is less dependence on SQL support. • See Comparison of routine implementations for a comparison of the features, including performance, of the supported external routine programming language options.
External routine implementation programming language	<ul style="list-style-type: none"> • See: Comparison of external routine APIs and programming languages for a comparison of the performance features that you should consider when selecting an external routine implementation. • Java (JDBC and SQLJ APIs) <ul style="list-style-type: none"> – Java routines with very large memory requirements are best created with the FENCED NOT THREADSAFE clause specified. Java routines with average memory requirements can be specified with the FENCED THREADSAFE clause. – For fenced threadsafe Java routine invocations, DB2 attempts to choose a threaded Java fenced mode process with a Java heap that is large enough to run the routine. Failure to isolate large heap consumers in their own process can result in out-of-Java-heap errors in multi-threaded Java db2fmp processes. FENCED THREADSAFE routines, in contrast, perform better because they can share a small number of JVMs. • C and C++ <ul style="list-style-type: none"> – In general C and C++ routines perform better than other external routine implementations and as well as SQL routines. – To perform their best C and C++ routines should be compiled in 32-bit format if they will be deployed to a 32-bit DB2 instance and in 64-bit format if they will be deployed to a 64-bit DB2 instance. • COBOL <ul style="list-style-type: none"> – In general COBOL performance is good, but COBOL is not a recommended routine implementation.

Table 41. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Number of SQL statements within the routine	<ul style="list-style-type: none"> • Routines should contain more than one SQL statement, otherwise the overhead of routine invocation is not performance cost effective. • Logic that must make several database queries, process intermediate results, and ultimately return a subset of the data that was worked with is the best logic for routine encapsulation. Complex data mining, and large updates requiring lookups of related data are examples of this type of logic. Heavy SQL processing is done on the database server and only the smaller data result set is passed back to the caller.
Complexity of SQL statements within the routine	<ul style="list-style-type: none"> • It makes good sense to include very complex queries within your routines so that you capitalize on the greater memory and performance capabilities of the database server. • Do not worry about the SQL statements being overly complex.
Static or dynamic SQL execution within routines	<ul style="list-style-type: none"> • In general static SQL performs better than dynamic SQL. In routines there are no additional differences when you use static or dynamic SQL.
Number of parameters to routines	<ul style="list-style-type: none"> • Minimizing the number of parameters to routines can improve routine performance as this minimizes the number of buffers to be passed between the routine and routine invoker.
Data types of routine parameters	<ul style="list-style-type: none"> • You can improve the performance of routines by using VARCHAR parameters instead of CHAR parameters in the routine definition. Using VARCHAR data types instead of CHAR data types prevents DB2 from padding parameters with spaces before passing the parameter and decreases the amount of time required to transmit the parameter across a network. For example, if your client application passes the string "A SHORT STRING" to a routine that expects a CHAR(200) parameter, DB2 has to pad the parameter with 186 spaces, null-terminate the string, then send the entire 200 character string and null-terminator across the network to the routine. In comparison, passing the same string, "A SHORT STRING", to a routine that expects a VARCHAR(200) parameter results in DB2 simply passing the 14 character string and a null terminator across the network.
Initialization of parameters to routines	<ul style="list-style-type: none"> • It is a good idea to always initialize input parameters to routines, particularly if the input routine parameter values are null. For null value routine parameters, a shorter or empty buffer can be passed to the routine instead of a full sized buffer, which can improve performance.

Table 41. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Number of local variables in routines	<ul style="list-style-type: none"> • Minimizing the number of local variables declared within a routine can improve performance by minimizing the number of SQL statements executed within the routine. • In general aim to use as few variables as possible. Re-use variables if this will not be semantically confusing.
Initialization of local variables in routines	<ul style="list-style-type: none"> • If possible, it is a good practice to initialize multiple local variables within a single SQL statement as this saves on the total SQL execution time for the routine.
Number of result sets returned by procedures	<ul style="list-style-type: none"> • If you can reduce the number of result sets returned by a routine you can improve routine performance.
Size of result sets returned by routines	<ul style="list-style-type: none"> • Make sure that for each result set returned by a routine, the query defining the result filters the columns returned and the number of rows returned as much as possible. Returning unnecessary columns or rows of data is not efficient and can result in sub-optimal routine performance.
Efficiency of logic within routines	<ul style="list-style-type: none"> • As with any application, the performance of a routine can be limited by a poorly implemented algorithm. Aim to be as efficient as possible when programming routines and apply generally recommend coding best practices as much as possible. • For SQL procedures, minimize the quantity of SQL PL statements if possible. These statements are intended to provide control-flow support around traditional SQL statements and are not intended to be used excessively within SQL procedures. • Analyze your SQL and wherever possible reduce your query to its simplest form. This can often be done by using CASE expressions instead of CAST statements or by collapsing multiple SQL statements into a single statement that uses a CASE expression as a switch.
Error handling within routines	<ul style="list-style-type: none"> • If you have extensive error handling in routines and anticipate many errors to be handled this can impact routine performance. Minimize error handling to improve routine performance. In SQL procedures, take care to ensure when using multiple error handlers that error handling cannot lead to cyclical firing of error handlers.

Table 41. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Run-time mode of routine (FENCED or NOT FENCED clause specification)	<p>NOT FENCED clause usage:</p> <ul style="list-style-type: none"> • In general, creating your routine with the NOT FENCED clause, which makes it runs in the same process as the DB2 database manager, is preferable over creating it with the FENCED clause, which makes it run in a special DB2 process outside of the engine's address space. • While you can expect improved routine performance when running routines as not fenced, user code in unfenced routines can accidentally or maliciously corrupt the database or damage the database control structures. You should only use the NOT FENCED clause when you need to maximize performance benefits, and if you deem the routine to be secure. (For information on assessing and mitigating the risks of registering C/C++ routines as NOT FENCED, refer to Security of routines. If the routine is not safe enough to run in the database manager's process, use the FENCED clause when creating the routine. To limit the creation and running of potentially unsafe code, DB2 requires that a user have a special privilege, CREATE_NOT_FENCED_ROUTINE in order to create NOT FENCED routines. • If an abnormal termination occurs while you are running a NOT FENCED routine, the database manager will attempt an appropriate recovery if the routine is registered as NO SQL. However, for routines not defined as NO SQL, the database manager will fail. • NOT FENCED routines must be precompiled with the WCHARTYPE NOCONVERT option if the routine uses GRAPHIC or DBCLOB data.

Table 41. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Run-time mode of routine (FENCED or NOT FENCED clause specification)	<p>FENCED THREADSAFE clause usage</p> <ul style="list-style-type: none"> • Routines created with the FENCED THREADSAFE clause run in the same process as other routines. More specifically, non-Java routines share one process, while Java(TM) routines share another process, separate from routines written in other languages. This separation protects Java routines from the potentially more error prone routines written in other languages. Also, the process for Java routines contains a JVM, which incurs a high memory cost and is not used by other routine types. Multiple invocations of FENCED THREADSAFE routines share resources, and therefore incur less system overhead than FENCED NOT THREADSAFE routines, which each run in their own dedicated process. • If you feel your routine is safe enough to run in the same process as other routines, use the THREADSAFE clause when registering it. As with NOT FENCED routines, information on assessing and mitigating the risks of registering C/C++ routines as FENCED THREADSAFE is in the topic, "Security considerations for routines". • If a FENCED THREADSAFE routine abnormally ends, only the thread running this routine is terminated. Other routines in the process continue running. However, the failure that caused this thread to abnormally end can adversely affect other routine threads in the process, causing them to trap, hang, or have damaged data. After one thread abends, the process is no longer used for new routine invocations. Once all the active users complete their jobs in this process, it is terminated. • When you register Java routines, they are deemed THREADSAFE unless you indicate otherwise. All other LANGUAGE types are NOT THREADSAFE by default. Routines using LANGUAGE OLE and OLE DB cannot be specified as THREADSAFE. • NOT FENCED routines must be THREADSAFE. It is not possible to register a routine as NOT FENCED NOT THREADSAFE (SQLCODE -104). • Users on UNIX(R) can see their Java and C THREADSAFE processes by looking for db2fmp (Java) or db2fmp (C).

Table 41. Performance considerations and routine performance recommendations (continued)

Performance consideration	Performance recommendation
Run-time mode of routine (FENCED or NOT FENCED clause specification)	<p>FENCED NOT THREADSAFE mode</p> <ul style="list-style-type: none"> FENCED NOT THREADSAFE routines each run in their own dedicated process. If you are running numerous routines, this can have a detrimental effect on database system performance. If the routine is not safe enough to run in the same process as other routines, use the NOT THREADSAFE clause when registering the routine. On UNIX, NOT THREADSAFE processes appear as db2fmp (pid) (where pid is the process id of the agent using the fenced mode process) or as db2fmp (idle) for a pooled NOT THREADSAFE db2fmp.
Level of SQL access in routine: NO SQL, CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA	<ul style="list-style-type: none"> Routines that are created with a lower level of SQL access clause will perform better than routines created with a higher level of SQL access clause. Therefore you should declare your routines with the most restrictive level of SQL access clause. For example, if your routine only reads SQL data, do not create it with the MODIFIES SQL DATA clause, but rather create it with the more restrictive READS SQL DATA clause.
Determinism of routine (DETERMINISTIC or NOT DETERMINISTIC clause specification)	<ul style="list-style-type: none"> Declaring a routine with the DETERMINISTIC or NOT DETERMINISTIC clause has no impact on routine performance.
Number and complexity of external actions made by routine (EXTERNAL ACTION clause specification)	<ul style="list-style-type: none"> Depending on the number of external actions and the complexity of external actions performed by an external routine, routine performance can be hindered. Factors that contribute to this are network traffic, access to files for writing or reading, the time required to execute the external action, and the risk associated with hangs in external action code or behaviors.
Routine invocation when input parameters are null (CALLED ON NULL INPUT clause specification)	<ul style="list-style-type: none"> If receiving null input parameter values results in no logic being executed and an immediate return by the routine, you can modify your routine so that it is not fully invoked when null input parameter values are detected. To create a routine that ends invocation early if routine input parameters are received, create the routine and specify the CALLED ON NULL INPUT clause.
Procedure parameters of type XML	<ul style="list-style-type: none"> The passing of parameters of data type XML is significantly less efficient in external procedures implemented in either the C or JAVA programming language than in SQL procedures. When passing one or more parameters of data type XML, consider using SQL procedures instead of external procedures.

Once routines are created and deployed, it might be harder to determine what environmental and routine specific factors are impacting routine performance, and hence it is important to design routines with performance in mind.

Related concepts:

- “Interoperability of routines” in *Developing SQL and External Routines*
- “Comparison of routine implementations” in *Developing SQL and External Routines*
- “Comparison of supported APIs and programming languages for external routine development” in *Developing SQL and External Routines*
- “Overview of routines” in *Developing SQL and External Routines*
- “Portability of routines” in *Developing SQL and External Routines*
- “Routines” in *Developing SQL and External Routines*
- “Security of routines” in *Developing SQL and External Routines*
- “Performance tuning guidelines” in *Performance Guide*

Related tasks:

- “Determining what SQL statements can be executed in routines” in *Developing SQL and External Routines*

Chapter 8. XML encoding

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.
- XML data that is in a binary application data type or binary data that is in a character data type 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 checks for internal encoding.

If the external encoding and the internal encoding are not Unicode encoding, the effective CCSID that is associated with the internal encoding must match the external encoding. Otherwise, an error occurs. If the external encoding and the internal encoding are Unicode encoding, and the encoding schemes do not match, the DB2 database server ignores the internal encoding.

Related concepts:

- “Encoding considerations for passing XML data in routine parameters” on page 220
- “Encoding considerations for retrieval of XML data from a database” on page 219
- “Native XML data store overview” on page 1
- “XML data type” on page 17
- “Application programming language support for XML” on page 165
- “Background information on XML internal encoding” on page 217
- “Encoding considerations for input of XML data to a database” on page 219

Related reference:

- “Native XML data store roadmap” in *DB2 Information Center*

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 processing instruction 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 42. 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
UTF-32 Big Endian	X'0000FEFF'	UTF-32
UTF-32 Little Endian	X'FFFE0000'	UTF-32

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 concepts:

- “Native XML data store overview” on page 1
- “XML data encoding” on page 217
- “XML data type” on page 17

Related reference:

- “Mappings of CCSIDs to encoding names for serialized XML output data” on page 344
- “Mappings of encoding names to effective CCSIDs for stored XML data” on page 333

Considerations

Encoding considerations for input of XML data to a database

When you store XML data in a DB2 table, observe the following rules:

- If the internal and external encoding are not Unicode encoding, for externally encoded XML data (data that is sent to the database server using character data types), any internally encoded declaration *must* match the external encoding. Otherwise, an error occurs, and the database manager rejects the document. If the external encoding and the internal encoding are Unicode encoding, and the encoding schemes do not match, the DB2 database server ignores the internal 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:

- “Updates of XML columns” on page 34
- “XML data encoding” on page 217
- “Background information on XML internal encoding” on page 217
- “Encoding considerations for passing XML data in routine parameters” on page 220
- “Encoding considerations for retrieval of XML data from a database” on page 219
- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “Encoding scenarios for input of externally encoded XML data to a database” on page 224
- “Encoding scenarios for input of internally encoded XML data to a database” on page 222
- “Insertion into XML columns” on page 32

Related reference:

- “Mappings of encoding names to effective CCSIDs for stored XML data” on page 333

Encoding considerations for retrieval of XML data from a database

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.

Data loss is less of a problem for Java and .NET applications than for other types of applications because Java and .NET 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:

- “Encoding considerations for input of XML data to a database” on page 219
- “Encoding considerations for passing XML data in routine parameters” on page 220
- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “Encoding scenarios for retrieval of XML data with explicit XMLSERIALIZE” on page 228
- “Encoding scenarios for retrieval of XML data with implicit serialization” on page 226
- “XML data encoding” on page 217
- “XML data querying” on page 45

Related reference:

- “Mappings of CCSIDs to encoding names for serialized XML output data” on page 344

Encoding considerations for passing XML data in routine parameters

In a DB2 database system, the following XML data types are available for parameters in a stored procedure or user-defined function definition:

XML For SQL procedures.

XML AS CLOB

For external SQL procedures and external user-defined functions.

Data in XML AS CLOB parameters is subject to character conversion, if the application encoding is not UTF-8. You should avoid the overhead of character conversion in an external user-defined function or stored procedure. Any application character or graphic data type can be used for the parameters in the calling application, but the source data should not contain an encoding declaration. 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.

Related concepts:

- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “Encoding considerations for input of XML data to a database” on page 219
- “Encoding considerations for retrieval of XML data from a database” on page 219
- “XML and XQuery support in SQL procedures” on page 189
- “XML data encoding” on page 217
- “XML data type support in external routines” on page 194

Encoding considerations for XML data in JDBC, SQLJ, and .NET applications

In general, there are fewer XML encoding considerations for Java applications than for DB2 CLI or embedded SQL applications. 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 because the application code page is always Unicode.

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 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 `getDB2Xmlxxx` methods, the JDBC driver adds an encoding declaration, as shown in the following table.

<code>getDB2Xmlxxx</code>	Encoding in declaration
<code>getDB2XmlString</code>	ISO-10646-UCS-2
<code>getDB2XmlBytes(String targetEncoding)</code>	Encoding specified by <i>targetEncoding</i>
<code>getDB2XmlAsciiStream</code>	US-ASCII
<code>getDB2XmlCharacterStream</code>	ISO-10646-UCS-2
<code>getDB2XmlBinaryStream(String targetEncoding)</code>	Encoding specified by <i>targetEncoding</i>

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 concepts:

- “XML data in JDBC applications” on page 177
- “XML data in SQLJ applications” on page 183
- “XML data handling in CLI applications - Overview” on page 167

- “Application programming language support for XML” on page 165
- “Background information on XML internal encoding” on page 217

Related tasks:

- “Declaring XML host variables in embedded SQL applications” on page 171

Related reference:

- “XMLSERIALIZE scalar function” in *SQL Reference, Volume 1*

Scenarios

Encoding scenarios for input of internally encoded XML data to a database

The following examples demonstrate how internal encoding affects 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 database.

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
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))
```

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))
```

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))
```

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))
```

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 concepts:

- “Background information on XML internal encoding” on page 217

- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “Encoding scenarios for input of externally encoded XML data to a database” on page 224
- “XML data encoding” on page 217

Related reference:

- “Mappings of encoding names to effective CCSIDs for stored XML data” on page 333

Encoding scenarios for input of externally encoded XML data to a database

The following examples demonstrate how external encoding affects data conversion and truncation during input of XML data to an XML column.

In general, when you use a character application data type, there is not a problem with code page conversion during input to a database.

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))
```

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))
```

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))
```

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)))
```

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 concepts:

- “XML data encoding” on page 217
- “Encoding considerations for input of XML data to a database” on page 219
- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “Encoding scenarios for input of internally encoded XML data to a database” on page 222

Encoding scenarios for retrieval of XML data with implicit serialization

The following examples demonstrate how the target encoding and application code page affect data conversion, truncation, and internal encoding during XML data retrieval with implicit serialization.

Only scenario 1 and scenario 2 apply to Java and .NET applications, because the application code page for Java applications is always Unicode. In general, code page conversion is not a problem for Java and .NET 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: None.

Internal encoding in the serialized data: For applications other than Java or .NET applications, the data is prefixed by the following XML declaration:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

For Java or .NET applications, no encoding declaration is added, unless you cast the data as the `com.ibm.db2.jcc.DB2Xml` type, and use a `getDB2Xmlxxx` method to retrieve the data. The declaration that is added depends on the `getDB2Xmlxxx` that you use.

Scenario 2:

Encoding source	Value
Target data encoding	UTF-16 Unicode
Target application data type	Graphic
Application code page	Any SBCS code page or CCSID 1208

Example output statements:

```
SELECT XMLCOL FROM T1
```

Character conversion: Data is converted from UTF-8 to UTF-16.

Data loss: None.

Truncation: Truncation can occur during conversion from UTF-8 to UTF-16, due to expansion.

Internal encoding in the serialized data: For applications other than Java or .NET 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 or .NET applications, no encoding declaration is added, unless you cast the data as the `com.ibm.db2.jcc.DB2Xml` type, and use a `getDB2Xmlxxx` method to retrieve the data. The declaration that is added depends on the `getDB2Xmlxxx` 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: None.

Internal encoding in the serialized 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: Truncation can occur during conversion from UTF-8 to CCSID 943 due to expansion.

Internal encoding in the serialized data: The data is prefixed by the following XML declaration:

```
<?xml version="1.0" encoding="Windows-31J" ?>
```

Related concepts:

- “Background information on XML internal encoding” on page 217
- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “Encoding scenarios for retrieval of XML data with explicit XMLSERIALIZE” on page 228
- “XML data encoding” on page 217
- “XML serialization” on page 86

Related reference:

- “Mappings of CCSIDs to encoding names for serialized XML output data” on page 344

Encoding scenarios for retrieval of XML data with explicit XMLSERIALIZE

The following examples demonstrate how the target encoding and application code page affect data conversion, truncation, and internal encoding during XML data retrieval with an explicit XMLSERIALIZE invocation.

Only scenario 1 and scenario 2 apply to Java and .NET 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: None.

Internal encoding in the serialized 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	Any SBCS code page or CCSID 1208

Example output statements:

```
SELECT XMLSERIALIZE(XMLCOL AS CLOB(1M) EXCLUDING XMLDECLARATION) FROM T1
```

Character conversion: Data is converted from UTF-8 to UTF-16.

Data loss: None.

Truncation: Truncation can occur during conversion from UTF-8 to UTF-16, due to expansion.

Internal encoding in the serialized 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: None.

Internal encoding in the serialized 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: Truncation can occur during conversion from UTF-8 to CCSID 943 due to expansion.

Internal encoding in the serialized 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.

Related concepts:

- “Character substitutions during code page conversions” in *Developing SQL and External Routines*
- “Background information on XML internal encoding” on page 217
- “Encoding considerations for XML data in JDBC, SQLJ, and .NET applications” on page 221
- “Encoding scenarios for retrieval of XML data with implicit serialization” on page 226
- “XML data encoding” on page 217

Related reference:

- “Mappings of CCSIDs to encoding names for serialized XML output data” on page 344
- “XMLSERIALIZE scalar function” in *SQL Reference, Volume 1*

Chapter 9. Restrictions on native XML data store

Native XML data store functionality is subject to some restrictions. An overview of the key restrictions is provided below. For further detail, refer to the documentation for a specific feature.

Restrictions on XML column definitions:

XML columns can only be defined in a table of a database defined with the UTF-8 code set. The XML document stored in an XML column must be well-formed. While there is no architectural limit on the size of an XML value stored in the database, serialized XML data that is exchanged with the database is limited to 2GB.

XML columns:

- cannot be included as columns of keys, including primary, foreign, and unique keys, dimension keys of multi-dimensional clustering (MDC) tables, sequence keys of range-clustered tables, distribution keys, and data partitioning keys.
- cannot be part of any index except an index over XML data
- cannot have a default value specified by the WITH DEFAULT clause; if the column is nullable, the default for the column is NULL
- cannot be used in a range-clustered table (RCT)
- cannot be used in a multi-dimensional clustering (MDC) table
- cannot be used in a table with a distribution key
- cannot be used in a table partitioned by range
- cannot be included in typed tables and typed views
- cannot be added to tables that have type-1 indexes defined on them (note that type-1 indexes are deprecated indexes; new indexes since DB2 UDB Version 8.1 are created as type-2 indexes)
- cannot be referenced in CHECK constraints (except for a VALIDATED predicate)
- cannot be referenced in generated columns
- cannot be referenced in the triggered-action of a CREATE TRIGGER statement
- cannot be specified in the select-list of scrollable cursors
- cause data blocking to be disabled when retrieving XML data

Additional restrictions exist for creating indexes on XML columns (also known as indexes over XML data). Please see the Related reference section below.

Restrictions on database partitions:

- The use of any features of the native XML data store will prevent future use of the database partitioning feature available with DB2 Enterprise Server Edition.
- An XML column or XML schema repository (XSR) object cannot be defined in a table of a database with more than one database partition defined.
- If a database is defined with a single database partition and includes XML columns or XSR objects, then a new database partition cannot be added.

Restrictions on utilities:

Loading data into tables containing XML columns using the load utility is not supported. Data movement of XML data should be performed using the import and export utilities.

Related concepts:

- “Check constraints” in *Developing SQL and External Routines*
- “Creation of tables with XML columns” on page 31
- “Database Partitioning Feature (DPF)” in *Administration Guide: Planning*
- “Distribution keys” in *Administration Guide: Planning*
- “Multidimensional clustering tables” in *Administration Guide: Planning*
- “Native XML data store overview” on page 1
- “Table partitioning” in *Administration Guide: Planning*
- “Typed tables” in *Developing SQL and External Routines*
- “Typed views” in *Developing SQL and External Routines*
- “XML data type” on page 17
- “Indexes” in *SQL Reference, Volume 1*
- “Restrictions on indexes over XML data” on page 126
- “Load overview” in *Data Movement Utilities Guide and Reference*
- “XML data movement overview” on page 155

Related reference:

- “Native XML data store roadmap” in *DB2 Information Center*
- “CREATE TRIGGER statement” in *SQL Reference, Volume 2*
- “VALIDATED predicate” in *SQL Reference, Volume 1*
- “Export/Import/Load utility unicode considerations” in *Data Movement Utilities Guide and Reference*

Chapter 10. Annotated XML schema decomposition

Annotated XML schema decomposition

Annotated XML schema decomposition

Annotated XML schema decomposition is a type of decomposition that operates based on annotations specified in an XML schema. Decomposition, sometimes referred to as "shredding", is the process of storing content from an XML document in columns of relational tables. After it is decomposed, the data then has the SQL type of the column it was inserted into.

An XML schema consists of one or more XML schema documents. In annotated XML schema decomposition, or schema-based decomposition, you control decomposition by annotating a document's XML schema with decomposition annotations. These annotations specify details such as the name of the target table and column the XML data is to be stored in, the default SQL schema for when a target table's SQL schema is not identified, as well as any transformation of the content before it is stored. Refer to the summary of decomposition annotations for further examples of what can be specified through these annotations.

The annotated schema documents must be stored in and registered with the XML schema repository (XSR). The schema must then be enabled for decomposition. If a schema document is not successfully registered, or if the XML schema is not enabled for decomposition with the XSR, then the decomposition process cannot infer the mapping and therefore cannot determine how elements and attributes should be decomposed; schema-based decomposition cannot occur in this case.

After the successful registration of the annotated schema, decomposition can be performed either by calling one of the decomposition stored procedures or by executing the DECOMPOSE XML DOCUMENT command.

Note that schema-based decomposition can be disabled or made inoperative. Refer to the topic on decomposition disablement for more information.

Related concepts:

- "Disablement of annotated XML schema decomposition" on page 239
- "XML decomposition annotations - Overview" on page 244
- "XML input and output overview" on page 3
- "XML schema, DTD, and external entity management using the XML schema repository (XSR)" on page 133

Related tasks:

- "Decomposing XML documents with annotated XML schemas" on page 236
- "Registering and enabling XML schemas for decomposition" on page 237

Related reference:

- "Native XML data store roadmap" in *DB2 Information Center*
- "DECOMPOSE XML DOCUMENT " on page 242
- "Annotated XML schema decomposition - Limits and restrictions" on page 317

- “xdbDecompXML stored procedures for annotated schema decomposition” on page 240
- “XML decomposition annotations - Summary” on page 246

Advantage of annotated XML schema decomposition

Annotated XML schema decomposition can be a solution for storing XML documents that conform to an XML schema, but where the XML schema does not readily match the definitions of the tables into which you are storing the documents. In cases where the XML schema does not clearly match the table structure, you might need to adjust either the XML schema or the relational schema, or both, in order for the documents to fit the table structure. Changes to the XML or relational schema, however, might not always be possible or could be prohibitively expensive, especially if existing applications expect the relational schema to have a particular structure.

Annotated XML schema decomposition addresses this problem by enabling you to decompose documents that are based on new or existing XML schemas, into new or existing tables. This is possible because of the various features available in annotated XML schema decomposition. These features, which are expressed as annotations added to the XML schema documents, offer flexibility in mapping an XML schema structure to a relational table structure.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “XML decomposition annotations - Overview” on page 244

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Decomposing XML documents with annotated XML schemas

When you want to store pieces of an XML document in columns of one or more tables, you can use annotated XML schema decomposition. This type of decomposition breaks an XML document down for storage in tables, based on the annotations specified in a registered annotated XML schema.

Prerequisites:

All tables and columns referenced in the set of annotated schema documents that compose the XML schema must exist in the database before the schema documents are registered with the XML schema repository (XSR).

Procedure:

To decompose XML documents using annotated XML schemas:

1. If using a database created from earlier versions of DB2 database products, including any pre-GA releases of DB2 Version 9.1, run the BIND command using the list file `xdb.lst`, which is found in the `sqllib/bnd` directory.
2. Annotate the schema documents with XML decomposition annotations.
3. Register the schema documents and enable the schema for decomposition.

4. If any of the registered schema documents that belongs to the XML schema have changed, then all documents for this XML schema must be registered again and the XML schema must be enabled for decomposition.
5. Decompose the XML document with either of the following methods, providing the XSR object name for the XML schema:
 - Call the `xdbDecompXML` stored procedure that is just large enough for the size of the document being decomposed.
 - Issue the `DECOMPOSE XML DOCUMENT` command.

Note: If you are using scripts or applications to decompose several documents of which the size is unknown, consider using the `DECOMPOSE XML DOCUMENT` command to decompose, rather than the `xdbDecompXML` stored procedure, as the command automatically calls the stored procedure appropriate for the size of the document.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Disablement of annotated XML schema decomposition” on page 239
- “Troubleshooting considerations for annotated XML schema decomposition” on page 319
- “XML decomposition annotations - Overview” on page 244
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133

Related tasks:

- “Registering and enabling XML schemas for decomposition” on page 237

Related reference:

- “`BIND` command” in *Command Reference*
- “`DECOMPOSE XML DOCUMENT`” on page 242
- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “`xdbDecompXML` stored procedures for annotated schema decomposition” on page 240

Registering and enabling XML schemas for decomposition

To decompose XML documents using annotated XML schemas, the annotated schema must first be registered with the XML Schema Repository (XSR) and enabled for decomposition. This topic describes how to register and enable XML schemas for schema-based decomposition.

Prerequisites:

- Ensure that at least one element or attribute declaration in the XML schema is annotated with an XML decomposition annotation. This annotated element or attribute must be a descendant of, or itself be, a global element of complex type.
- Ensure that the `applheapsz` configuration parameter is set to at least 1024.

Procedure:

Choose one of the following methods to register and enable XML schemas for decomposition:

Stored procedures:

1. Call the XSR_REGISTER stored procedure, passing in the primary schema document.
2. If the XML schema consists of more than one schema document, call the XSR_ADDSCHEMADOC stored procedure for each of the schema documents that have not yet been registered.
3. Call the XSR_COMPLETE stored procedure with the *isusedfordecomposition* parameter set to 1.

Commands:

1. If the XML schema consists of only one schema document, issue the REGISTER XML SCHEMA command with the COMPLETE and ENABLE DECOMPOSITION options.
2. If the XML schema consists of more than one schema document:
 - a. For each schema document except the last, issue the REGISTER XML SCHEMA command.
 - b. For the last schema document that has not yet been registered, issue the REGISTER XML SCHEMA command with the COMPLETE and ENABLE DECOMPOSITION options.

JDBC interface:

1. Call the DB2Connection.registerDB2XMLSchema method and set the *isUsedForDecomposition* boolean parameter to true to enable decomposition. This method exists in two forms: one form for XML schema documents that are input from an InputStream object, and one form for XML schema documents that are in a String.

Notes:

1. If the XML schema was previously registered using any of these methods, but not enabled for decomposition, you can enable the schema for decomposition by issuing the ALTER XSROBJECT SQL statement with the ENABLE DECOMPOSITION option.
2. When an XML schema is enabled for decomposition, a dependency is created between each table referenced in the schema and the XSR object that corresponds to this schema. This dependency prevents any table referenced in the schema from being renamed. The XSR object for the XML schema must be disabled for decomposition in order for a referenced table to be renamed. The tables referenced by XSR objects can be found in the SYSCAT.XSROBJECTDEP catalog view.

Once an annotated schema has been successfully registered and enabled for decomposition, you can decompose XML documents.

Related concepts:

- “XSR_ADDSCHEMADOC procedure” on page 142
- “XSR_COMPLETE procedure” on page 143
- “XSR_REGISTER procedure” on page 141
- “Annotated XML schema decomposition” on page 235

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Related reference:

- “ALTER XSROBJECT statement” in *SQL Reference, Volume 2*
- “REGISTER XMLSCHEMA ” on page 147
- “SYSCAT.XSROBJECTDEP catalog view” in *SQL Reference, Volume 1*
- “applheapsz - Application heap size configuration parameter” in *Performance Guide*

Disabling of annotated XML schema decomposition

Annotated XML schema decomposition can be made inoperative by DB2 under certain conditions or explicitly disabled by a user.

Conditions that make decomposition inoperative:

Schema-based decomposition is automatically made inoperative if any of the following conditions are satisfied, for annotated schemas that have been previously registered and enabled for decomposition. (Note that an XML schema that is made inoperative for decomposition can still be used for validation performed outside of the context of decomposition, such as with the XMLVALIDATE SQL/XML function.) The corrective action needed to re-enable decomposition is listed for each condition.

Table 43. Conditions that make decomposition inoperative and corresponding corrective action

Condition	Action to re-enable decomposition
Table referenced in annotation is dropped	Remove the reference to the dropped table from the schema document, re-register the entire annotated schema, and enable the schema for decomposition
Column referenced in annotation has its data type changed to a type that is compatible with the XML schema type	Re-enable the schema for decomposition by executing the ALTER XSROBJECT SQL statement with the ENABLE DECOMPOSITION option
Column referenced in annotation has its data type changed to a type that is incompatible with the XML schema type	Adjust the annotations as required, re-register the entire annotated schema, and enable the schema for decomposition
Document that belongs to the annotated schema changes	Re-register all of the documents that form that schema and enable the schema for decomposition

For more information, refer to the task documentation on registering annotated schemas and enabling decomposition.

Explicit disablement:

You can explicitly disable schema-based decomposition by executing either of the following SQL statements, specifying the XSR object that corresponds to the annotated schema you want to disable:

- ALTER XSROBJECT with the DISABLE DECOMPOSITION option

Note: An XML schema that is disabled for decomposition can still be used for validation.

- DROP with the XSROBJECT option

Note: Which option to choose depends on what the XML schema is needed for. If the schema is needed for validation, then it should be disabled for decomposition, rather than dropped. If the schema is used only for decomposition, and you do not expect to use it again for decomposition, then you can drop the XSR object.

Related concepts:

- “XSR objects” on page 133
- “Annotated XML schema decomposition” on page 235

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236
- “Registering and enabling XML schemas for decomposition” on page 237

Related reference:

- “ALTER XSROBJECT statement” in *SQL Reference, Volume 2*
- “DROP statement” in *SQL Reference, Volume 2*
- “XMLVALIDATE scalar function” in *SQL Reference, Volume 1*

xdbDecompXML stored procedures for annotated schema decomposition

Annotated XML schema decomposition can be invoked by calling one of six stored procedures:

- xdbDecompXML
- xdbDecompXML10MB
- xdbDecompXML25MB
- xdbDecompXML50MB
- xdbDecompXML75MB
- xdbDecompXML100MB

These stored procedures differ only in the size of the *xmldoc* argument, which specifies the size of the input document to be decomposed. Call the stored procedure that is just large enough for the size of the document you want to decompose, to minimize system memory usage. For example, to decompose a document 1MB in size, use the xdbDecompXML stored procedure.

The syntax for xdbDecompXML is presented below; refer to the description of the *xmldoc* argument for specifications of the *xmldoc* argument for the xdbDecompXML10MB, xdbDecompXML25MB, xdbDecompXML50MB, xdbDecompXML75MB, and xdbDecompXML100MB stored procedures.

Syntax:

```
►► xdbDecompXML (—rschema—, —xmlschemaname—, —xmldoc—, —documentid—, —  
► validation—, —reserved—, —reserved—, —reserved—) ►►
```

The schema of the stored procedure is SYSPROC.

The xdbDecompXML procedure decomposes a single XML document. The procedure executes with the read stability isolation level.

The stored procedure runs atomically; if it fails during execution, all operations performed by the stored procedure are rolled back. To commit the changes made by `xdbDecompXML`, the caller must execute the `COMMIT SQL` statement, as the stored procedure itself does not execute `COMMIT`.

The authorization ID belonging to the statement that calls this stored procedure must have one of the following:

- `CONTROL` privilege on all target tables referenced in the set of annotated schema documents
- `SYSADM` or `DBADM` authority

or all of the following:

- `INSERT` privilege on all target tables referenced in the annotated schema
- `SELECT`, `INSERT`, `UPDATE`, or `DELETE` privilege, as applicable, on any table referenced by the `db2-xdb:expression` or `db2-xdb:condition` annotation

rschema

An input argument of type `VARCHAR(128)` that specifies the SQL schema part of the two-part XSR object name registered with the XML schema repository. If this value is `NULL`, then the SQL schema part is assumed to be the current value of the `CURRENT SCHEMA` special register.

xmlschemaname

An input argument of type `VARCHAR(128)` that specifies the schema name of the two-part XSR object name registered with the XML schema repository. This value cannot be `NULL`.

xmldoc

An input argument of type `BLOB(1M)` that specifies the buffer containing the XML document to be decomposed.

Note:

- For the `xdbDecompXML10MB` stored procedure, this argument is of type `BLOB(10M)`.
- For the `xdbDecompXML25MB` stored procedure, this argument is of type `BLOB(25M)`.
- For the `xdbDecompXML50MB` stored procedure, this argument is of type `BLOB(50M)`.
- For the `xdbDecompXML75MB` stored procedure, this argument is of type `BLOB(75M)`.
- For the `xdbDecompXML100MB` stored procedure, this argument is of type `BLOB(100M)`.

documentid

An input argument of type `VARCHAR(1024)` that specifies the identifier for the input XML document to be decomposed. The value provided in this argument will be substituted for any use of `$DECOMP_CONTENT` specified in the `db2-xdb:expression` or `db2-xdb:condition` annotations in the corresponding XML schema.

validation

An input argument of type `INTEGER` that indicates whether validation will be performed on the document before it is decomposed. The possible values are:

- 0** Validation is not performed on the input document before it is decomposed.
- 1** Validation is performed on the input document against DTDs or XML

schema documents previously registered with the XML schema repository. The input XML document is only decomposed if the validation is successful.

reserved

The *reserved* arguments are input arguments reserved for future use. The values passed for these arguments must be NULL.

Output:

There is no explicit output argument for this stored procedure. Check the sqlcode field of the SQLCA structure for any errors that might have occurred during decomposition. The possible sqlcode values following the completion of decomposition are:

0 Document was successfully decomposed.

positive integer

Document was successfully decomposed, but with warning conditions. The warnings are logged in the db2diag.log file, located in the first failure data capture (FFDC) storage directory.

negative integer

Document could not be decomposed. The sqlcode indicates the reason for the failure. Check the db2diag.log file for details of the failure.

Related concepts:

- “Authorization and privileges” in *SQL Reference, Volume 1*
- “Table and view privileges” in *Administration Guide: Implementation*
- “Annotated XML schema decomposition” on page 235
- “Analyzing db2diag.log files using db2diag” in *Troubleshooting Guide*

Related reference:

- “COMMIT statement” in *SQL Reference, Volume 2*
- “CURRENT SCHEMA special register” in *SQL Reference, Volume 1*
- “db2-xdb:condition decomposition annotation” on page 266
- “db2-xdb:expression decomposition annotation” on page 263
- “Keywords for annotated XML schema decomposition” on page 284

DECOMPOSE XML DOCUMENT

This command invokes a stored procedure to decompose a single XML document using a registered and decomposition-enabled XML schema..

Authorization:

One of the following groups of privileges or authorities is required:

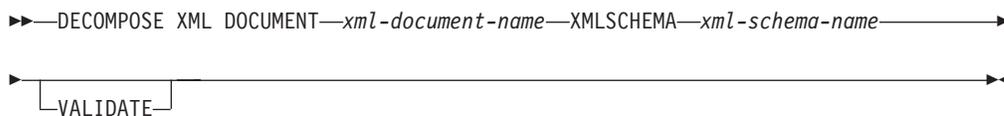
- All of the following privileges:
 - INSERT privileges on the target table, as required for the operation specified in the action file
 - SELECT, INSERT, UPDATE or DELETE privileges as required, on any table referenced in the db2-xdb:expression or db2-xdb:condition annotation
- One of the following privileges or authorities:
 - CONTROL privilege on the target table

- SYSADM or DBADM authority

Required connection:

Database.

Command syntax:



Command parameters:

DECOMPOSE XML DOCUMENT *xml-document-name*

xml-document-name is the file path and file name of the input XML document to be decomposed.

XMLSCHEMA *xml-schema-name*

xml-schema-name is the name of an existing XML schema registered with the XML schema repository to be used for document decomposition. *xml-schema-name* is a qualified SQL identifier consisting of an optional SQL schema name followed by a period and the XML schema name. If the SQL schema name is not specified, it is assumed to be the value of the DB2 special register CURRENT SCHEMA.

VALIDATE

This parameter indicates that the input XML document is to be validated first, then decomposed only if the document is valid. If VALIDATE is not specified, the input XML document will not be validated before decomposition.

Examples:

The following example specifies that the XML document `~/gb/document1.xml` is to be validated and decomposed with the registered XML schema `DB2INST1.GENBANKSCHEMA`.

```
DECOMPOSE XML DOCUMENT ./gb/document1.xml
XMLSCHEMA DB2INST1.GENBANKSCHEMA
VALIDATE
```

The following example specifies that the XML document `./gb/document2.xml` is to be decomposed without validation with the registered XML schema `DB2INST2."GENBANK SCHEMA1"`, on the assumption that the value of the DB2 special register CURRENT SCHEMA is set to `DB2INST2`.

```
DECOMPOSE XML DOCUMENT ./gb/document2.xml
XMLSCHEMA "GENBANK SCHEMA1"
```

Related concepts:

- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133

Annotations

XML decomposition annotations - Overview

The annotated XML schema decomposition process relies on annotations added to XML schema documents. These decomposition annotations function as mappings between the elements or attributes of the XML document to their target tables and columns in the database. The decomposition process refers to these annotations to determine how to decompose an XML document.

The XML decomposition annotations belong to the `http://www.ibm.com/xmlns/prod/db2/xdb1` namespace and are identified by the "db2-xdb" prefix throughout the documentation. You can select your own prefix; however, if you do, you must bind your prefix to the following namespace: `http://www.ibm.com/xmlns/prod/db2/xdb1`. The decomposition process recognizes only annotations that are under this namespace at the time the XML schema is enabled for decomposition.

The decomposition annotations are only recognized by the decomposition process if they are added to element and attribute declarations, or as global annotations, in the schema document. They are either specified as attributes or as part of an `<xs:annotation>` child element of the element or attribute declaration. Annotations added to complex types, references, or other XML schema constructs are ignored.

Although these annotations exist in the XML schema documents, they do not affect the original structure of the schema document, nor do they participate in the validation of XML documents. They are only referred to by the XML decomposition process.

Two annotations that are core features of the decomposition process are: `db2-xdb:rowSet` and `db2-xdb:column`. These annotations specify the decomposed value's target table and column, respectively. These two annotations must be specified in order for the decomposition process to successfully complete. Other annotations are optional, but can be used for further control of how the decomposition process operates.

Related concepts:

- "Annotated XML schema decomposition" on page 235
- "Checklist for annotated XML schema decomposition" on page 291
- "XML decomposition annotations - Specification and scope" on page 244

Related reference:

- "XML decomposition annotations - Summary" on page 246

XML decomposition annotations - Specification and scope

Annotations for decomposition can be specified as either element or attribute declarations in an XML schema document. Annotations can be specified as either:

- a simple attribute on an element or attribute declaration, or
- a structured (complex) child element of an element or attribute declaration, which consists of simple elements and attributes

Annotations as attributes:

Annotations specified as simple attributes on element or attribute declarations apply only to that element or attribute on which it is specified.

For example, the `db2-xdb:rowSet` and `db2-xdb:column` decomposition annotations can be specified as attributes. These annotations would be specified as follows:

```
<xs:element name="isbn" type="xs:string"
            db2-xdb:rowSet="TEXTBOOKS" db2-xdb:column="ISBN"/>
```

The `db2-xdb:rowSet` and `db2-xdb:column` annotations apply only to this element named `isbn`.

Annotations as structured child elements:

Annotations specified as structured children elements of an element or attribute declaration must be specified in the schema document within the `<xs:annotation><xs:appinfo></xs:appinfo></xs:annotation>` hierarchy defined by XML Schema.

For example, the `db2-xdb:rowSet` and `db2-xdb:column` annotations can be specified as children elements (they are children of the `<db2-xdb:rowSetMapping>` annotation) as follows:

```
<xs:element name="isbn" type="xs:string">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetMapping>
        <db2-xdb:rowSet>TEXTBOOKS</db2-xdb:rowSet>
        <db2-xdb:column>ISBN</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

Specifying the `db2-xdb:rowSet` and `db2-xdb:column` annotations as children elements is equivalent to specifying these annotations as attributes (which was shown previously). While more verbose than the method of specifying annotations as attributes, specifying annotations as children elements is required when you need to specify more than one `<db2-xdb:rowSetMapping>` for an element or attribute; that is, when you need to specify multiple mappings on the same element or attribute declaration.

Global annotations:

When an annotation is specified as a child of the `<xs:schema>` element, it is a global annotation that applies to all of the XML schema documents that make up the XML schema.

For example, the `<db2-xdb:defaultSQLSchema>` annotation indicates the default SQL schema for all unqualified tables referenced in the XML schema.

`<db2-xdb:defaultSQLSchema>` must be specified as a child element of `<xs:schema>`:

```
<xs:schema>
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:defaultSQLSchema>admin</db2-xdb:defaultSQLSchema>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>
```

This declaration specifies that all unqualified tables across all schema documents that form this XML schema will have the SQL schema of "admin".

Refer to the documentation for a specific annotation to determine how a particular annotation can be specified.

Related concepts:

- "Annotated XML schema decomposition" on page 235
- "XML decomposition annotations - Overview" on page 244

Related reference:

- "db2-xdb:column decomposition annotation" on page 257
- "db2-xdb:defaultSQLSchema decomposition annotation" on page 247
- "db2-xdb:rowSet decomposition annotation" on page 249
- "db2-xdb:rowSetMapping decomposition annotation" on page 280
- "XML decomposition annotations - Summary" on page 246

XML decomposition annotations - Summary

DB2 supports a set of annotations used by the annotated XML schema decomposition process to map elements and attributes from an XML document to target database tables. The following summary of some of the XML decomposition annotations is grouped by the tasks and actions you use the annotations to perform. For more information about a specific annotation, refer to the detailed documentation about it.

Table 44. Specifying the SQL schema

Action	XML decomposition annotation
Specify the default SQL schema for all tables that do not specify their SQL schema	db2-xdb:defaultSQLSchema
Specify an SQL schema different from the default for a specific table	db2-xdb:table (<db2-xdb:SQLSchema> child element)

Table 45. Mapping XML elements or attributes to target base tables

Action	XML decomposition annotation
Map a single element or attribute to single column and table pair	db2-xdb:rowSet with db2-xdb:column as attribute annotations or db2-xdb:rowSetMapping
Map a single element or attribute to one or more distinct column and table pairs	db2-xdb:rowSetMapping
Map multiple elements or attributes to single column and table pair	db2-xdb:table

Table 46. Specifying the XML data to be decomposed

Action	XML decomposition annotation
Specify the type of content to be inserted for an element of complex type (text, string, or markup)	db2-xdb:contentHandling

Table 46. Specifying the XML data to be decomposed (continued)

Action	XML decomposition annotation
Specify any content transformation to be applied before insertion	db2-xdb:normalization db2-xdb:expression db2-xdb:truncate
Filter the data to be decomposed based on the item's content or the context in which it appears	db2-xdb:condition db2-xdb:locationPath

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “db2-xdb:column decomposition annotation” on page 257
- “db2-xdb:condition decomposition annotation” on page 266
- “db2-xdb:contentHandling decomposition annotation” on page 269
- “db2-xdb:defaultSQLSchema decomposition annotation” on page 247
- “Schema for XML decomposition annotations” on page 322
- “db2-xdb:expression decomposition annotation” on page 263
- “db2-xdb:locationPath decomposition annotation” on page 259
- “db2-xdb:normalization decomposition annotation” on page 274
- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “db2-xdb:table decomposition annotation” on page 254
- “db2-xdb:truncate decomposition annotation” on page 278

db2-xdb:defaultSQLSchema decomposition annotation

The db2-xdb:defaultSQLSchema annotation specifies the default SQL schema for all table names referenced in the XML schema that are not explicitly qualified using the db2-xdb:table annotation.

<db2-xdb:defaultSQLSchema> belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Child element of <xs:appinfo> that is a child of a global <xs:annotation> element.

How to specify:

db2-xdb:defaultSQLSchema is specified in the following way (where *value* represents a valid value for the annotation):

```
<xs:schema>
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:defaultSQLSchema>value</db2-xdb:defaultSQLSchema>
    
```

```

    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>

```

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

Either an ordinary or delimited SQL schema name. Ordinary, or undelimited, SQL schema names are case-insensitive. To specify a delimited SQL schema, use quotation marks that are normally used to delimit SQL identifiers. SQL schema names that contain the special characters '<' and '&' must be escaped in the XML schema document.

Details:

All tables referenced in annotated schemas must be qualified with their SQL schema. Tables can be qualified in two ways, either by explicitly specifying the <db2-xdb:SQLSchema> child element of the <db2-xdb:table> annotation or by using the <db2-xdb:defaultSQLSchema> global annotation. For any unqualified table name, the value specified in <db2-xdb:defaultSQLSchema> is used as its SQL schema name. If multiple schema documents in an annotated schema specify this annotation, all values must be the same.

Example:

The following example shows how the ordinary, or undelimited, SQL identifier admin is defined as the SQL schema for all unqualified tables in the annotated schema:

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2/xdb1">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:defaultSQLSchema>admin</db2-xdb:defaultSQLSchema>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>

```

The following example shows how the delimited SQL identifier admin schema is defined as the SQL schema for all unqualified tables in the annotated schema. Note that admin schema must be delimited with quotation marks:

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2/xdb1">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:defaultSQLSchema>"admin schema"</db2-xdb:defaultSQLSchema>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>

```

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Schemas” in *SQL Reference, Volume 1*

- “XML decomposition annotations - Overview” on page 244
- “XML decomposition annotations - Specification and scope” on page 244

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Related reference:

- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “db2-xdb:table decomposition annotation” on page 254
- “Identifiers” in *SQL Reference, Volume 1*
- “Schema for XML decomposition annotations” on page 322

db2-xdb:rowSet decomposition annotation

The db2-xdb:rowSet annotation specifies an XML element or attribute mapping to a target base table.

db2-xdb:rowSet belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Attribute of <xs:element> or <xs:attribute>, or child element of <db2-xdb:rowSetMapping>

How to specify:

db2-xdb:rowSet is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element db2-xdb:rowSet="*value*" />
- <xs:attribute db2-xdb:rowSet="*value*" />
- <db2-xdb:rowSetMapping>
 <db2-xdb:rowSet>*value*</db2-xdb:rowSet>
 ...
 </db2-xdb:rowSetMapping>

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

Any identifier that adheres to the rules for SQL identifiers. Refer to the identifiers documentation for more information.

Details:

The db2-xdb:rowSet annotation maps an XML element or attribute to a target base table. This annotation can either identify a table name directly, or identify a rowSet name in more complex mappings, where the rowSet is then associated with a table name through the db2-xdb:table annotation. In simple mappings, this annotation

specifies the name of the table the value is to be decomposed into. In more complex mappings, where multiple rowSets (each with a distinct name) map to the same table, then this annotation names the rowSet, rather than the table name.

The target base table into which this XML element's or attribute's value will be decomposed is determined by the presence of other annotations in the set of schema documents that form the annotated schema:

- If the value of `db2-xdb:rowSet` does not match any of the `<db2-xdb:rowSet>` children elements of the `<db2-xdb:table>` global annotation, then the name of the target table is the value specified by this annotation, qualified by an SQL schema defined by the `<db2-xdb:defaultSQLSchema>` global annotation. This usage of `db2-xdb:rowSet` is for the case in which, for a particular table, there is only one set of elements or attributes that maps to the table.
- If the value of `db2-xdb:rowSet` matches a `<db2-xdb:rowSet>` child element of the `<db2-xdb:table>` global annotation, then the name of the target table is the table named in the `<db2-xdb:name>` child of `<db2-xdb:table>`. This usage of `db2-xdb:rowSet` is for the more complex case in which, for a particular table, there are multiple (possibly overlapping) sets of elements or attributes that map to that table.

Important: Ensure that the table that this annotation refers to exists in the database when the XML schema is registered with the XML schema repository. (The columns specified in the `db2-xdb:column` annotations must also exist when registering the XML schema.) If the table does not exist, then an error is returned when the XML schema is enabled for decomposition. If `<db2-xdb:table>` specifies an object other than a table, then an error is returned as well.

When the `db2-xdb:rowSet` annotation is used, either the `db2-xdb:column` annotation or the `db2-xdb:condition` annotation must be specified. The combination of `db2-xdb:rowSet` and `db2-xdb:column` describe the table and column to which this element or attribute will be decomposed into. The combination of `db2-xdb:rowSet` and `db2-xdb:condition` specifies the condition that must be true for any rows of that rowSet to be inserted into the table (referred to either directly, or indirectly through the `<db2-xdb:table>` annotation).

Example:

The two ways of using `db2-xdb:rowSet` listed above, are demonstrated next.

Single set of elements or attributes mapped to a table:

Assume for the following section of an annotated schema that the `BOOKCONTENTS` table belongs to the SQL schema specified by `<db2-xdb:defaultSQLSchema>`, and that there is no global `<db2-xdb:table>` element present which has a `<db2-xdb:rowSet>` child element that matches "BOOKCONTENTS".

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="authorID" type="xs:integer" />
      <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="isbn" type="xs:string"
      db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="ISBN" />
    <xs:attribute name="title" type="xs:string" />
  </xs:complexType>
```

```

</xs:element>

<xs:complexType name="chapterType">
  <xs:sequence>
    <xs:element name="paragraph" type="paragraphType" maxOccurs="unbounded"
      db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="CHPTCONTENT" />
  </xs:sequence>
  <xs:attribute name="number" type="xs:integer"
    db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="CHPTNUM" />
  <xs:attribute name="title" type="xs:string"
    db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="CHPTTITLE" />
</xs:complexType>

<xs:simpleType name="paragraphType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

```

Consider the following element from an XML document:

```

<book isbn="1-11-111111-1" title="My First XML Book">
  <authorID>22</authorID>
  <!-- this book does not have a preface -->
  <chapter number="1" title="Introduction to XML">
    <paragraph>XML is fun...</paragraph>
    ...
  </chapter>
  <chapter number="2" title="XML and Databases">
    <paragraph>XML can be used with...</paragraph>
  </chapter>
  ...
  <chapter number="10" title="Further Reading">
    <paragraph>Recommended tutorials...</paragraph>
  </chapter>
  ...
</book>

```

The BOOKCONTENTS table is then populated as follows:

Table 47. BOOKCONTENTS

ISBN	CHPTNUM	CHPTTITLE	CHPTCONTENT
1-11-111111-1	1	Introduction to XML	XML is fun...
1-11-111111-1	2	XML and Databases	XML can be used with...
...
1-11-111111-1	10	Further Reading	Recommended tutorials...

Multiple sets of elements or attributes mapped to the same table:

For the case where there exists a <db2-xdb:rowSet> child element of the <db2-xdb:table> global annotation that matches the value specified in the db2-xdb:rowSet annotation, the element or attribute is mapped to a table through the <db2-xdb:table> annotation. Assume for the following section of an annotated schema that the ALLBOOKS table belongs to the SQL schema specified by <db2-xdb:defaultSQLSchema>.

```

<!-- global annotation -->
<xs:annotation>
  <xs:appinfo>
    <db2-xdb:table>
      <db2-xdb:name>ALLBOOKS</db2-xdb:name>
      <db2-xdb:rowSet>book</db2-xdb:rowSet>

```

```

        <db2-xdb:rowSet>textbook</db2-xdb:rowSet>
    </db2-xdb:table>
</xs:appinfo>
</xs:annotation>

<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="authorID" type="xs:integer"
        db2-xdb:rowSet="book" db2-xdb:column="AUTHORID" />
      <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="isbn" type="xs:string"
      db2-xdb:rowSet="book" db2-xdb:column="ISBN" />
    <xs:attribute name="title" type="xs:string"
      db2-xdb:rowSet="book" db2-xdb:column="TITLE" />
  </xs:complexType>
</xs:element>
<xs:element name="textbook">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="isbn" type="xs:string"
        db2-xdb:rowSet="textbook" db2-xdb:column="ISBN" />
      <xs:element name="title" type="xs:string"
        db2-xdb:rowSet="textbook" db2-xdb:column="TITLE" />
      <xs:element name="primaryauthorID" type="xs:integer"
        db2-xdb:rowSet="textbook" db2-xdb:column="AUTHORID" />
      <xs:element name="coauthorID" type="xs:integer"
        minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="subject" type="xs:string" />
      <xs:element name="edition" type="xs:integer" />
      <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="chapterType">
  <xs:sequence>
    <xs:element name="paragraph" type="paragraphType" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="number" type="xs:integer" />
  <xs:attribute name="title" type="xs:string" />
</xs:complexType>

<xs:simpleType name="paragraphType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

```

Consider the following elements from an XML document:

```

<book isbn="1-11-111111-1" title="My First XML Book">
  <authorID>22</authorID>
  <!-- this book does not have a preface -->
  <chapter number="1" title="Introduction to XML">
    <paragraph>XML is fun...</paragraph>
  </chapter>
  <chapter number="2" title="XML and Databases">
    <paragraph>XML can be used with...</paragraph>
  </chapter>
  <chapter number="10" title="Further Reading">
    <paragraph>Recommended tutorials...</paragraph>
  </chapter>
</book>

<textbook>
  <isbn>0-11-011111-0</isbn>
  <title>Programming with XML</title>

```

```

<primaryauthorID>435</primaryauthorID>
<subject>Programming</subject>
<edition>4</edition>
<chapter number="1" title="Programming Basics">
  <paragraph>Before you being programming...</paragraph>
</chapter>
<chapter number="2" title="Writing a Program">
  <paragraph>Now that you have learned the basics...</paragraph>
</chapter>
...
<chapter number="10" title="Advanced techniques">
  <paragraph>You can apply advanced techniques...</paragraph>
</chapter>
</textbook>

```

In this example, there are two sets of elements or attributes that map to the table ALLBOOKS:

- /book/@isbn, /book/@authorID, /book/title
- /textbook/isbn, /textbook/primaryauthorID, /textbook/title

The sets are distinguished by associating different rowSet names to each.

Table 48. ALLBOOKS

ISBN	TITLE	AUTHORID
1-11-111111-1	My First XML Book	22
0-11-011111-0	Programming with XML	435

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Schemas” in *SQL Reference, Volume 1*
- “XML decomposition annotations - Overview” on page 244
- “XML decomposition annotations - Specification and scope” on page 244
- “XML schema, DTD, and external entity management using the XML schema repository (XSR)” on page 133
- “rowSets in annotated XML schema decomposition” on page 294

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Related reference:

- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “db2-xdb:column decomposition annotation” on page 257
- “db2-xdb:defaultSQLSchema decomposition annotation” on page 247
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “db2-xdb:table decomposition annotation” on page 254
- “Schema for XML decomposition annotations” on page 322
- “Identifiers” in *SQL Reference, Volume 1*

db2-xdb:table decomposition annotation

The <db2-xdb:table> annotation maps multiple XML elements or attributes to the same target column; or enables you to specify a target table that has an SQL schema different from the default SQL schema specified by <db2-xdb:defaultSQLSchema>.

<db2-xdb:table> belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Global child element of <xs:appinfo> (which is a child element of <xs:annotation>)

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid structure:

The following are supported children elements of <db2-xdb:table>, listed in the order in which they must appear if they are specified:

<db2-xdb:SQLSchema>

(Optional) The SQL schema of the table.

<db2-xdb:name>

The name of the base table. This table name, when qualified with the value of either the preceding <db2-xdb:SQLSchema> annotation or the <db2-xdb:defaultSQLSchema> annotation, must be unique among all <db2-xdb:table> annotations across the set of XML schema documents that form the annotated schema.

<db2-xdb:rowSet>

All elements and attributes that specify the same value for <db2-xdb:rowSet> form a row. Because more than one <db2-xdb:rowSet> element can be specified for the same value of <db2-xdb:name>, more than one set of mappings can be associated with a single table. The combination of the <db2-xdb:rowSet> value with the columns specified in the <db2-xdb:column> annotation allows more than one set of elements or attributes from a single XML document to be mapped to columns of the same table.

At least one <db2-xdb:rowSet> element must be specified, and each <db2-xdb:rowSet> element must be unique among all <db2-xdb:table> annotations across the set of XML schema documents that form the annotated schema, for the annotation to be valid.

Whitespace within the character content of the children elements of <db2-xdb:table> is significant and not normalized. Content of these elements must follow the spelling rules for SQL identifiers. Undelimited values are case-insensitive; for delimited values, quotation marks are used as the delimiter. SQL identifiers that contain the special characters '<' and '&', must be escaped.

Details:

The `<db2-xdb:table>` annotation must be used in either of the following cases:

- when multiple ancestry lines are mapped to the same column of a table (mappings involving single location paths, meaning that there is only one set of column mappings for the table, do not need to use this annotation; the `db2-xdb:rowSet` annotation can be used instead)
- when the table that is to hold the decomposed data is not of the same SQL schema as is defined by the `<db2-xdb:defaultSQLSchema>` annotation.

Only base tables can be specified; other types of tables, such as typed, summary, temporary, or materialized query tables, are not supported for this mapping. Views, table aliases, and nicknames are also not currently permitted for this annotation.

Example:

The following example shows how the `<db2-xdb:table>` annotation can be used to group related elements and attributes together to form a row, when multiple location paths are being mapped to the same column. Consider first the following elements from an XML document (modified slightly from examples used for other annotations).

```
<root>
...
  <book isbn="1-11-111111-1" title="My First XML Book">
    <authorID>22</authorID>
    <email>author22@anyemail.com</email>
    <!-- this book does not have a preface -->
    <chapter number="1" title="Introduction to XML">
      <paragraph>XML is fun...</paragraph>
      ...
    </chapter>
    <chapter number="2" title="XML and Databases">
      <paragraph>XML can be used with...</paragraph>
    </chapter>
    ...
    <chapter number="10" title="Further Reading">
      <paragraph>Recommended tutorials...</paragraph>
    </chapter>
  </book>
  ...
  <author ID="0800" email="author800@email.com">
    <firstname>Alexander</firstname>
    <lastname>Smith</lastname>
    <activeStatus>0</activeStatus>
  </author>
  ...
</root>
```

Assume that the purpose of this decomposition mapping is to insert rows that consist of author IDs and their corresponding email addresses into the same table, `AUTHORSCONTACT`. Notice that author IDs and email addresses appear in both the `<book>` element and the `<author>` element. Thus, more than one location path will need to be mapped to the same columns of the same table. The `<db2-xdb:table>` annotation, therefore, must be used. A section from the annotated schema is presented next, showing how `<db2-xdb:table>` is used to associate multiple paths to the same table.

```
<!-- global annotation -->
<xs:annotation>
  <xs:appinfo>
    <db2-xdb:defaultSQLSchema>adminSchema</db2-xdb:defaultSQLSchema>
    <db2-xdb:table>
```

```

        <db2-xdb:SQLSchema>user1</db2-xdb:SQLSchema>
        <db2-xdb:name>AUTHORSCONTACT</db2-xdb:name>
        <db2-xdb:rowSet>bookRowSet</db2-xdb:rowSet>
        <db2-xdb:rowSet>authorRowSet</db2-xdb:rowSet>
    </db2-xdb:table>
</xs:appinfo>
</xs:annotation>

<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="authorID" type="xs:integer"
        db2-xdb:rowSet="bookRowSet" db2-xdb:column="AUTHID" />
      <xs:element name="email" type="xs:string"
        db2-xdb:rowSet="bookRowSet" db2-xdb:column="EMAILADDR" />
      <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="isbn" type="xs:string" />
    <xs:attribute name="title" type="xs:string" />
  </xs:complexType>
</xs:element>

<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string" />
      <xs:element name="lastname" type="xs:string" />
      <xs:element name="activeStatus" type="xs:boolean" />
    </xs:sequence>
    <xs:attribute name="ID" type="xs:integer"
      db2-xdb:rowSet="authorRowSet" db2-xdb:column="AUTHID" />
    <xs:attribute name="email" type="xs:string"
      db2-xdb:rowSet="authorRowSet" db2-xdb:column="EMAILADDR" />
  </xs:complexType>
</xs:element>

```

The `<db2-xdb:table>` annotation identifies the name of the target table for a mapping with the `db2-xdb:name` child element. In this example, `AUTHORSCONTACT` is the target table. To ensure that the ID and email addresses from the `<book>` element are kept separate from those of the `<author>` element (that is, each row contains logically related values), the `<db2-xdb:rowSet>` element is used to associate related items. Even though in this example, the `<book>` and `<author>` elements are separate entities, there can be cases where the entities to be mapped are not separate and require a logical separation, which can be achieved through the use of rowSets.

Note that the `AUTHORSCONTACT` table exists in an SQL schema different from the default SQL schema, and the `<db2-xdb:SQLSchema>` element is used to specify this. The resulting `AUTHORSCONTACT` table is shown below:

Table 49. *AUTHORSCONTACT*

AUTHID	EMAILADDR
22	author22@anyemail.com
0800	author800@email.com

This example illustrates how the logical grouping of values through rowSets ensure that unrelated values are not unintentionally mapped to the same table and column pair. In this example, `/root/book/authorID` and `/root/author/@ID` are mapped to the same table and column pair. Similarly, `/root/book/email` and `/root/author/@email` are mapped to the same table and column pair. Consider the

case if rowSets were not available. If, for example, the /root/book/email element was not present in an instance of the <author> element and rowSets could not be used, then it would not be possible to determine if the email from the <author> element should be associated with /root/book/authorID or /root/author/@ID or both. Thus the rowSets associated with a single table in the <db2-xdb:table> annotation helps logically distinguish among different sets of rows.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244
- “XML decomposition annotations - Specification and scope” on page 244
- “Schemas” in *SQL Reference, Volume 1*

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Related reference:

- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “db2-xdb:defaultSQLSchema decomposition annotation” on page 247
- “db2-xdb:rowSet decomposition annotation” on page 249
- “Schema for XML decomposition annotations” on page 322

db2-xdb:column decomposition annotation

The db2-xdb:column annotation specifies a column name of the table to which an XML element or attribute has been mapped.

db2-xdb:column belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Attribute of <xs:element> or <xs:attribute>, or child element of <db2-xdb:rowSetMapping>

How to specify:

db2-xdb:column is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element db2-xdb:rowSet="*value*" db2-xdb:column="*value*" />
- <xs:attribute db2-xdb:rowSet="*value*" db2-xdb:column="*value*" />
- <db2-xdb:rowSetMapping>
 <db2-xdb:rowSet>*value*</db2-xdb:rowSet>
 <db2-xdb:column>*value*</db2-xdb:column>
 ...
</db2-xdb:rowSetMapping>

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

Any base table column name that adheres to the following:

- Undelimited column names are case-insensitive. For delimited column names, escape the delimiter with ". For example, to specify a two word column name, "col one", db2-xdb:column would be set as follows:
db2-xdb:column=""col one"";

(Note that these conditions are requirements specific to this annotation.)

- Only columns of the following data types can be specified in this annotation: all data types supported by the CREATE TABLE SQL statement, except DATALINK and user-defined structured types.

Details:

The db2-xdb:column annotation, specified as an attribute in the declaration of an XML element or attribute, or as a child element of <db2-xdb:rowSetMapping>, maps an XML element or attribute to the column name of a target table. When this annotation is used, the db2-xdb:rowSet annotation must be specified as well. Together they describe the table and column that will hold the decomposed value for this element or attribute.

Example:

The following example shows how content from the <book> element can be inserted into columns of a table called BOOKCONTENTS, using the db2-xdb:column annotation. A section of the annotated schema is presented first.

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="authorID" type="xs:integer" />
      <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="isbn" type="xs:string"
      db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="ISBN" />
    <xs:attribute name="title" type="xs:string" />
  </xs:complexType>
</xs:element>

<xs:complexType name="chapterType">
  <xs:sequence>
    <xs:element name="paragraph" type="paragraphType" maxOccurs="unbounded"
      db2-xdb:rowSet="BOOKCONTENTS"
      db2-xdb:column="CHPTCONTENT" />
  </xs:sequence>
  <xs:attribute name="number" type="xs:integer"
    db2-xdb:rowSet="BOOKCONTENTS"
    db2-xdb:column="CHPTNUM" />
  <xs:attribute name="title" type="xs:string"
    db2-xdb:rowSet="BOOKCONTENTS"
    db2-xdb:column="CHPTTITLE" />
</xs:complexType>

<xs:simpleType name="paragraphType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
```

The <book> element that is being mapped is presented next, followed by the resulting BOOKCONTENTS table after the decomposition has completed. .

```
<book isbn="1-11-111111-1" title="My First XML Book">
  <authorID>22</authorID>
  <!-- this book does not have a preface -->
  <chapter number="1" title="Introduction to XML">
    <paragraph>XML is fun...</paragraph>
    ...
  </chapter>
  <chapter number="2" title="XML and Databases">
    <paragraph>XML can be used with...</paragraph>
  </chapter>
  ...
  <chapter number="10" title="Further Reading">
    <paragraph>Recommended tutorials...</paragraph>
  </chapter>
</book>
```

Table 50. BOOKCONTENTS

ISBN	CHPTNUM	CHPTTITLE	CHPTCONTENT
1-11-111111-1	1	Introduction to XML	XML is fun...
1-11-111111-1	2	XML and Databases	XML can be used with...
...
1-11-111111-1	10	Further Reading	Recommended tutorials...

Related concepts:

- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244
- “XML decomposition annotations - Specification and scope” on page 244
- “Annotated XML schema decomposition” on page 235

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Related reference:

- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “Schema for XML decomposition annotations” on page 322
- “CREATE TABLE statement” in *SQL Reference, Volume 2*

db2-xdb:locationPath decomposition annotation

The db2-xdb:locationPath annotation maps an XML element or attribute globally declared or as part of a reusable group, to different table and column pairs, depending on the ancestry of the element or attribute. Reusable groups are globally declared named complex types, named model groups, and named attribute groups.

db2-xdb:locationPath belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements

and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Attribute of <xs:element> or <xs:attribute>, or attribute of <db2-xdb:rowSetMapping>

How to specify:

db2-xdb:locationPath is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element db2-xdb:locationPath="*value*" />
- <xs:attribute db2-xdb:locationPath="*value*" />
- <db2-xdb:rowSetMapping db2-xdb:locationPath="*value*">
 <db2-xdb:rowSet>*value*</db2-xdb:rowSet>
 ...
</db2-xdb:rowSetMapping>

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

The value of db2-xdb:locationPath must have the following syntax:

```
location path := '/' (locationstep '/')* lastlocationstep
locationstep := (prefix:)? name
lastlocationstep := locationstep | '@' (prefix:)? name
```

where name is an element or attribute name, and prefix is a namespace prefix. Note that all namespace prefixes used in the location path must have been associated with a namespace in the schema document that contains the annotation specifying this location path. A namespace prefix binding can be created by adding a namespace declaration to the <xs:schema> element of the schema document.

Details:

The db2-xdb:locationPath annotation is used to describe the mappings for elements or attributes that are either declared globally or as part of either a:

- named model group
- named attribute group
- global complex type declaration
- global element or attribute of simple or complex type

For element or attribute declarations that cannot be reused (local declarations that are not part of named complex type definitions or named model or attribute groups), the db2-xdb:locationPath annotation has no effect.

db2-xdb:locationPath should be used when global element or attribute declarations are used as references from various ancestry lines. (for example: <xs:element ref="abc">). Because annotations cannot be specified directly on references, they must instead be specified on the corresponding global element or attribute declaration. Because the corresponding element or attribute declaration is global,

the element or attribute can be referenced from many different contexts within the XML schema. In general, `db2-xdb:locationPath` should be used to distinguish the mappings in these different contexts. For named complex types, model groups, and attribute groups, the element and attribute declarations should be annotated for each context in which they are mapped for decomposition. The `db2-xdb:locationPath` annotation should be used to specify the target rowSet and column pair for each `locationPath`. The same `db2-xdb:locationPath` value can be used for different rowSet and column pairs.

Note: When both a default namespace and `attributeFormDefault="unqualified"` are specified, then decomposition annotations for unqualified attributes will be ignored. This is because the annotation processing treats the attribute as if it belonged to the default namespace. The `attributeFormDefault="unqualified"` setting, however, indicates that the attribute actually belongs to the global namespace. In this case, the mapping for this attribute is ignored, and no value is inserted.

Example:

The following example shows how the same attribute can be mapped to different tables depending on the context in which this attribute appears. A section of the annotated schema is presented first.

```
<!-- global attribute -->
<xs:attribute name="title" type="xs:string"
  db2-xdb:rowSet="BOOKS"
  db2-xdb:column="TITLE"
  db2-xdb:locationPath="/books/book/@title">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetMapping db2-xdb:locationPath="/books/book/chapter/@title">
        <db2-xdb:rowSet>BOOKCONTENTS</db2-xdb:rowSet>
        <db2-xdb:column>CHPTTITLE</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</xs:attribute>

<xs:element name="books">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="book">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="authorID" type="xs:integer" />
            <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
          </xs:sequence>
          <xs:attribute name="isbn" type="xs:string" />
          <xs:attribute ref="title" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="chapterType">
  <xs:sequence>
    <xs:element name="paragraph" type="paragraphType" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="number" type="xs:integer" />
  <xs:attribute ref="title" />
</xs:complexType>
```

```

<xs:simpleType name="paragraphType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

```

Note that there is only one attribute declaration named "title", but there are two references to this attribute in different contexts. One reference is from the <book> element, and the other is from the <chapter> element. The value of the "title" attribute needs to be decomposed into different tables depending on the context. This annotated schema specifies that a "title" value is decomposed into the BOOKS table if it is a book title and into the BOOKCONTENTS table if it is a chapter title.

The <books> element that is being mapped is presented next, followed by the resulting BOOKS table after the decomposition has completed.

```

<books>
  <book isbn="1-11-111111-1" title="My First XML Book">
    <authorID>22</authorID>
    <!-- this book does not have a preface -->
    <chapter number="1" title="Introduction to XML">
      <paragraph>XML is fun...</paragraph>
      ...
    </chapter>
    <chapter number="2" title="XML and Databases">
      <paragraph>XML can be used with...</paragraph>
    </chapter>
    ...
    <chapter number="10" title="Further Reading">
      <paragraph>Recommended tutorials...</paragraph>
    </chapter>
  </book>
  ...
</books>

```

Table 51. BOOKS

ISBN	TITLE	CONTENT
NULL	My First XML Book	NULL

Table 52. BOOKCONTENTS

ISBN	CHPTNUM	CHPTTITLE	CHPTCONTENT
NULL	NULL	Introduction to XML	NULL
NULL	NULL	XML and Databases	NULL
...
NULL	NULL	Further Reading	NULL

Related concepts:

- "Annotated XML schema decomposition" on page 235
- "XML decomposition annotations - Overview" on page 244
- "XML decomposition annotations - Specification and scope" on page 244

Related tasks:

- "Decomposing XML documents with annotated XML schemas" on page 236

Related reference:

- "Annotated XML schema decomposition - Limits and restrictions" on page 317

- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “Schema for XML decomposition annotations” on page 322

db2-xdb:expression decomposition annotation

The db2-xdb:expression annotation specifies a customized expression, the result of which is inserted into the table this element is mapped to.

db2-xdb:expression belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Attribute of <xs:element> or <xs:attribute>, or optional child element of <db2-xdb:rowSetMapping>, effective only on annotations that include a column mapping

How to specify:

db2-xdb:expression is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element db2-xdb:expression="value" db2-xdb:column="value" />
- <xs:attribute db2-xdb:expression="value" db2-xdb:column="value" />
- <db2-xdb:rowSetMapping>
 - <db2-xdb:rowSet>value</db2-xdb:rowSet>
 - <db2-xdb:column>value</db2-xdb:column>
 - <db2-xdb:expression>value</db2-xdb:expression>
 - ...
</db2-xdb:rowSetMapping>

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

The value of db2-xdb:expression must have the following syntax, which constitutes a subset of SQL expressions:

```
expression := function (arglist) | constant | $DECOMP_CONTENT | $DECOMP_ELEMENTID |
  $DECOMP_DOCUMENTID | (scalar-fullselect) | expression operator expression |
  (expression) | special-register | CAST (expression AS data-type) |
  XMLCAST (expression AS data-type) | XML-function
```

```
operator := + | - | * | / | CONCAT
```

```
arglist := expression | arglist, expression
```

Details:

The db2-xdb:expression annotation enables you to specify a customized expression, which is applied to the content of the XML element or attribute being annotated when \$DECOMP_CONTENT is used. The result of evaluating this expression is then inserted into the column specified during decomposition.

This annotation is also useful in cases where you want to insert constant values (such as the name of an element), or generated values that do not appear in the document.

`db2-xdb:expression` must be specified using valid SQL expressions, and the type of the evaluated expression must be statically determinable and compatible with the type of the target column that the value is to be inserted into. The following subset of SQL expressions are supported; any other SQL expressions not described below are unsupported and have an undefined behavior in the context of this annotation.

function (arglist)

A built-in or user-defined scalar SQL function. The arguments of a scalar function are individual scalar values. A scalar function returns a single value (possibly null). Refer to the documentation on functions for more information.

constant

A value, sometimes called a literal, that is a string constant or a numeric constant. Refer to the documentation on constants for more information.

\$DECOMP_CONTENT

The value of the mapped XML element or attribute from the document, constructed according to the setting of the `db2-xdb:contentHandling` annotation. Refer to the decomposition keywords documentation for more information.

\$DECOMP_ELEMENTID

A system-generated integer identifier that uniquely identifies within the XML document, the element or attribute this annotation describes. Refer to the decomposition keywords documentation for more information.

\$DECOMP_DOCUMENTID

The string value specified in the *documentid* input parameter of the `xdbDecompXML` stored procedure, which identifies the XML document being decomposed. Refer to the decomposition keywords documentation for more information.

(scalar-fullselect)

A fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the NULL value.

expression operator expression

The result of two supported expression operands, as defined in the supported values listing above. Refer to the documentation on expressions for details on expression operations.

(expression)

An expression enclosed in parentheses that conforms to the list of supported expressions defined above.

special-register

The name of a supported special register. This setting evaluates to the value of the special register for the current server. Refer to the documentation for special registers for a complete listing of supported special registers.

CAST (expression AS data-type)

The expression cast to the specified SQL data type, if the expression is not NULL. If the expression is NULL, the result is a null value of the SQL data type specified. When inserting a NULL value into a column, the expression

must cast NULL into a compatible column type (for example: CAST (NULL AS INTEGER), for an integer column).

XMLCAST (expression AS data-type)

The expression cast to the specified data type, if the expression is not NULL. The expression or the target data type must be the XML type. If the expression is NULL, the target type must be XML, and the result is a null XML value.

XML-function

Any supported SQL/XML function.

Example:

The following example shows how the db2-xdb:expression annotation can be used to apply a value from the XML document to a user-defined function. The result returned from the UDF is then inserted into the database, rather than the value from the document itself. A section of the annotated schema is presented first.

```
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string" />
      <xs:element name="lastname" type="xs:string" />
      <xs:element name="activeStatus" type="xs:boolean" />
      <xs:attribute name="ID" type="xs:integer"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="NUMBOOKS"
        db2-xdb:expression="AuthNumBooks (INTEGER ($DECOMP_CONTENT))" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Assume that there is a user-defined function called AuthNumBooks that takes an integer parameter, which represents the author's ID, and returns the total number of books that author has in the system.

The <author> element that is being mapped is presented next.

```
<author ID="22">
  <firstname>Ann</firstname>
  <lastname>Brown</lastname>
  <activeStatus>1</activeStatus>
</author>
```

\$DECOMP_CONTENT is replaced with the value "22" from the instance of the ID attribute. Because \$DECOMP_CONTENT is always substituted with a character type, and because the AuthNumBooks UDF takes an integer parameter, the db2-xdb:expression annotation must cast \$DECOMP_CONTENT to an integer. Assume that the UDF returns the integer 8 for this author whose ID is 22; 8 is then inserted into the NUMBOOKS column of the AUTHORS table, as shown next.

Table 53. AUTHORS

AUTHID	FIRSTNAME	SURNAME	ACTIVE	NUMBOOKS
NULL	NULL	NULL	NULL	8

Related concepts:

- "User-defined functions" in *SQL Reference, Volume 1*
- "Annotated XML schema decomposition" on page 235
- "XML decomposition annotations - Overview" on page 244

- “XML decomposition annotations - Specification and scope” on page 244

Related reference:

- “Constants” in *SQL Reference, Volume 1*
- “Expressions” in *SQL Reference, Volume 1*
- “Special registers” in *SQL Reference, Volume 1*
- “Supported functions and administrative SQL routines and views” in *SQL Reference, Volume 1*
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “Keywords for annotated XML schema decomposition” on page 284
- “Schema for XML decomposition annotations” on page 322

db2-xdb:condition decomposition annotation

The db2-xdb:condition annotation specifies a condition that determines if a row will be inserted into a table. A row that satisfies the condition might be inserted (depending on other conditions for the rowSet, if any); a row that does not satisfy the condition will not be inserted.

db2-xdb:condition belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Attribute of <xs:element> or <xs:attribute>, or optional child element of <db2-xdb:rowSetMapping>. Condition is applied regardless of whether the annotation to which it belongs contains a column mapping.

How to specify:

db2-xdb:condition is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element db2-xdb:condition="*value*" />
- <xs:attribute db2-xdb:condition="*value*" />
- <db2-xdb:rowSetMapping>
 - <db2-xdb:rowSet>*value*</db2-xdb:rowSet>
 - <db2-xdb:condition>*value*</db2-xdb:condition>
 - ...
</db2-xdb:rowSetMapping>

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

SQL predicates of the following types: basic, quantified, BETWEEN, EXISTS, IN, IS VALIDATED, LIKE, NULL, and XMLEXISTS. The predicates must also consist of expressions that are supported by the db2-xdb:expression annotation.

Details:

The `db2-xdb:condition` annotation enables you to specify conditions under which values are inserted into the database during decomposition. This annotation filters rows by applying user-specified conditions. The rows that satisfy the specified conditions are inserted into the database; rows that do not meet the conditions are not inserted during decomposition.

If the `db2-xdb:condition` annotation is specified on multiple element or attribute declarations of the same rowSet, then the row will be inserted only when the logical AND of all conditions evaluate to true.

Column names in `db2-xdb:condition`:

Because `db2-xdb:condition` consists of SQL predicates, column names can be specified in this annotation. If a `db2-xdb:condition` annotation involving a rowSet contains an unqualified column name, there must exist a mapping to that column among all of the mappings involving that rowSet. Other column names, when used in predicates containing SELECT statements, must be qualified. If `db2-xdb:condition` specifies an unqualified column name, but the element or attribute for which `db2-xdb:condition` is specified does not have a column mapping specified, then when the condition is evaluated, the value that is evaluated is the content of the element or attribute that maps to the referenced column name.

Consider the following example:

```
<xs:element name="a" type="xs:string"
  db2-xdb:rowSet="rowSetA" db2-xdb:condition="columnX='abc'" />
<xs:element name="b" type="xs:string"
  db2-xdb:rowSet="rowSetB" db2-xdb:column="columnX" />
```

Notice that `<a>` does not have a column mapping specified, but the condition references the column `"columnX"`. When the condition is evaluated, `"columnX"` in the condition will be replaced with the value from ``, because `` has specified a column mapping for `"columnX"`, while `<a>` does not have a column mapping. If the XML document contained:

```
<a>abc</a>
<b>def</b>
```

then the condition would evaluate to false in this case, because the value from ``, `"def"`, is evaluated in the condition.

If `$DECOMP_CONTENT` (a decomposition keyword that specifies the value of the mapped element or attribute as character data), instead of the column name, is used in the `db2-xdb:condition` attached to the element `<a>` declaration, then the condition is evaluated using the value of `<a>`, rather than ``.

```
<xs:element name="a" type="xs:string"
  db2-xdb:rowSet="rowSetA" db2-xdb:condition="$DECOMP_CONTENT='abc'" />
<xs:element name="b" type="xs:string"
  db2-xdb:rowSet="rowSetB" db2-xdb:column="columnX" />
```

If the XML document contained:

```
<a>abc</a>
<b>def</b>
```

then the condition would evaluate to true in this case, because the value from `<a>`, `"abc"`, is used in the evaluation.

This conditional processing, using column names and \$DECOMP_CONTENT, can be useful in cases where you only want to decompose a value based on the value of another element or attribute that will not be inserted into the database.

Conditions specified on mapped elements or attributes absent from the document:

If a condition is specified on an element or attribute, but that element or attribute does not appear in the XML document, then the condition is still applied. For example, consider the following element mapping from an annotated schema document:

```
<xs:element name="intElem" type="xs:integer"
  db2-xdb:rowSet="rowSetA" db2-xdb:column="colInt"
  db2-xdb:condition="colInt > 100" default="0" />
```

If the <intElem> element does not appear in the XML document, the condition "colInt > 100" is still evaluated. Because <intElem> does not appear, a default value of 0 is used in the condition evaluation for "colInt". The condition is then evaluated as: 0 > 100, which evaluates to false. The corresponding row is therefore not inserted during decomposition.

Example:

Consider the following <author> element from an XML document:

```
<author ID="0800">
  <firstname>Alexander</firstname>
  <lastname>Smith</lastname>
  <activeStatus>1</activeStatus>
</author>
```

Depending on the conditions specified by db2-xdb:condition, the values from this <author> element might or might not be inserted into the target tables during decomposition. Two cases are presented next.

All conditions satisfied:

The following section from the annotated schema that corresponds to the <author> element above, specifies that this element should only be decomposed if the author's ID falls between 1 and 999, the <firstname> and <lastname> elements are not NULL, and the value of the <activeStatus> element equals 1:

```
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="GIVENNAME"
        db2-xdb:condition="$DECOMP_CONTENT IS NOT NULL" />
      <xs:element name="lastname" type="xs:string"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="SURNAME"
        db2-xdb:condition="$DECOMP_CONTENT IS NOT NULL" />
      <xs:element name="activeStatus" type="xs:integer"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="statusCode"
        db2-xdb:condition="$DECOMP_CONTENT=1" />
      <xs:attribute name="ID" type="xs:integer"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="AUTHID"
        db2-xdb:condition="$DECOMP_CONTENT BETWEEN 1 and 999" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Because all of the conditions specified by `db2-xdb:condition` are satisfied by the values in the example `<author>` element above, the `AUTHORS` table is populated with the data from the `<author>` element.

Table 54. *AUTHORS*

AUTHID	GIVENNAME	SURNAME	STATUSCODE	NUMBOOKS
0800	Alexander	Smith	1	NULL

One condition fails:

The following annotated schema specifies that the `<author>` element should only be decomposed if the author's ID falls between 1 and 100, and the `<firstname>` and `<lastname>` elements are not NULL:

```
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="GIVENNAME"
        db2-xdb:condition="$DECOMP_CONTENT IS NOT NULL"/>
      <xs:element name="lastname" type="xs:string"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="SURNAME"
        db2-xdb:condition="$DECOMP_CONTENT IS NOT NULL"/>
      <xs:element name="activeStatus" type="xs:integer" />
      <xs:attribute name="ID" type="xs:integer"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="AUTHID"
        db2-xdb:condition="$DECOMP_CONTENT BETWEEN 1 and 100" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Although the `<firstname>` and `<lastname>` elements of the example `<author>` element meet the conditions specified, the value of the ID attribute does not, and so the entire row is not inserted during decomposition. This is because the logical AND of all three conditions specified on the `AUTHORS` table is evaluated. In this case, one of the conditions is false, and so the logical AND evaluates to false, and therefore, no rows are inserted.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “XML decomposition annotations - Overview” on page 244
- “XML decomposition annotations - Specification and scope” on page 244

Related reference:

- “Predicates” in *SQL Reference, Volume 1*
- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “`db2-xdb:expression` decomposition annotation” on page 263
- “`db2-xdb:rowSetMapping` decomposition annotation” on page 280
- “Keywords for annotated XML schema decomposition” on page 284
- “Schema for XML decomposition annotations” on page 322

db2-xdb:contentHandling decomposition annotation

The `db2-xdb:contentHandling` annotation specifies the type of content that will be decomposed into a table for an element of complex type or simple type.

db2-xdb:contentHandling belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Attribute of <xs:element>, or attribute of <db2-xdb:rowSetMapping>, that applies to complex type or simple type element declarations

How to specify:

db2-xdb:contentHandling is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element db2-xdb:contentHandling="value" />
- <db2-xdb:rowSetMapping db2-xdb:contentHandling="value">
 <db2-xdb:rowSet>value</db2-xdb:rowSet>
 ...
</db2-xdb:rowSetMapping>

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

One of the following case-sensitive tokens:

- text
- stringValue
- serializeSubtree

Details:

The db2-xdb:contentHandling annotation, specified as an attribute in the declaration of an XML element, indicates what value is to be inserted into the tables and columns specified by db2-xdb:rowSet and db2-xdb:column, respectively, during decomposition. The three valid values for db2-xdb:contentHandling are:

text

- What is inserted: the concatenation of character data (including character content of CDATA sections) within this element.
- What is excluded: this element's comments and processing instructions, CDATA section delimiters ("<![CDATA[" "]]>"), as well as this element's descendants (including tags and content).

stringValue

- What is inserted: the concatenation of this element's character data (including character content of CDATA sections) with the character data in this element's descendants, in document order.
- What is excluded: comments, processing instructions, CDATA section delimiters ("<![CDATA[" "]]>"), and the start and end tags of this element's descendants.

serializeSubtree

- What is inserted: the markup of everything between this element's start and end tags, including this element's start and end tags. This includes comments, processing instructions, and CDATA section delimiters ("`<![CDATA[" "]]>`").
- What is excluded: nothing.
- Notes: The serialized string that is inserted might not be identical to the corresponding section in the XML document because of factors such as: default values specified in the XML schema, expansion of entities, order of attributes, whitespace normalization of attributes, and processing of CDATA sections.

Because the serialized string that results from this setting is an XML entity, there are code page issues that should be considered. If the target column is of character or graphic types, the XML fragment is inserted in the database's code page. When such an entity is passed by an application to an XML processor, the application must explicitly inform the processor of the entity's encoding, because the processor would not automatically detect encodings other than UTF-8. If the target column is of type BLOB, however, then the XML entity is inserted in UTF-8 encoding. In this case, the XML entity can be passed to the XML processor without needing to specify an encoding.

If an XML element declaration that is annotated for decomposition is of complex type and contains complex content, but does not have `db2-xdb:contentHandling` specified, then the default behavior follows the "serializeSubtree" setting. For all other cases of annotated element declarations, the default behavior if `db2-xdb:contentHandling` is not specified follows the "stringValue" setting.

If an element is declared to be of complex type and has an element-only or empty content model (that is, the "mixed" attribute of the element declaration is not set to true or 1), then `db2-xdb:contentHandling` cannot be set to "text".

Specifying the `db2-xdb:contentHandling` annotation on an element does not affect the decomposition of any of the element's descendants.

The setting of `db2-xdb:contentHandling` affects the value that is substituted for `$DECOMP_CONTENT` in either of the `db2-xdb:expression` or `db2-xdb:condition` annotations. The substituted value is first processed according to the `db2-xdb:contentHandling` setting, before it is passed for evaluation.

Note that if validation has been performed, either prior to decomposition or during, the content being processed by `db2-xdb:contentHandling` will already have its entities resolved.

Example:

The following example illustrates how the different settings of the `db2-xdb:contentHandling` annotation can be used to yield different results in the target table. The annotated schema is presented first, showing how the `<paragraph>` element is annotated with `db2-xdb:contentHandling`. (The annotated schema is only presented once, with `db2-xdb:contentHandling` set to "text". Subsequent examples in this section assume the same annotated schema, which differ only by the value `db2-xdb:contentHandling` is set to.)

```
<xs:schema>
  <xs:element name="books">
    <xs:complexType>
      <xs:sequence>
```

```

    <xs:element name="book">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="authorID" type="xs:integer" />
          <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="isbn" type="xs:string"
          db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="ISBN" />
        <xs:attribute name="title" type="xs:string" />
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>

<xs:complexType name="chapterType">
  <xs:sequence>
    <xs:element name="paragraph" type="paragraphType" maxOccurs="unbounded"
      db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="CHPTCONTENT"
      db2-xdb:contentHandling="text" />
  </xs:sequence>
  <xs:attribute name="number" type="xs:integer"
    db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="CHPTNUM" />
  <xs:attribute name="title" type="xs:string"
    db2-xdb:rowSet="BOOKCONTENTS" db2-xdb:column="CHPTTITLE" />
</xs:complexType>

<xs:complexType name="paragraphType" mixed="1">
  <xs:choice>
    <xs:element name="b" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
  </xs:choice>
</xs:complexType>
</xs:schema>

```

The <books> element that is being mapped is presented next.

```

<books>
  <book isbn="1-11-111111-1" title="My First XML Book">
    <authorID>22</authorID>
    <chapter number="1" title="Introduction to XML">
      <paragraph>XML is <b>lots of</b> fun...</paragraph>
    </chapter>
    <chapter number="2" title="XML and Databases">
      <paragraph><!-- Start of chapter -->XML can be used with...</paragraph>
      <paragraph><?processInstr example?>
        Escape characters such as <![CDATA[ <, >, and & ]]>...</paragraph>
    </chapter>
    ...
    <chapter number="10" title="Further Reading">
      <paragraph>Recommended tutorials...</paragraph>
    </chapter>
  </book>
  ...
</books>

```

The next three tables show the result of decomposing the same XML element with differing values for db2-xdb:contentHandling.

Note: The resulting tables below contain quotation marks around the values in the CHPTTITLE and CHPTCONTENT columns. These quotation marks do not exist in the columns, but are only presented here to show the boundaries and whitespaces of the inserted strings.

db2-xdb:contentHandling="text":

Table 55. BOOKCONTENTS

ISBN	CHPTNUM	CHPTTITLE	CHPTCONTENT
1-11-111111-1	1	"Introduction to XML"	"XML is fun..."
1-11-111111-1	2	"XML and Databases"	"XML can be used with..."
1-11-111111-1	2	"XML and Databases"	" Escape characters such as <, >, and & ..."
...
1-11-111111-1	10	"Further Reading"	"Recommended tutorials..."

Observe how the content of the element of the first paragraph of chapter 1 is not inserted when the "text" setting is used. This is because the "text" setting excludes any content from descendants. Notice also that the comment and processing instruction from the first paragraph of chapter 2 are excluded when the "text" setting is used. Whitespace from the concatenation of character data from the <paragraph> elements is preserved.

db2-xdb:contentHandling="stringValue":

Table 56. BOOKCONTENTS

ISBN	CHPTNUM	CHPTTITLE	CHPTCONTENT
1-11-111111-1	1	"Introduction to XML"	"XML is lots of fun..."
1-11-111111-1	2	"XML and Databases"	"XML can be used with..."
1-11-111111-1	2	"XML and Databases"	" Escape characters such as <, >, and & ..."
...
1-11-111111-1	10	"Further Reading"	"Recommended tutorials..."

The difference between this table and the previous table is found in the CHPTCONTENT column of the first row. Notice how the string "lots of", which comes from the descendant of the <paragraph> element, has been inserted. When db2-xdb:contentHandling was set to "text", this string was excluded, because the "text" setting excludes the content of descendants. The "stringValue" setting, however, includes content from descendants. Like the "text" setting, comments and processing instructions are not inserted, and whitespace is preserved.

db2-xdb:contentHandling="serializeSubtree":

Table 57. BOOKCONTENTS

ISBN	CHPTNUM	CHPTTITLE	CHPTCONTENT
1-11-111111-1	1	"Introduction to XML"	"<paragraph>XML is lots of fun...</paragraph>"
1-11-111111-1	2	"XML and Databases"	"<paragraph><!-- Start of chapter -->XML can be used with...</paragraph>"
1-11-111111-1	2	"XML and Databases"	"<paragraph><?processInstr example?> Escape characters such as <![CDATA[<, >, and &]]>...</paragraph>"
...
1-11-111111-1	10	"Further Reading"	"<paragraph>Recommended tutorials...</paragraph>"

The difference between this table and the previous two tables is that all markup from the descendants of <paragraph> elements are inserted (including the <paragraph> start and end tags). This includes the start and end tags in the CHPTCONTENT column of the first row, as well as the comment and processing instruction in the second and third rows, respectively. As in the previous two examples, whitespace from the XML document has been preserved.

Related concepts:

- "Annotated XML schema decomposition" on page 235
- "Treatment of CDATA sections in annotated XML schema decomposition" on page 287
- "Effect of validation on XML decomposition results" on page 286
- "XML data encoding" on page 217
- "XML decomposition annotations - Overview" on page 244
- "XML decomposition annotations - Specification and scope" on page 244

Related tasks:

- "Decomposing XML documents with annotated XML schemas" on page 236

Related reference:

- "Annotated XML schema decomposition - Limits and restrictions" on page 317
- "db2-xdb:condition decomposition annotation" on page 266
- "db2-xdb:expression decomposition annotation" on page 263
- "Schema for XML decomposition annotations" on page 322

db2-xdb:normalization decomposition annotation

The db2-xdb:normalization annotation specifies the normalization of whitespace in the XML data to be inserted or to be substituted for \$DECOMP_CONTENT (when used with db2-xdb:expression).

db2-xdb:normalization belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Attribute of <xs:element> or <xs:attribute>, or attribute of <db2-xdb:rowSetMapping>

How to specify:

db2-xdb:normalization is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element db2-xdb:normalization="*value*" />
- <xs:attribute db2-xdb:normalization="*value*" />
- <db2-xdb:rowSetMapping db2-xdb:normalization="*value*">
 <db2-xdb:rowSet>*value*</db2-xdb:rowSet>
 ...
</db2-xdb:rowSetMapping>

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

One of the following case-sensitive tokens:

- canonical
- original (default)
- whitespaceStrip

Note: The db2-xdb:normalization attribute is only valid for mappings between certain XML schema types and SQL character types. Refer to the Details section for the list of supported XML schema types that can be normalized for SQL character columns.

Details:

When inserting XML values into character type target columns (CHAR, VARCHAR, LONG VARCHAR, CLOB, DBCLOB, GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC), it might be necessary to normalize the data being inserted. Different types of normalization can be specified with the db2-xdb:normalization annotation; the valid values, which are case-sensitive settings are:

canonical

The XML value is converted to its canonical form, according to its XML schema type, before being inserted into the target column or substituted for occurrences of \$DECOMP_CONTENT in the same mapping as this db2-xdb:normalization annotation.

original

The original character data, following any processing by an XML parser, of the element content or attribute value (depending on whether this mapping is for an XML element or XML attribute) is inserted into the

target column or substituted for occurrences of \$DECOMP_CONTENT in the same mapping as this db2-xdb:normalization annotation. If the db2-xdb:normalization attribute is not specified for a mapping where this annotation is relevant, then the decomposition process normalizes data according to the "original" setting.

whitespaceStrip

The XML value has all leading and trailing whitespace removed, and consecutive whitespace is collapsed into a single whitespace character, before being inserted into the target column or substituted for occurrences of \$DECOMP_CONTENT in the same mapping as this db2-xdb:normalization annotation.

db2-xdb:normalization is applicable when an element or attribute of (or derived from) one of these atomic XML schema types is mapped to a column of character type (CHAR, VARCHAR, LONG VARCHAR, CLOB, DBCLOB, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC).

- byte, unsigned byte
- integer, positiveInteger, negativeInteger, nonPositiveInteger, nonNegativeInteger
- int, unsignedInt
- long, unsignedLong
- short, unsignedShort
- decimal
- float
- double
- boolean
- time
- date
- dateTime

db2-xdb:normalization will be ignored if specified on any other types. Note that these are XML schema types for which the W3C recommendation, XML Schema Part 2: Datatypes Second Edition, has a canonical representation.

Because the db2-xdb:normalization annotation is only valid for certain XML schema to SQL character type mappings, when the annotation is specified for unsupported mappings, it is ignored.

Example:

The following example shows how whitespace normalization can be controlled with the db2-xdb:normalization annotation. The annotated schema is presented first.

```
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="FIRSTNAME" />
      <xs:element name="lastname" type="xs:string"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="SURNAME"
        db2-xdb:normalization="whitespaceStrip" />
      <xs:element name="activeStatus" type="xs:boolean"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="ACTIVE"
        db2-xdb:normalization="canonical" />
    <xs:attribute name="ID" type="xs:integer"
      db2-xdb:rowSet="AUTHORS" db2-xdb:column="AUTHID"/>
```

```

                                db2-xdb:normalization="whitespaceStrip" />
</xs:sequence>
  </xs:complexType>
</xs:element>

```

The <author> element that is being mapped is presented next (notable whitespaces are represented below by the '_' underscore character for the purpose of demonstration), followed by the resulting AUTHORS table after the decomposition has completed.

```

<author ID=" _22">
  <firstname>Ann</firstname>
  <lastname> _Brown_ </lastname>
  <activeStatus>1</activeStatus>
</author>

```

Table 58. AUTHORS

AUTHID	FIRSTNAME	SURNAME	ACTIVE	NUMBOOKS
22	Ann	_Brown_	true	NULL

The "whitespaceStrip" setting causes the leading whitespace from the "ID" attribute to be removed before the value is inserted into the target table. Notice, however, that the leading and trailing whitespace from the <lastname> element is not stripped, even though the "whitespaceStrip" setting was specified. This is because the <lastname> element has an XML schema type of string, which is not an applicable type for db2-xdb:normalization. The <activeStatus> child element of <author> is defined as a boolean type, and the canonical representation of boolean types is either the literal "true" or "false". The "canonical" setting for the <activeStatus> element results in the canonical form of "1", which is "true", being inserted into the ACTIVE column of the AUTHORS table.

If in the XML schema presented above, the "ID" attribute had been annotated with db2-xdb:normalization="original" instead, then the original value from the document, "_22" (where the underscore character represents whitespace), would have been inserted into the AUTHID column.

Related concepts:

- "XML decomposition annotations - Specification and scope" on page 244
- "Annotated XML schema decomposition" on page 235
- "XML decomposition annotations - Overview" on page 244

Related tasks:

- "Decomposing XML documents with annotated XML schemas" on page 236

Related reference:

- "Schema for XML decomposition annotations" on page 322
- "Annotated XML schema decomposition - Limits and restrictions" on page 317
- "db2-xdb:expression decomposition annotation" on page 263
- "db2-xdb:rowSetMapping decomposition annotation" on page 280
- "Keywords for annotated XML schema decomposition" on page 284

db2-xdb:truncate decomposition annotation

The db2-xdb:truncate annotation specifies whether truncation is permitted when an XML value is inserted into a character target column.

db2-xdb:truncate belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Attribute of <xs:element> or <xs:attribute>, or attribute of <db2-xdb:rowSetMapping>

How to specify:

db2-xdb:truncate is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element db2-xdb:truncate="*value*" />
- <xs:attribute db2-xdb:truncate="*value*" />
- <db2-xdb:rowSetMapping db2-xdb:truncate="*value*">
 <db2-xdb:rowSet>*value*</db2-xdb:rowSet>
 ...
</db2-xdb:rowSetMapping>

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid values:

One of the following tokens:

- 0 (equivalent to false; default)
- 1 (equivalent to true)
- false (case-sensitive; default)
- true (case-sensitive)

Details:

An XML value being inserted into a target character column might be larger than the column size; in this case, the value must be truncated for a successful decomposition. The db2-xdb:truncate attribute indicates whether or not truncation will be permitted when the value is too large for the target column. If this attribute is set to "false" or "0", to indicate that truncation is not permitted, and the XML value being inserted is too large for the target column, an error occurs during decomposition of the XML document and the value is not inserted. The "true" or "1" settings indicate that data truncation is allowed during insertion.

db2-xdb:truncate is only applicable when the target column is either:

- of a character type, or
- of DATE, TIME, or TIMESTAMP type, and the XML value is of type xs:date, xs:time, or xs:dateTime, respectively.

If the `db2-xdb:expression` annotation is specified on the same element or attribute declaration as `db2-xdb:truncate`, then the value of `db2-xdb:truncate` is ignored, as the expression can perform truncation if it is defined as such.

When decomposing into SQL datetime columns, XML values that specify a timezone and have an XML schema type of date, time, or timestamp, `db2-xdb:truncate` must be set to "true" or "1". This is because the structure of SQL datetime types does not provide for timezone specification.

Example:

The following example shows how truncation can be applied to an `<author>` element. A section of the annotated schema is presented first.

```
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"
        db2-xdb:rowSet="AUTHORS" db2-xdb:column="FIRSTNAME"
        db2-xdb:truncate="true" />
      <xs:element name="lastname" type="xs:string" />
      <xs:element name="activeStatus" type="xs:boolean" />
      <xs:element name="activated" type="xs:date"
        db2-xdb:truncate="true" />
      <xs:attribute name="ID" type="xs:integer" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The `<author>` element that is being mapped is presented next.

```
<author ID="0800">
  <firstname>Alexander</firstname>
  <lastname>Smith</lastname>
  <activeStatus>0</activeStatus>
  <activated>2001-10-31Z</activated>
</author>
```

Assume that the `FIRSTNAME` column was defined as a `CHAR` SQL type of size 7, and that the `ACTIVEDATE` column was defined as a `DATE` SQL type. The `AUTHORS` table that results after the decomposition has completed is presented next.

Table 59. AUTHORS

AUTHID	FIRSTNAME	SURNAME	ACTIVE	ACTIVEDATE	NUMBOOKS
NULL	Alexand	NULL	NULL	2001-10-31	NULL

Because the `<firstname>` value "Alexander" is larger than the SQL column size, truncation is necessary in order to insert the value. Notice also that because the `<activated>` element contained a timezone in the XML document, `db2-xdb:truncate` was set to "true" to ensure the date was successfully inserted during decomposition.

Because truncation is required in order to insert the value from the `<firstname>` element or the `<activated>` element, if `db2-xdb:truncate` was not specified, then the default value of `db2-xdb:truncate` is assumed (truncation not permitted), and an error would have been generated to indicate that a row has not been inserted.

Related concepts:

- "Annotated XML schema decomposition" on page 235

- “XML decomposition annotations - Overview” on page 244
- “XML decomposition annotations - Specification and scope” on page 244

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Related reference:

- “Datetime values” in *SQL Reference, Volume 1*
- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “db2-xdb:expression decomposition annotation” on page 263
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “Schema for XML decomposition annotations” on page 322

db2-xdb:rowSetMapping decomposition annotation

The <db2-xdb:rowSetMapping> annotation maps a single XML element or attribute to one or more column and table pairs.

<db2-xdb:rowSetMapping> belongs to the set of decomposition annotations that can be added to an XML schema document to describe the mappings between elements and attributes of XML documents to DB2 base tables. The decomposition process uses the annotated XML schema to determine how elements and attributes of an XML document should be decomposed into DB2 tables.

Annotation type:

Child element of <xs:appinfo> (which is a child element of <xs:annotation>) that is a child element of <xs:element> or <xs:attribute>

How to specify:

db2-xdb:rowSetMapping is specified in any of the following ways (where *value* represents a valid value for the annotation):

- <xs:element>


```

      <xs:annotation>
        <xs:appinfo>
          <db2-xdb:rowSetMapping>
            <db2-xdb:rowSet>value</db2-xdb:rowSet>
            ...
          </db2-xdb:rowSetMapping>
        </xs:appinfo>
      </xs:annotation>
      ...
    </xs:element>
      
```
- <xs:attribute>


```

      <xs:annotation>
        <xs:appinfo>
          <db2-xdb:rowSetMapping>
            <db2-xdb:rowSet>value</db2-xdb:rowSet>
            ...
          </db2-xdb:rowSetMapping>
        </xs:appinfo>
      </xs:annotation>
      ...
    </xs:attribute>
      
```

Namespace:

<http://www.ibm.com/xmlns/prod/db2/xdb1>

Valid structure:

The following are supported attributes of <db2-xdb:rowSetMapping>:

db2-xdb:contentHandling

Enables specification of the type of content that will be decomposed into a table for an element that is of complex type.

db2-xdb:locationPath

Enables mapping of an XML element or attribute declared as part of a reusable group, to different table and column pairs, depending on the ancestry of the element or attribute.

db2-xdb:normalization

Enables specification of the normalization behavior for the content of the XML element or attribute mapped to a character target column, before the content is inserted.

db2-xdb:truncate

Enables specification of whether truncation is permitted when an XML value is inserted into a character target column.

These attributes of <db2-xdb:rowSetMapping> are also available as attributes of XML element or attribute declarations; the same behaviors and requirements apply to these whether they are attributes of <db2-xdb:rowSetMapping> or of <xs:element> or <xs:attribute>. Refer to the individual corresponding documentation of these annotations for details.

The following are supported children elements of <db2-xdb:rowSetMapping>, listed in the order in which they must appear if they are specified:

<db2-xdb:rowSet>

Maps an XML element or attribute to a target base table.

<db2-xdb:column>

(Optional) Maps an XML element or attribute to a base table column. This element is required if db2-xdb:expression is present in the db2-xdb:rowSetMapping annotation.

<db2-xdb:column> can be optional in cases where a value is not intended to be inserted into the table, but is only used for conditional processing. For example, if an element is to be decomposed based on the value of another element, then the other element does not require a column mapping, as its value is not being inserted.

<db2-xdb:expression>

(Optional) Specifies a customized expression, the result of which is inserted into the table named by the db2-xdb:rowSet attribute.

If db2-xdb:expression specifies \$DECOMP_CONTENT and db2-xdb:normalization is specified in the same mapping, then the \$DECOMP_CONTENT value for db2-xdb:expression will be normalized before it is passed to the expression for evaluation, if applicable.

<db2-xdb:condition>

(Optional) Specifies a condition for evaluation.

Note that these children elements of <db2-xdb:rowSetMapping> have the same semantics and syntax as their corresponding attribute annotations, except that quotation marks do not need to be escaped.

For further details, refer to the corresponding documentation of the attribute versions of these annotations.

Details:

<db2-xdb:rowSetMapping> can be used to map an XML element or attribute to a single target table and column, to multiple target columns of the same table, or to multiple tables and columns. There are two equivalent methods for mapping to a single table and column: the combination of the db2-xdb:rowSet and db2-xdb:column annotations (which are attributes of the element or attribute being mapped), or specifying <db2-xdb:rowSetMapping> (which is a child element of the element or attribute being mapped). Both methods yield the same results and differ only in their notation.

All whitespace in the character content of the child elements of <db2-xdb:rowSetMapping> is significant; no whitespace normalization is performed. For delimited SQL identifiers specified in the children elements, the quotation marks delimiter must be included in the character content and not escaped. The '&' and '<' characters used in SQL identifiers, however, must be escaped.

Example:

The following example shows how a single attribute, named "isbn", can be mapped to more than one table with the <db2-xdb:rowSetMapping> annotation. A section of the annotated schema is presented first. It shows how the isbn value is mapped to both the BOOKS and BOOKCONTENTS tables.

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="authorID" type="xs:integer"/>
      <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="isbn" type="xs:string">
      <xs:annotation>
        <xs:appinfo>
          <db2-xdb:rowSetMapping>
            <db2-xdb:rowSet>BOOKS</db2-xdb:rowSet>
            <db2-xdb:column>ISBN</db2-xdb:column>
          </db2-xdb:rowSetMapping>
          <db2-xdb:rowSetMapping>
            <db2-xdb:rowSet>BOOKCONTENTS</db2-xdb:rowSet>
            <db2-xdb:column>ISBN</db2-xdb:column>
          </db2-xdb:rowSetMapping>
        </xs:appinfo>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute name="title" type="xs:string" />
  </xs:complexType>
</xs:element>
```

The <book> element that is being mapped is presented next, followed by the resulting BOOKS and BOOKCONTENTS tables after the decomposition has completed.

```

<book isbn="1-11-111111-1" title="My First XML Book">
  <authorID>22</authorID>
  <!-- this book does not have a preface -->
  <chapter number="1" title="Introduction to XML">
    <paragraph>XML is fun...</paragraph>
    ...
  </chapter>
  ...
</book>

```

Table 60. BOOKS

ISBN	TITLE	CONTENT
1-11-111111-1	NULL	NULL

Table 61. BOOKCONTENTS

ISBN	CHPTNUM	CHPTTITLE	CHPTCONTENT
1-11-111111-1	NULL	NULL	NULL

Alternative mapping using combination of <db2-xdb:rowSetMapping> and db2-xdb:rowSet and db2-xdb:column:

The following section of an annotated schema is equivalent to the XML schema fragment presented above, as it yields the same decomposition results. The difference between the two schemas is that this schema replaces one mapping with the db2-xdb:rowSet and db2-xdb:column combination, instead of using only the <db2-xdb:rowSetMapping> annotation.

```

<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="authorID" type="xs:integer"/>
      <xs:element name="chapter" type="chapterType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="isbn" type="xs:string"
      db2-xdb:rowSet="BOOKS" db2-xdb:column="ISBN" >
      <xs:annotation>
        <xs:appinfo>
          <db2-xdb:rowSetMapping>
            <db2-xdb:rowSet>BOOKCONTENTS</db2-xdb:rowSet>
            <db2-xdb:column>ISBN</db2-xdb:column>
          </db2-xdb:rowSetMapping>
        </xs:appinfo>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute name="title" type="xs:string" />
  </xs:complexType>
</xs:element>

```

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244
- “XML decomposition annotations - Specification and scope” on page 244

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Related reference:

- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “db2-xdb:column decomposition annotation” on page 257
- “db2-xdb:condition decomposition annotation” on page 266
- “db2-xdb:contentHandling decomposition annotation” on page 269
- “db2-xdb:expression decomposition annotation” on page 263
- “db2-xdb:locationPath decomposition annotation” on page 259
- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:truncate decomposition annotation” on page 278
- “Schema for XML decomposition annotations” on page 322
- “db2-xdb:normalization decomposition annotation” on page 274

Keywords for annotated XML schema decomposition

Annotated XML schema decomposition offers decomposition keywords for use in the db2-xdb:condition and db2-xdb:expression annotations.

\$DECOMP_CONTENT

The value of the mapped XML element or attribute from the document, constructed according to the setting of the db2-xdb:contentHandling annotation. The value replaced for \$DECOMP_CONTENT in the expression should always be considered a character type. Refer to the limits and restrictions documentation for the maximum string length of and maximum number of \$DECOMP_CONTENT instances supported. If db2-xdb:expression specifies \$DECOMP_CONTENT and db2-xdb:normalization is specified in the same mapping, then the \$DECOMP_CONTENT value for db2-xdb:expression will be normalized before it is passed to the expression for evaluation, if applicable.

\$DECOMP_CONTENT can be used to process the value of the mapped element or attribute, using customized expressions, rather than directly inserting that value.

\$DECOMP_DOCUMENTID

The string value specified in the *documentid* input parameter of the xdbDecompXML stored procedure, which identifies the XML document being decomposed. When the document is decomposed, the input value provided to the xdbDecompXML stored procedure is used as the value substituted for \$DECOMP_DOCUMENTID.

Applications can pass into xdbDecompXML, uniquely generated document IDs. These IDs can then be directly inserted into the database. The IDs can also be passed into expressions that generate unique identifiers for elements or attributes. \$DECOMP_DOCUMENTID can be used, therefore, to insert unique identifiers that are not present in the XML document.

\$DECOMP_ELEMENTID

A system-generated integer identifier that uniquely identifies within the XML document, the element or attribute this annotation describes. This value will remain unchanged between decomposition operations for the same XML document, as long as the document does not change in any of the following ways: element addition, element deletion, or a change in an element’s position in document order. If the document is modified in these ways and decomposed again, elements might not have the same identifier as they did following the previous decomposition.

\$DECOMP_ELEMENTID specified on an attribute is defined to be the value of the \$DECOMP_ELEMENTID for the element to which this attribute belongs.

The value generated by \$DECOMP_ELEMENTID can also be used to indicate the order of elements in the original document. This can be useful in cases where the XML document needs to be re-composed from relational tables.

Related concepts:

- “Annotated XML schema decomposition” on page 235

Related reference:

- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “db2-xdb:condition decomposition annotation” on page 266
- “db2-xdb:contentHandling decomposition annotation” on page 269
- “db2-xdb:expression decomposition annotation” on page 263
- “db2-xdb:normalization decomposition annotation” on page 274
- “xdbDecompXML stored procedures for annotated schema decomposition” on page 240

Decomposition results

How decomposition results are formed in annotated XML schema decomposition

While typical decomposition processes decompose only XML element or attribute content, annotated XML schema decomposition supports the insertion of values that do not necessarily exist in the XML document.

Decomposed content can be any of the following:

- the value of an attribute in the XML document
- the value of an element in the XML document, where the exact content depends on the setting of the <db2-xdb:contentHandling> annotation:
 - text - character data from this element only (not its descendants)
 - stringValue - character data from this element and its descendants
 - serializedSubtree - markup of all content between this element’s start and end tags

Refer to the <db2-xdb:contentHandling> documentation for more information.

- a value based on the content of a mapped attribute or element in the XML document
- a generated value that is independent of any values in the XML document

The latter two values are possible through the db2-xdb:expression annotation. This annotation allows you to specify an expression, the result of which is inserted during decomposition.

The value from an XML document can be applied to an expression to generate a result, thereby transforming the data before it is inserted into the target column. An expression can also generate a value that is based on the mapped element or attribute (such as the element’s name). db2-xdb:expression also allows constants to

be specified, where the constant might or might not be related to the mapped value from the XML document. `db2-xdb:expression` enables you to combine any of these techniques to generate a value for insertion.

Note that the expression is invoked as many times as the element or attribute with which it is associated, is encountered in the XML document.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Examples of mappings in annotated XML schema decomposition” on page 296
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “`db2-xdb:contentHandling` decomposition annotation” on page 269
- “`db2-xdb:expression` decomposition annotation” on page 263
- “XML decomposition annotations - Summary” on page 246

Effect of validation on XML decomposition results

Annotated XML schema decomposition does not require that the input document be validated during decomposition, but validation is recommended, whether it is performed before decomposition or during decomposition. You can perform validation before decomposition (using the `XMLVALIDATE SQL/XML` function), or during decomposition as part of the call to the `xdbDecompXML` stored procedure or the `DECOMPOSE XML DOCUMENT` command. Validating the XML documents being decomposed ensures that:

- values are decomposed into tables only when the entire document is valid according to the specified XML schema (this ensures that only valid values are stored in the database)
- default values defined for an element or attribute are inserted into the database (when validation is performed using one of the `xdbDecompXML` decomposition stored procedures and when the element or attribute does not appear in the XML document)
- all entities in the XML document will be resolved, for validation performed during decomposition (if an entity in the XML document has not been registered prior to decomposition, an error will be returned)
- non-default whitespace normalization occurs, as specified in the schema (when validation is performed using one of the `xdbDecompXML` decomposition stored procedures)

Validating the input documents against the registered XML schema is recommended, because the decomposition process assumes that input documents are valid according to the corresponding annotated schema. If the validation is not performed and the input documents are invalid, decomposition can insert different rows for the same input document (compared to when validation is performed, for reasons such as entity resolution or default attribute additions), or the decomposition might yield unexpected results. The results of and the side effects on existing data from decomposing an invalid document are undefined.

Note that errors in the schema, such as non-deterministic content models, or incorrect type derivations can cause the decomposition process to fail when validation is performed during decomposition. Verify that your annotated schema is correct and re-register the schema before attempting the decomposition again.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Troubleshooting considerations for annotated XML schema decomposition” on page 319

Related tasks:

- “Registering and enabling XML schemas for decomposition” on page 237

Related reference:

- “DECOMPOSE XML DOCUMENT ” on page 242
- “XMLVALIDATE scalar function” in *SQL Reference, Volume 1*
- “xdbDecompXML stored procedures for annotated schema decomposition” on page 240

Treatment of CDATA sections in annotated XML schema decomposition

The contents of CDATA sections are inserted into the database for elements that are annotated for decomposition. The CDATA section delimiters (“<![CDATA[” and “]]>”) are not inserted. The CDATA contents are subject to line-ending normalization by the XML parser.

If the XML element declaration in the XML schema is annotated with the attribute `db2-xdb:contentHandling="serializeSubtree"`, however, then the CDATA section, including the CDATA delimiters, are inserted.

Related concepts:

- “Annotated XML schema decomposition” on page 235

Related reference:

- “db2-xdb:contentHandling decomposition annotation” on page 269

NULL values and empty strings in annotated XML schema decomposition

Annotated XML schema decomposition inserts NULL values or empty strings under certain conditions.

XML elements:

The following table shows when an empty string or a NULL value is inserted into the database for elements in the XML document.

Table 62. NULL handling for mapped elements

Condition	Empty string	NULL value
Element missing from document		↙
Element satisfies all of the following conditions: <ul style="list-style-type: none">• is present in the document• contains the <code>xsi:nil="true"</code> or <code>xsi:nil="1"</code> attribute in the start tag		↙

Table 62. NULL handling for mapped elements (continued)

Condition	Empty string	NULL value
<p>Element satisfies all of the following conditions:</p> <ul style="list-style-type: none"> • is present and empty in the document • does not contain the <code>xsi:nil="true"</code> or <code>xsi:nil="1"</code> attribute in the start tag • is derived from or declared to be of list type, union type, complex type with mixed content, or the following atomic built-in types: <code>xsd:string</code>, <code>xsd:normalizedString</code>, <code>xsd:token</code>, <code>xsd:hexBinary</code>, <code>xsd:base64Binary</code>, <code>xsd:anyURI</code>, <code>xsd:anySimpleType</code>; any other types will result in an error. 	↙	
<p>Notes:</p> <ol style="list-style-type: none"> 1. If a mapping involves the <code>db2-xdb:condition</code> or <code>db2-xdb:expression</code> annotations, then the empty string or NULL value (as shown in this table) is passed as the argument for expression evaluation. 2. If a target column is of type CHAR or GRAPHIC, an empty string is inserted as a string of blank characters. 		

XML attributes:

The following table shows when an empty string or a NULL value is inserted into the database when XML attributes annotated for decomposition contain NULL values in the document or are missing.

Table 63. NULL handling for mapped attributes

Condition	Empty string	NULL value
Attribute missing from document (either because no validation was performed, or there was no default value provided by validation)		↙
<p>Attribute satisfies all of the following conditions:</p> <ul style="list-style-type: none"> • is present and empty in the document • is derived from or declared to be of list type, union type, or the following atomic built-in types: <code>xsd:string</code>, <code>xsd:normalizedString</code>, <code>xsd:token</code>, <code>xsd:hexBinary</code>, <code>xsd:base64Binary</code>, <code>xsd:anyURI</code>, <code>xsd:anySimpleType</code>; any other types will result in an error. 	↙	
<p>Note: If a mapping involves the <code>db2-xdb:condition</code> or <code>db2-xdb:expression</code> annotations, then the empty string or NULL value (as shown in this table) is passed as the argument for expression evaluation.</p>		

Related concepts:

- “Annotated XML schema decomposition” on page 235

Related reference:

- “`db2-xdb:condition` decomposition annotation” on page 266
- “`db2-xdb:expression` decomposition annotation” on page 263

Recommendations

Annotations of derived complex types for annotated XML schema decomposition

When annotating complex types derived by restriction or extension for decomposition, you need to apply additional mappings.

Derived by restriction:

Complex types that are derived by restriction require that the common elements and attributes from the base type be repeated in the definition of the derived type. Decomposition annotations that are present in the base type, therefore, must also be included in the derived type.

Derived by extension:

In the definition of complex types derived by extension, only the elements and attributes that are in addition to the base type are specified. If the decomposition mappings for the derived type differ from the mappings of the base type, then decomposition annotations must be added to the base type to clearly differentiate the mappings of the base from the derived types.

The following example shows how a type derived by extension, `outOfPrintBookType`, can be mapped to a different table than its base type, `bookType`. Notice how the `db2-xdb:locationPath` annotation is specified in the `bookType` base type to clearly differentiate which mappings apply to the base type, and which apply to the derived type. The `<lastPublished>` and `<publisher>` elements of the derived type `outOfPrintType` do not require the `db2-xdb:locationPath` annotation in this example, as these elements are only involved in a single mapping.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2-xdb1">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:table>
        <db2-xdb:name>BOOKS</db2-xdb:name>
        <db2-xdb:rowSet>inPrintRowSet</db2-xdb:rowSet>
      </db2-xdb:table>
      <db2-xdb:table>
        <db2-xdb:name>OUTOFPRINT</db2-xdb:name>
        <db2-xdb:rowSet>outOfPrintRowSet</db2-xdb:rowSet>
      </db2-xdb:table>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="books">
    <xs:complexType>
      <xs:choice>
        <xs:element name="book" type="bookType"
          minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="outOfPrintBook" type="outOfPrintBookType"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="bookType">
    <xs:sequence>
      <xs:element name="authorID" type="xs:integer"/>
      <xs:element name="chapter" type="chapterType" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

```

</xs:sequence>
<xs:attribute name="title" type="xs:string"
    db2-xdb:locationPath="/books/book/@title"
    db2-xdb:rowSet="inPrintRowSet" db2-xdb:column="TITLE">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetMapping db2-xdb:locationPath="/books/outOfPrintBook/@title">
        <db2-xdb:rowSet>outOfPrintRowSet</db2-xdb:rowSet>
        <db2-xdb:column>TITLE</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="isbn" type="xs:string"
    db2-xdb:locationPath="/books/book/@isbn"
    db2-xdb:rowSet="inPrintRowSet" db2-xdb:column="ISBN">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetMapping db2-xdb:locationPath="/books/outOfPrintBook/@isbn">
        <db2-xdb:rowSet>outOfPrintRowSet</db2-xdb:rowSet>
        <db2-xdb:column>ISBN</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</xs:attribute>
</xs:complexType>
<xs:complexType name="outOfPrintBookType">
  <xs:complexContent>
    <xs:extension base="bookType">
      <xs:sequence>
        <xs:element name="lastPublished" type="xs:date"
            db2-xdb:rowSet="outOfPrintRowSet" db2-xdb:column="LASTPUBDATE"/>
        <xs:element name="publisher" type="xs:string"
            db2-xdb:rowSet="outOfPrintRowSet" db2-xdb:column="PUBLISHER"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:simpleType name="paragraphType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:complexType name="chapterType">
  <xs:sequence>
    <xs:element name="paragraph" type="paragraphType" maxOccurs="unbounded"
        db2-xdb:locationPath="/books/book/chapter/paragraph"
        db2-xdb:rowSet="inPrintRowSet" db2-xdb:column="CONTENT">
      <xs:annotation>
        <xs:appinfo>
          <db2-xdb:rowSetMapping
            db2-xdb:locationPath="/books/outOfPrintBook/chapter/paragraph">
            <db2-xdb:rowSet>outOfPrintBook</db2-xdb:rowSet>
            <db2-xdb:column>CONTENT</db2-xdb:column>
          </db2-xdb:rowSetMapping>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="number" type="xs:integer"/>
  <xs:attribute name="title" type="xs:string"/>
</xs:complexType>
</xs:schema>

```

The annotations indicate that values from the <book> element will be decomposed into the BOOKS table, while values from the <outOfPrintBook> element will be decomposed into the OUTOFPRINT table.

Consider the following element from an XML document:

```
<books>
  <book isbn="1-11-111111-1" title="My First XML Book">
    <authorID>22</authorID>
    <chapter number="1" title="Introduction to XML">
      <paragraph>XML is fun...</paragraph>
    </chapter>
    <chapter number="2" title="XML and Databases">
      <paragraph>XML can be used with...</paragraph>
    </chapter>
  </book>
  <outOfPrintBook isbn="7-77-777777-7" title="Early XML Book">
    <authorID>41</authorID>
    <chapter number="1" title="Introductory XML">
      <paragraph>Early XML...</paragraph>
    </chapter>
    <chapter number="2" title="What is XML">
      <paragraph>XML is an emerging technology...</paragraph>
    </chapter>
    <lastPublished>2000-01-31</lastPublished>
    <publisher>Early Publishers Group</publisher>
  </outOfPrintBook>
</books>
```

The following tables result from decomposing the document that this element belongs to, using the preceding annotated schema:

Table 64. BOOKS

ISBN	TITLE	CONTENT
1-11-111111-1	My First XML Book	XML is fun...
1-11-111111-1	My First XML Book	XML can be used with...

Table 65. OUTOFPRINT

ISBN	TITLE	CONTENT	LASTPUBDATE	PUBLISHER
7-77-777777-7	Early XML Book	Early XML...	2000-01-31	Early Publishers Group
7-77-777777-7	Early XML Book	XML is an emerging technology...	2000-01-31	Early Publishers Group

Related concepts:

- “Annotated XML schema decomposition” on page 235

Related reference:

- “db2-xdb:column decomposition annotation” on page 257
- “db2-xdb:locationPath decomposition annotation” on page 259
- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “db2-xdb:table decomposition annotation” on page 254

Checklist for annotated XML schema decomposition

Annotated XML schema decomposition can be a complex task because it requires you to map possibly multiple XML elements and attributes to multiple columns

and tables in the database. This mapping can also involve transforming the XML data before inserting it, or applying conditions for insertion.

The following are items to consider when annotating your XML schema, along with pointers to related documentation:

- Understand what decomposition annotations are available to you.
- Ensure, during mapping, that the type of the column is compatible with the XML schema type of the element or attribute it is being mapped to.
- Structure your XML schema to minimize demands made on system memory resources.
- Ensure complex types derived by restriction or extension are properly annotated.
- Confirm that no decomposition limits and restrictions are violated.
- Ensure that the tables and columns referenced in the annotation exist at the time the schema is registered with the XSR.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Annotations of derived complex types for annotated XML schema decomposition” on page 289
- “Troubleshooting considerations for annotated XML schema decomposition” on page 319
- “XML schema structuring recommendations for decomposition” on page 292

Related tasks:

- “Registering and enabling XML schemas for decomposition” on page 237

Related reference:

- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “XML schema to SQL types compatibility for annotated schema decomposition” on page 307
- “XML decomposition annotations - Summary” on page 246

XML schema structuring recommendations for decomposition

You can minimize the demands made on your system’s memory resources from annotated schema decomposition, by adjusting the order of elements in your annotated XML schema. For very large documents, following this recommendation might make a difference in whether the document can be decomposed without having to increase the amount of available memory for the DB2 database system. For sibling elements that are annotated for decomposition, elements of simple types should be placed before the sibling elements of complex type in the annotated schema. Similarly, sibling elements that have the maxOccurs attribute set to 1 should be placed before siblings that have maxOccurs > 1.

The memory consumption required by annotated schema decomposition is affected by the structure of the XML schema because each item that forms a row must be held in memory until all of the items that form the row are processed. These schema structuring recommendations organize the items of a row in such a way as to minimize the number of items that must be kept in memory.

The following example shows the recommended XML schema structuring for mapped sibling elements contrasted with the less optimal structuring. Notice how

<complexElem>, which is of complex type, is placed before <status>, which is of simple type, in the less optimal example. Placing <complexElem> after the <id> and <status> elements improves decomposition runtime efficiency.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2-xdb1">
  <!-- Recommended structuring with simple types placed before
         the recurring element <wrapper>, which is of complex type -->
  <xs:complexType name="typeA">
    <xs:sequence>
      <xs:element name="id" type="xs:integer"
                 db2-xdb:rowSet="relA" db2-xdb:column="ID" />
      <xs:element name="status" type="xs:string"
                 db2-xdb:rowSet="relA" db2-xdb:column="status" />
      <xs:element name="wrapper" type="typeX" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <!-- Less optimal structuring with recurring complex type element
         appearing before the simple type element -->
  <!--
  <xs:complexType name="typeA">
    <xs:sequence>
      <xs:element name="id" type="xs:integer"
                 db2-xdb:rowSet="relA" db2-xdb:column="ID" />
      <xs:element name="wrapper" type="typeX" maxOccurs="unbounded"/>
      <xs:element name="status" type="xs:string"
                 db2-xdb:rowSet="relA" db2-xdb:column="status" />
    </xs:sequence>
  </xs:complexType> -->

  <xs:complexType name="typeX">
    <xs:sequence>
      <xs:element name="elem1" type="xs:string"
                 db2-xdb:rowSet="relA" db2-xdb:column="elem1" />
      <xs:element name="elem2" type="xs:long"
                 db2-xdb:rowSet="relA" db2-xdb:column="elem2" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="A" type="typeA" />
</xs:schema>
```

Note that <id>, <status>, <elem1>, and <elem2> are mapped to the same rowSet, that is, together they form a row. Memory associated with a row is released when a row is complete. In the less optimal case presented above, none of the rows associated with the rowSet relA can be considered complete until the <status> element is reached in the document. The <wrapper> element must be processed first, however, as it occurs before the <status> element. This means that all instances of <wrapper> must be buffered in memory until the <status> element is reached (or the end of <A> is reached, if <status> is absent from the document).

The impact of this structure becomes significant if there are a high number of instances of an element. For example, if there were 10 000 instances of the <wrapper> element, then all 10 000 instances would have to be held in memory until the rowSet was complete. In the optimal case presented above, however, memory associated with the rows of rowset relA, can be released when <elem2> is reached.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Checklist for annotated XML schema decomposition” on page 291

Related reference:

- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:column decomposition annotation” on page 257

Mapping examples

rowSets in annotated XML schema decomposition

db2-xdb:rowSet identifies the target table into which a value is decomposed. This annotation can be set either to a table name or a rowSet name.

A rowSet is specified with the db2-xdb:rowSet annotation, which is added to the XML schema document as either an attribute of an element or attribute declaration, or a child of the <db2-xdb:rowSetMapping> annotation.

Every mapping, across all schema documents that form the XML schema, which has the same db2-xdb:rowSet value for an instance of an element or attribute, defines a row.

For example, consider the following XML document:

```
<publications>
  <textbook title="Programming with XML">
    <isbn>0-11-011111-0</isbn>
    <author>Mary Brown</author>
    <author>Alex Page</author>
    <publicationDate>2002</publicationDate>
    <university>University of London</university>
  </textbook>
  <childrensbook title="Children's Fables">
    <isbn>5-55-555555-5</isbn>
    <author>Bob Carter</author>
    <author>Melaine Snowe</author>
    <publicationDate>1999</publicationDate>
  </childrensbook>
</publications>
```

To decompose this document such that each book’s isbn and title (whether it is a textbook or children’s book) is inserted into the same table (named ALLPUBLICATIONS), multiple rowSets must be defined: a rowSet to group values related to textbooks, and another to group values related to children’s books.

rowSets, in this case, ensure that only values that are semantically related are grouped together to form a row. That is, the use of rowSets will group the isbn value for a textbook with its title, and the isbn value for a children’s book with its title. This ensures that a row does not contain the isbn value from a textbook, while having the title from a children’s book.

Without rowSets, it is impossible to determine which values should be grouped together to form a row that is still semantically correct.

The application of rowSets in an XML schema document is presented next. The two rowSets, textbk_rowSet and childrens_rowSet, are specified on the isbn element declaration of the <textbook> and <childrensbook> elements respectively. These rowsets are then associated with the ALLPUBLICATIONS table through the <db2-xdb:table> annotation.

Note that using the rowSet annotation not as a table identifier, but as a rowSet identifier allows you to easily change table names referenced in the XML schema. This is because, when the value of db2-xdb:rowSet represents an identifier rather than a table name, you need to use the <db2-xdb:table><db2-xdb:name></db2-xdb:table></db2-xdb:table> annotation to actually specify the table name. With this method, you only need to update the table name in one place, if required.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2/xdb1"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:defaultSQLSchema>admin</db2-xdb:defaultSQLSchema>
      <db2-xdb:table>
        <db2-xdb:name>ALLPUBLICATIONS</db2-xdb:name>
        <db2-xdb:rowSet>textbk_rowSet</db2-xdb:rowSet>
        <db2-xdb:rowSet>childrens_rowSet</db2-xdb:rowSet>
      </db2-xdb:table>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="publications">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="textbook" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="isbn" type="xs:string"
                db2-xdb:rowSet="textbk_rowSet" db2-xdb:column="PUBS_ISBN"/>
              <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
              <xs:element name="publicationDate" type="xs:gYear"/>
              <xs:element name="university" type="xs:string"
                maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="title" type="xs:string" use="required"
              db2-xdb:rowSet="textbk_rowSet" db2-xdb:column="PUBS_TITLE"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="childrensbook" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="isbn" type="xs:string"
                db2-xdb:rowSet="childrens_rowSet" db2-xdb:column="PUBS_ISBN"/>
              <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
              <xs:element name="publicationDate" type="xs:gYear"/>
            </xs:sequence>
            <xs:attribute name="title" type="xs:string" use="required"
              db2-xdb:rowSet="childrens_rowSet" db2-xdb:column="PUBS_TITLE"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The table that results from decomposing using this annotated XML schema is shown next.

Table 66. ALLPUBLICATIONS

ISBN	PUBS_TITLE
0-11-011111-0	Programming with XML
5-55-555555-5	Children's Fables

While the example presented above shows a simple case of decomposing using rowSets, rowSets can be used in more complex mappings to group together multiple items from different parts of an XML schema to form rows on the same table and column pair.

Conditional transformations:

rowSets allow you to apply different transformations to the values being decomposed, depending on the values themselves.

For example, consider the following two instances of an element named "temperature":

```
<temperature unit="Celsius">49</temperature>  
<temperature unit="Fahrenheit">49</temperature>
```

If the values of these elements are to be inserted into the same table, and you want the table to contain consistent values (all Fahrenheit values, for example), then you need to convert the values that have the attribute unit="Celsius" to Fahrenheit before inserting. You can do this by mapping all elements with the attribute unit="Celsius" to one rowSet and all elements with the attribute unit="Fahrenheit" to another rowSet. The rowSet for Celsius values can then have a conversion formula applied before insertion.

Related concepts:

- "Annotated XML schema decomposition" on page 235
- "Examples of mappings in annotated XML schema decomposition" on page 296

Related reference:

- "db2-xdb:column decomposition annotation" on page 257
- "db2-xdb:defaultSQLSchema decomposition annotation" on page 247
- "db2-xdb:rowSet decomposition annotation" on page 249
- "db2-xdb:rowSetMapping decomposition annotation" on page 280
- "db2-xdb:table decomposition annotation" on page 254

Examples of mappings in annotated XML schema decomposition

Annotated XML schema decomposition relies on mappings to determine how to decompose an XML document into tables. Mappings are expressed as annotations added to the XML schema document. These mappings describe how you want an XML document to be decomposed into tables.

The following examples show some common mapping scenarios:

- Mapping to an XML column
- A value mapped to a single table that yields a single row
- A value mapped to a single table that yields multiple rows
- A value mapped to multiple tables
- Grouping multiple values mapped to a single table
- Multiple values from different contexts mapped to a single table

Related concepts:

- "Annotated XML schema decomposition" on page 235

- “XML decomposition annotations - Overview” on page 244

Decomposition annotation example: Mapping to an XML column

In annotated XML schema decomposition, you can map an XML fragment to a column defined using the XML data type.

Consider the following XML document:

```
<publications>
  <textbook title="Programming with XML">
    <isbn>0-11-011111-0</isbn>
    <author>Mary Brown</author>
    <author>Alex Page</author>
    <publicationDate>2002</publicationDate>
    <university>University of London</university>
  </textbook>
</publications>
```

If you wanted to store the `<textbook>` XML element and book title as follows, you would add annotations to the declarations of the `<textbook>` element and title attribute in the corresponding XML schema document. The annotations should specify the DETAILS and TITLE columns, where the DETAILS column has been defined with the XML type, as well as the TEXTBOOKS table.

Table 67. TEXTBOOKS

TITLE	DETAILS
Programming with XML	<pre><textbook title="Programming with XML"> <isbn>0-11-011111-0</isbn> <author>Mary Brown</author> <author>Alex Page</author> <publicationDate>2002</publicationDate> <university>University of London</university> </textbook></pre>

Depending on the annotation, an annotation can be specified in the schema document as an attribute or an element. Some annotations can be specified as either. Refer to the documentation for each specific annotation to determine how a particular annotation can be specified.

Specify the target table and column using either `db2-xdb:rowSet` and `db2-xdb:column` as attributes of `<xs:element>` or `<xs:attribute>` or the `<db2-xdb:rowSet>` and `<db2-xdb:column>` children elements of `<db2-xdb:rowSetMapping>`. Specifying these mappings as elements or attributes are equivalent.

The following fragment of the XML schema document shows how two mappings are added to the `<textbook>` element and title attribute by specifying annotations as attributes.

```
<xs:element name="publications">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="textbook" maxOccurs="unbounded"
        db2-xdb:rowSet="TEXTBOOKS" db2-xdb:column="DETAILS">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="isbn" type="xs:string"/>
          <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
```

```

        <xs:element name="publicationDate" type="xs:gYear"/>
        <xs:element name="university" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="title" type="xs:string" use="required"
        db2-xdb:rowSet="TEXTBOOKS" db2-xdb:column="TITLE"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>

```

The `db2-xdb:rowSet` annotations specify the name of the target table, and the `db2-xdb:column` annotations specify the name of the target column. Because the `<textbook>` element is of complex type and contains complex content, and the `db2-xdb:contentHandling` annotation was not specified, by default, all markup within the element (including its start and end tags) is inserted into the XML column according to the `serializeSubtree` setting of `db2-xdb:contentHandling`. Whitespace within the XML document is preserved. Refer to the `db2-xdb:contentHandling` documentation for more detail.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Examples of mappings in annotated XML schema decomposition” on page 296
- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “`db2-xdb:column` decomposition annotation” on page 257
- “`db2-xdb:contentHandling` decomposition annotation” on page 269
- “`db2-xdb:rowSet` decomposition annotation” on page 249
- “`db2-xdb:rowSetMapping` decomposition annotation” on page 280
- “XML decomposition annotations - Summary” on page 246

Decomposition annotation example: A value mapped to a single table that yields a single row

Mapping a value from an XML document to a single table and column pair is a simple form of mapping in annotated XML schema decomposition. The result of this mapping depends on the relationship between items mapped to the same `rowSet`. If the values that are mapped together in a single `rowSet` have a one to one relationship, as determined by the value of the `maxOccurs` attribute of the element or the containing model group declaration, a single row will be formed for each instance of the mapped item in the XML document. If the values in a single `rowSet` have a one to many relationship, where one value appears only once in the document for multiple instances of another item, as indicated by the value of the `maxOccurs` attribute, then multiple rows will result when the XML document is decomposed.

This example shows the simpler case of a one to one relationship between values in a `rowSet`.

Consider the following XML document:

```

<publications>
  <textbook title="Programming with XML">
    <isbn>0-11-011111-0</isbn>
  </textbook>
</publications>

```

```

    <author>Mary Brown</author>
    <author>Alex Page</author>
    <publicationDate>2002</publicationDate>
    <university>University of London</university>
  </textbook>
</publications>

```

If you wanted the values of the `<isbn>` and `<publicationDate>` elements, as well as the title attribute, to be decomposed into the TEXTBOOKS table as follows, you need to add annotations to the declarations for these elements and attributes in the corresponding XML schema document. The annotations would specify the table and column names that each item is mapped to.

Table 68. TEXTBOOKS

ISBN	TITLE	DATE
0-11-011111-0	Programming with XML	2002

Depending on the annotation, an annotation can be specified in the schema document as an attribute or an element. Some annotations can be specified as either. Refer to the documentation for each specific annotation to determine how a particular annotation can be specified.

For the case of mapping a value to single table and column pair, you need to specify the table and column on the value being mapped. This is done using either `db2-xdb:rowSet` and `db2-xdb:column` as attributes of `<xs:element>` or `<xs:attribute>` or the `<db2-xdb:rowSet>` and `<db2-xdb:column>` children elements of `<db2-xdb:rowSetMapping>`. Specifying these mappings as elements or attributes are equivalent.

The following example shows how to map elements and attributes from the `<textbook>` element to the TEXTBOOKS table by specifying annotations as attributes.

```

<xs:element name="publications">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="textbook" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="isbn" type="xs:string"
              db2-xdb:rowSet="TEXTBOOKS" db2-xdb:column="ISBN"/>
            <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
            <xs:element name="publicationDate" type="xs:gYear"
              db2-xdb:rowSet="TEXTBOOKS" db2-xdb:column="DATE"/>
            <xs:element name="university" type="xs:string" maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="title" type="xs:string" use="required"
            db2-xdb:rowSet="TEXTBOOKS" db2-xdb:column="TITLE"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

The `maxOccurs` XML Schema attribute has a default value of 1, and so each of the items mapped to the TEXTBOOKS rowSet has a one to one relationship with each other. Because of this one to one relationship, a single row is formed for each instance of the `<textbook>` element.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Examples of mappings in annotated XML schema decomposition” on page 296
- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “db2-xdb:column decomposition annotation” on page 257
- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “XML decomposition annotations - Summary” on page 246

Decomposition annotation example: A value mapped to a single table that yields multiple rows

Mapping a value from an XML document to a single table and column pair is a simple form of mapping in annotated XML schema decomposition. The result of this mapping depends on the relationship between items mapped to the same rowSet. If the values that are mapped together in a single rowSet have a one to one relationship, as determined by the value of the maxOccurs attribute of the element or the containing model group declaration, a single row will be formed for each instance of the mapped item in the XML document. If the values in a single rowSet have a one to many relationship, where one value appears only once in the document for multiple instances of another item, as indicated by the value of the maxOccurs attribute, then multiple rows will result when the XML document is decomposed.

This example shows the more complex case of a one to many relationship between values in a rowSet.

Consider the following XML document:

```
<textbook title="Programming with XML">
  <isbn>0-11-011111-0</isbn>
  <author>Mary Brown</author>
  <author>Alex Page</author>
  <publicationDate>2002</publicationDate>
  <university>University of London</university>
</textbook>
```

If you wanted to store the ISBN and authors for a textbook as follows, you would add annotations to the declarations of the <isbn> and <author> elements in the corresponding XML schema document. The annotations should specify the ISBN and AUTHNAME columns, as well as the TEXTBOOK_AUTH table.

Table 69. TEXTBOOKS_AUTH

ISBN	AUTHNAME
0-11-011111-0	Mary Brown
0-11-011111-0	Alex Page

Depending on the annotation, an annotation can be specified in the schema document as an attribute or an element. Some annotations can be specified as either. Refer to the documentation for each specific annotation to determine how a particular annotation can be specified.

For the case of mapping a value to single table and column pair, you need to specify the table and column on the value being mapped. This is done using either `db2-xdb:rowSet` and `db2-xdb:column` as attributes of `<xs:element>` or `<xs:attribute>`, or the `<db2-xdb:rowSet>` and `<db2-xdb:column>` children elements of `<db2-xdb:rowSetMapping>`.

Specifying these mappings as elements or attributes is equivalent. The mappings are specified as elements in the XML schema document presented next.

```
<xs:element name="textbook" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="isbn" type="xs:string">
        <xs:annotation>
          <xs:appinfo>
            <db2-xdb:rowSetMapping>
              <db2-xdb:rowSet>TEXTBOOKS_AUTH</db2-xdb:rowSet>
              <db2-xdb:column>ISBN</db2-xdb:column>
            </db2-xdb:rowSetMapping>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      <xs:element name="author" type="xs:string" maxOccurs="unbounded">
        <xs:annotation>
          <xs:appinfo>
            <db2-xdb:rowSetMapping>
              <db2-xdb:rowSet>TEXTBOOKS_AUTH</db2-xdb:rowSet>
              <db2-xdb:column>AUTHNAME</db2-xdb:column>
            </db2-xdb:rowSetMapping>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      <xs:element name="publicationDate" type="xs:gYear"/>
      <xs:element name="university" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="title" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

Notice how the `<isbn>` element is mapped only once to the ISBN column, yet it appears in two rows in the table. This happens automatically during the decomposition process because there are multiple authors per ISBN value. The value of `<isbn>` is duplicated in each row for every author.

This behavior occurs because a one to many relationship is detected between the `<isbn>` and `<author>` elements, as the `maxOccurs` attribute for `<author>` is greater than 1.

Note that a one to many relationship can involve more than two items, and include sets of items. The one to many relationship can also be deeply nested, where an item already involved in a one to many relationship can participate in another one to many relationship.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Examples of mappings in annotated XML schema decomposition” on page 296
- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “db2-xdb:column decomposition annotation” on page 257
- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “XML decomposition annotations - Summary” on page 246

Decomposition annotation example: A value mapped to multiple tables

In annotated XML schema decomposition, a single value from an XML document can be mapped to multiple tables. This example shows how to annotate an XML schema document to map a single value to two tables.

Consider the following XML document.

```
<textbook title="Programming with XML">
  <isbn>0-11-011111-0</isbn>
  <author>Mary Brown</author>
  <author>Alex Page</author>
  <publicationDate>2002</publicationDate>
  <university>University of London</university>
</textbook>
```

To map a textbook’s ISBN to the following two tables, you need to create two mappings on the <isbn> element. This can be done by adding multiple <db2-xdb:rowSetMapping> elements to the <isbn> element declaration in the XML schema document.

Table 70. TEXTBOOKS

ISBN	TITLE
0-11-011111-0	Programming with XML

Table 71. SCHOOLPUBS

ISBN	SCHOOL
0-11-011111-0	University of London

The following fragment of the XML schema document shows how two mappings are added to the <isbn> element declaration to specify the mappings to two tables. The value of the title attribute and <university> element also included in the mappings.

```
<xs:element name="textbook" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="isbn" type="xs:string">
        <xs:annotation>
          <xs:appinfo>
            <db2-xdb:rowSetMapping>
              <db2-xdb:rowSet>TEXTBOOKS</db2-xdb:rowSet>
              <db2-xdb:column>ISBN</db2-xdb:column>
            </db2-xdb:rowSetMapping>
            <db2-xdb:rowSetMapping>
              <db2-xdb:rowSet>SCHOOLPUBS</db2-xdb:rowSet>
              <db2-xdb:column>ISBN</db2-xdb:column>
            </db2-xdb:rowSetMapping>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```

<xs:element name="publicationDate" type="xs:gYear"/>
<xs:element name="university" type="xs:string" maxOccurs="unbounded">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetMapping>
        <db2-xdb:rowSet>SCHOOLPUBS</db2-xdb:rowSet>
        <db2-xdb:column>SCHOOL</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
</xs:sequence>
<xs:attribute name="title" type="xs:string" use="required">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetMapping>
        <db2-xdb:rowSet>TEXTBOOKS</db2-xdb:rowSet>
        <db2-xdb:column>TITLE</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</xs:attribute>
</xs:complexType>
</xs:element>

```

Complex types that appear multiple times:

If a complex type is referred to in multiple places in an XML schema, you can map it to different tables and columns depending on its location in the schema, using the `db2-xdb:locationPath` annotation.

In this case, the complex type element or attribute declaration needs to be annotated with multiple `<db2-xdb:rowSetMapping>` annotations (one for each mapping) where each mapping is distinguished by the `db2-xdb:locationPath` attribute.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Examples of mappings in annotated XML schema decomposition” on page 296
- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “`db2-xdb:column` decomposition annotation” on page 257
- “`db2-xdb:locationPath` decomposition annotation” on page 259
- “`db2-xdb:rowSet` decomposition annotation” on page 249
- “`db2-xdb:rowSetMapping` decomposition annotation” on page 280
- “XML decomposition annotations - Summary” on page 246

Decomposition annotation example: Grouping multiple values mapped to a single table

In annotated XML schema decomposition, you can map multiple values from unrelated elements to the same table, while preserving the relationship between logically-related values. This is possible by declaring multiple `rowSets`, which are used to group related items to form a row.

For example, consider the following XML document:

```
<publications>
  <textbook title="Programming with XML">
    <isbn>0-11-011111-0</isbn>
    <author>Mary Brown</author>
    <author>Alex Page</author>
    <publicationDate>2002</publicationDate>
    <university>University of London</university>
  </textbook>
  <childrensbook title="Children's Fables">
    <isbn>5-55-555555-5</isbn>
    <author>Bob Carter</author>
    <author>Melaine Snowe</author>
    <publicationDate>1999</publicationDate>
  </childrensbook>
</publications>
```

To generate the following table after decomposition, you need to ensure that values relating to a textbook are not grouped in the same row as values associated with a children's book. Use multiple rowSets to group related values and yield logically meaningful rows.

Table 72. ALLPUBLICATIONS

PUBS_ISBN	PUBS_TITLE
0-11-011111-0	Programming with XML
5-55-555555-5	Children's Fables

In a simple mapping scenario, where you are mapping a single value to a single table and column pair, you could just specify the table and column you want to map the value to.

This example shows a more complex case, however, where multiple values are mapped to the same table and must be logically grouped. If you were to simply map each ISBN and title to the PUBS_ISBN and PUBS_TITLE columns, without the use of rowSets, the decomposition process would not be able to determine which ISBN value belonged with which title value. By using rowSets, you can group logically related values to form a meaningful row.

The following XML schema document shows how two rowSets are defined to distinguish values of the <textbook> element from values of the <childrensbook> element.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2/xdb1"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:table>
        <db2-xdb:name>ALLPUBLICATIONS</db2-xdb:name>
        <db2-xdb:rowSet>textbk_rowSet</db2-xdb:rowSet>
        <db2-xdb:rowSet>childrens_rowSet</db2-xdb:rowSet>
      </db2-xdb:table>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="publications">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="textbook" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
```

```

        <xs:element name="isbn" type="xs:string"
            db2-xdb:rowSet="textbk_rowSet" db2-xdb:column="PUBS_ISBN"/>
        <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
        <xs:element name="publicationDate" type="xs:gYear"/>
        <xs:element name="university" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="title" type="xs:string" use="required"
        db2-xdb:rowSet="textbk_rowSet" db2-xdb:column="PUBS_TITLE"/>
</xs:complexType>
</xs:element>
<xs:element name="childrensbook" maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="isbn" type="xs:string"
                db2-xdb:rowSet="childrens_rowSet" db2-xdb:column="PUBS_ISBN"/>
            <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
            <xs:element name="publicationDate" type="xs:gYear"/>
        </xs:sequence>
        <xs:attribute name="title" type="xs:string" use="required"
            db2-xdb:rowSet="childrens_rowSet" db2-xdb:column="PUBS_TITLE"/>
    </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

Notice how the `db2-xdb:rowSet` mappings in each of the element and attribute declarations do not specify the name of a table, but rather the name of a rowSet. The rowSets are associated with the ALLPUBLICATIONS table in the `<db2-xdb:table>` annotation, which must be specified as a child of `<xs:schema>`.

By specifying multiple rowSets that map to the same table, you can ensure that logically related values form a row in the table.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Examples of mappings in annotated XML schema decomposition” on page 296
- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “db2-xdb:column decomposition annotation” on page 257
- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “db2-xdb:table decomposition annotation” on page 254
- “XML decomposition annotations - Summary” on page 246

Decomposition annotation example: Multiple values from different contexts mapped to a single table

In annotated XML schema decomposition, you can map multiple values to the same table and column, such that a single column can contain values that have come from different parts of a document. This is possible by declaring multiple rowSets.

For example, consider the following XML document:

```

<publications>
  <textbook title="Principles of Mathematics">
    <isbn>1-11-111111-1</isbn>
    <author>Alice Braun</author>
    <publisher>Math Pubs</publisher>
    <publicationDate>2002</publicationDate>
    <university>University of London</university>
  </textbook>
</publications>

```

You can map both the author and the publisher to the same table that contains contacts for a particular book.

Table 73. BOOKCONTACTS

ISBN	CONTACT
1-11-111111-1	Alice Braun
1-11-111111-1	Math Pubs

The values in the CONTACT column of the resulting table come from different parts of the XML document: one row might contain an author's name (from the <author> element, while another row contains a publisher's name (from the <publisher> element).

The following XML schema document shows how multiple rowSets can be used to generate this table.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2/xdb1"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:table>
        <db2-xdb:name>BOOKCONTACTS</db2-xdb:name>
        <db2-xdb:rowSet>author_rowSet</db2-xdb:rowSet>
        <db2-xdb:rowSet>publisher_rowSet</db2-xdb:rowSet>
      </db2-xdb:table>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="publications">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="textbook" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="isbn" type="xs:string">
                <xs:annotation>
                  <xs:appinfo>
                    <db2-xdb:rowSetMapping>
                      <db2-xdb:rowSet>author_rowSet</db2-xdb:rowSet>
                      <db2-xdb:column>ISBN</db2-xdb:column>
                    </db2-xdb:rowSetMapping>
                    <db2-xdb:rowSetMapping>
                      <db2-xdb:rowSet>publisher_rowSet</db2-xdb:rowSet>
                      <db2-xdb:column>ISBN</db2-xdb:column>
                    </db2-xdb:rowSetMapping>
                  </xs:appinfo>
                </xs:annotation>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="author" type="xs:string" maxOccurs="unbounded">
          <xs:annotation>
            <xs:appinfo>
              <db2-xdb:rowSetMapping>
                <db2-xdb:rowSet>author_rowSet</db2-xdb:rowSet>
              </db2-xdb:rowSetMapping>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

```

```

        <db2-xdb:column>CONTACT</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
<xs:element name="publisher" type="xs:string">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetMapping>
        <db2-xdb:rowSet>publisher_rowSet</db2-xdb:rowSet>
        <db2-xdb:column>CONTACT</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
<xs:element name="publicationDate" type="xs:gYear"/>
<xs:element name="university" type="xs:string"
  maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="title" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

Notice how the `db2-xdb:rowSet` mappings in each of the element declarations do not specify the name of a table, but rather the name of a rowSet. The rowSets are associated with the `BOOKCONTACTS` table in the `<db2-xdb:table>` annotation, which must be specified as a child of `<xs:schema>`.

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “Examples of mappings in annotated XML schema decomposition” on page 296
- “rowSets in annotated XML schema decomposition” on page 294
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “db2-xdb:column decomposition annotation” on page 257
- “db2-xdb:rowSet decomposition annotation” on page 249
- “db2-xdb:rowSetMapping decomposition annotation” on page 280
- “db2-xdb:table decomposition annotation” on page 254
- “XML decomposition annotations - Summary” on page 246

Decomposition XML schema to SQL type compatibility

XML schema to SQL types compatibility for annotated schema decomposition

Annotated XML schema decomposition only supports the decomposition of XML values into columns that are of a compatible SQL type. You can use the `db2-xdb:expression` annotation to convert values to a compatible format, however. The following tables describe the SQL column types that are compatible with the XML schema types. Refer to each table for specific conditions of compatibility.

- SQL character types

- SQL datetime types
- SQL graphic types
- SQL LOB and for bit data types
- SQL numeric types

Related concepts:

- “Annotated XML schema decomposition” on page 235

Related reference:

- “Annotated XML schema decomposition - Limits and restrictions” on page 317

XML schema to SQL type compatibility for decomposition - Character types

Annotated XML schema decomposition enables XML values to be stored in columns of tables. XML values can only be decomposed, however, into compatible SQL columns. The following table lists which XML schema types are compatible with character string SQL column types.

Legend:

- 1a** Compatible if the length of the XML input string, in bytes, is less than or equal to the length of the target column in bytes. If the input string is longer than the target column, then the string is compatible only if `db2-xdb:truncate` is set to “true” or “1” for this column mapping. String length is computed after normalization, where the input string is normalized according to the whitespace facet of the XML schema type.
- 1b** Compatible according to the conditions described in 1a. The value that is inserted into the target column is the string of concatenated list items, each separated by a single space (in accordance with the “collapse” whitespace facet for lists).
- 2** Compatible if the length of the XML input string, in bytes, is less than or equal to the length of the target column in bytes. If the input string is longer than the target column, then the string is compatible only if `db2-xdb:truncate` is set to “true” or “1” for this column mapping. The input string is normalized according to the whitespace facet of the XML schema type. The encoded (original) string is inserted.
- 3a** Compatible if the length of the XML input string, computed after processing according to the `db2-xdb:normalization` setting, is less than or equal to the length of the target column. Also compatible if `db2-xdb:truncate` is set to “true” or “1” for this column mapping.
- 3b** Compatible according to the conditions described in 3a. The `db2-xdb:normalization` annotation is used to determine the format of the inserted string.
- 4a** Compatible if the length of the XML input string, in bytes, is less than or equal to the length of the target column in bytes. If the input string is longer than the target column, then the string is compatible only if `db2-xdb:truncate` is set to “true” or “1” for this column mapping. The value that is inserted into the target column in either case is the character content of the element or attribute.
- 4b** Compatible according to the conditions described in 5a. Additionally, the

annotation on the element declaration must specify `db2-xdb:contentHandling="serializeSubtree"`.

- 5 Compatible if the string length of the URI, in bytes, is less than or equal to the length of the target column in bytes. If the input string is longer than the target column, then the string is compatible only if `db2-xdb:truncate` is set to "true" or "1" for this column mapping. Note that the URI itself, not the resource the URI points to, is inserted.

Table 74. Compatible XML schema and SQL character string types

XML schema type	SQL type		
	CHAR	VARCHAR	LONG VARCHAR
string, normalizedString, token	1a*	1a	1a
base64Binary	2*	2	2
hexBinary	2*	2	2
byte, unsigned byte	3b*	3b	3b
integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger	3b*	3b	3b
int	3b*	3b	3b
unsignedInt	3a*	3a	3a
long	3b*	3b	3b
unsignedLong	3a*	3a	3a
short	3b*	3b	3b
unsignedShort	3a*	3a	3a
decimal	3b*	3b	3b
float	3b*	3b	3b
double	3b*	3b	3b
boolean	3b*	3b	3b
time	3b*	3b	3b
dateTime	3b*	3b	3b
duration, gMonth, gYear, gDay, gMonthDay, gYearMonth	3a*	3a	3a
date	3b*	3b	3b
Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY	1a*	1a	1a
ENTITIES, NMTOKENS, IDREFS, list types	1b*	1b	1b
anyURI	5*	5	5
language	1a*	1a	1a
anySimpleType, union types	4a*	4a	4a
anyType	4b*	4b	4b

* If the length of the input XML string is less than the defined length of the target column, then the string is right-padded with blanks when inserted.

Related concepts:

- "Annotated XML schema decomposition" on page 235

Related reference:

- "db2-xdb:contentHandling decomposition annotation" on page 269
- "db2-xdb:normalization decomposition annotation" on page 274
- "db2-xdb:truncate decomposition annotation" on page 278

- “XML schema to SQL types compatibility for annotated schema decomposition” on page 307
- “Character strings” in *SQL Reference, Volume 1*

XML schema to SQL type compatibility for decomposition - Datetime types

Annotated XML schema decomposition enables XML values to be stored in columns of tables. XML values can only be decomposed, however, into compatible SQL columns. The following table lists which XML schema types are compatible with datetime SQL column types.

Legend:

- Data types are not compatible for annotated XML schema decomposition.
- 1 Compatible if the string is of a valid date format: *yyyy-mm-dd*, *mm/dd/yyyy*, or *dd.mm.yyyy*.
- 2 Compatible if the string is of a valid time format: *hh.mm.ss*, *hh:mm AM* or *PM*, or *hh:mm:ss*.
- 3 Compatible if the string is of a valid timestamp format: *yyyy-mm-dd-hh.mm.ss.nnnnnn* or *yyyy-mm-dd hh.mm.ss.nnnnnn*.
- 4
 - For XML values that contain subseconds, compatible only if the decomposition annotation specifies `db2-xdb:truncate` as “true” or “1”.
 - For XML values with time zone indicators, compatible if `db2-xdb:truncate` is set to “true” or “1”; values are inserted without the time zone.
- 5 Compatible if the year is composed of four digits and is not preceded by the ‘-’ sign. Compatible if the XML value does not have a time zone indicator. If the XML value has a time zone indicator, then the values are compatible if `db2-xdb:truncate` is set to “true” or “1”.
- 6 Compatible if the XML value does not have a time zone indicator. If the XML value has a time zone indicator, then the values are compatible if `db2-xdb:truncate` is set to “true” or “1”.
- 7
 - Compatible if the year is composed of four digits and is not preceded by the ‘-’ sign.
 - For XML values with time zone indicators, compatible if `db2-xdb:truncate` is set to “true” or “1”. (Values are inserted without the time zone in this case.)
 - If subseconds are specified with more than six digits, compatible if `db2-xdb:truncate` is set to “true” or “1”.
- 8
 - Compatible if the year is composed of four digits and is not preceded by the ‘-’ sign.
 - For XML values with time zone indicators, compatible if `db2-xdb:truncate` is set to “true” or “1”. (Date values are inserted without the time zone in this case.)

Table 75. Compatible XML schema and SQL datetime types

XML schema type	SQL type		
	DATE	TIME	TIMESTAMP
string, normalizedString, token	1	2	3
base64Binary	–	–	–
hexBinary	–	–	–
byte, unsigned byte	–	–	–
integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger	–	–	–
int, unsignedInt	–	–	–
long, unsignedLong	–	–	–
short, unsignedShort	–	–	–
decimal	–	–	–
float	–	–	–
double	–	–	–
boolean	–	–	–
time	–	4	–
dateTime	5	6	7
duration, gMonth, gYear, gDay, gMonthDay, gYearMonth	–	–	–
date	8	–	–
Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY	–	–	–
ENTITIES, NMTOKENS, IDREFS, list types	–	–	–
anyURI	–	–	–
language	–	–	–
anySimpleType, union types	–	–	–
anyType	–	–	–

Related concepts:

- “Annotated XML schema decomposition” on page 235

Related tasks:

- “Decomposing XML documents with annotated XML schemas” on page 236

Related reference:

- “db2-xdb:truncate decomposition annotation” on page 278
- “XML schema to SQL types compatibility for annotated schema decomposition” on page 307
- “Datetime values” in *SQL Reference, Volume 1*

XML schema to SQL type compatibility for decomposition - Graphic types

Annotated XML schema decomposition enables XML values to be stored in columns of tables. XML values can only be decomposed, however, into compatible SQL columns. The following table lists which XML schema types are compatible with SQL graphic column types.

Legend:

- Data types are not compatible for annotated XML schema decomposition.
- 1a** Compatible if the length of the XML input string, in bytes, is less than or equal to the length of the target column in bytes. The input string must be composed of double-byte characters. If the input string is longer than the target column, then the string is compatible only if db2-xdb:truncate is set to "true" or "1" for this column mapping. String length is computed after normalization, where the input string is normalized according to the whitespace facet of the XML schema type.
- 1b** Compatible according to the conditions described in 1a. The value that is inserted into the target column is the string of concatenated list items, each separated by a single space (in accordance with the "collapse" whitespace facet for lists).
- 2a** Compatible if the length of the XML input string, in bytes, is less than or equal to the length of the target column in bytes. The input string must be composed of double-byte characters. If the input string is longer than the target column, then the string is compatible only if db2-xdb:truncate is set to "true" or "1" for this column mapping. The value that is inserted into the target column in either case is the character content of the element or attribute.
- 2b** Compatible according to the conditions described in 2a. Additionally, the annotation on the element declaration must specify db2-xdb:contentHandling="serializeSubtree".

Table 76. Compatible XML schema and SQL graphic types

XML schema type	SQL type		
	GRAPHIC	VARGRAPHIC	LONG VARGRAPHIC
string, normalizedString, token	1a*	1a	1a
base64Binary	-	-	-
hexBinary	-	-	-
byte, unsigned byte	-	-	-
integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger	-	-	-
int, unsignedInt	-	-	-
long, unsignedLong	-	-	-
short, unsignedShort	-	-	-
decimal	-	-	-
float	-	-	-
double	-	-	-
boolean	-	-	-
time	-	-	-
dateTime	-	-	-
duration, gMonth, gYear, gDay, gMonthDay, gYearMonth	-	-	-
date	-	-	-
Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY	1a*	1a	1a
ENTITIES, NMTOKENS, IDREFS, list types	1b*	1b	1b

Table 76. Compatible XML schema and SQL graphic types (continued)

XML schema type	SQL type		
	GRAPHIC	VARGRAPHIC	LONG VARGRAPHIC
anyURI	–	–	–
language	–	–	–
anySimpleType, union types	2a*	2a	2a
anyType	2b*	2b	2b

* If the length of the input XML string is less than the defined length of the target column, then the string is right-padded with blanks when inserted.

Related concepts:

- “Annotated XML schema decomposition” on page 235

Related reference:

- “db2-xdb:truncate decomposition annotation” on page 278
- “Graphic strings” in *SQL Reference, Volume 1*
- “XML schema to SQL types compatibility for annotated schema decomposition” on page 307

XML schema to SQL type compatibility for decomposition - LOB and for bit data types

Annotated XML schema decomposition enables XML values to be stored in columns of tables. XML values can only be decomposed, however, into compatible SQL columns. The following table lists which XML schema types are compatible with SQL LOB and for bit data column types.

Legend:

- Data types are not compatible for annotated XML schema decomposition.
- 1a** Compatible if the length of the XML input string, in bytes, is less than or equal to the length of the target column in bytes. If the input string is longer than the target column, then the string is compatible only if db2-xdb:truncate is set to “true” or “1” for this column mapping. String length is computed after normalization, where the input string is normalized according to the whitespace facet of the XML schema type.
- 1b** Compatible according to the conditions described in 1a. The value that is inserted into the target column is the string of concatenated list items, each separated by a single space (in accordance with the “collapse” whitespace facet for lists).
- 1c** Compatible according to the conditions described in 1a. The value that is inserted into the target column is the encoded (original) string.
- 1d** Compatible according to the conditions described in 1a. The value that is inserted into the target column is the decoded string.
- 2** Compatible if the length of the XML input string, in bytes, computed after processing according to the db2-xdb:normalization setting, is less than or equal to the length of the target column in bytes. Also compatible if db2-xdb:truncate is set to “true” or “1” for this column mapping.
- 3** Compatible if the string length of the URI, in bytes, is less than or equal to the length of the target column in bytes. If the input string is longer than

the target column, then the string is compatible only if db2-xdb:truncate is set to "true" or "1" for this column mapping. Note that the URI itself, not the resource the URI points to, is inserted.

- 4a Compatible if the length of the XML input string, in bytes, is less than or equal to the length of the target column in bytes. If the input string is longer than the target column, then the string is compatible only if db2-xdb:truncate is set to "true" or "1" for this column mapping. The value that is inserted into the target column in either case is the character content of the element or attribute.
- 4b Compatible according to the conditions described in 4a. Additionally, the annotation on the element declaration must specify db2-xdb:contentHandling="serializeSubtree".

Table 77. Compatible XML schema and SQL LOB and for bit data types

XML schema type	SQL type			
	CHAR for bit data	BLOB, LONG VARCHAR for bit data, VARCHAR for bit data	CLOB	DBCLOB
string, normalizedString, token	1a ¹	1a	1a	1a ²
base64Binary	1d ¹	1d	1c	–
hexBinary	1d ¹	1d	1c	–
byte, unsigned byte	–	–	2 ³	–
integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger	–	–	2 ³	–
int	–	–	2 ³	–
unsignedInt	–	–	2	–
long	–	–	2 ³	–
unsignedLong	–	–	2	–
short	–	–	2 ³	–
unsignedShort	–	–	2	–
decimal	–	–	2 ³	–
float	–	–	2 ³	–
double	–	–	2 ³	–
boolean	–	–	2 ³	–
time	–	–	2 ³	–
dateTime	–	–	2 ³	–
duration, gMonth, gYear, gDay, gMonthDay, gYearMonth	–	–	2 ³	–
date	–	–	2 ³	–
Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY	1a ¹	1a	1a	1a ²
ENTITIES, NMTOKENS, IDREFS, list types	1b ¹	1b	1b	1b ²
anyURI	3 ¹	3	3	–
language	1a ¹	1a	1a	–
anySimpleType, union types	4a ¹	4a	4a	4a ²
anyType	4b ¹	4b	4b	4b ²

Table 77. Compatible XML schema and SQL LOB and for bit data types (continued)

XML schema type	SQL type		
	CHAR for bit data	BLOB, LONG VARCHAR for bit data, VARCHAR for bit data	CLOB
Notes: <ol style="list-style-type: none"> 1. If the length of the input XML string is less than the defined length of the target column, then the string is right-padded with blanks when inserted. 2. XML input value must be composed of double-byte characters. 3. The db2-xdb:normalization annotation is used to determine the format of the inserted string. 			

Related reference:

- “Binary strings” in *SQL Reference, Volume 1*
- “Large objects (LOBs)” in *SQL Reference, Volume 1*
- “db2-xdb:normalization decomposition annotation” on page 274
- “db2-xdb:truncate decomposition annotation” on page 278
- “XML schema to SQL types compatibility for annotated schema decomposition” on page 307

XML schema to SQL type compatibility for decomposition - Numeric types

Annotated XML schema decomposition enables XML values to be stored in columns of tables. XML values can only be decomposed, however, into compatible SQL columns. The following table lists which XML schema types are compatible with numeric SQL column types.

Legend:

- Data types are not compatible for annotated XML schema decomposition.
- 1 Compatible if the string is in an acceptable lexical form for the target SQL type and can be converted to a numeric value in the range of the SQL type. Loss of significant digits can occur.
- 2 Compatible, and where –0 is in the value space of the XML type, –0 is stored as 0 in the database.
- 3 Compatible if the XML type is in the range of the SQL type. Where –0 is in the value space of the XML type, –0 is stored as 0 in the database.
- 4 Compatible if the XML value is in the range of the SQL type. Loss of significant digits can occur. Where –0 is in the value space of the XML type, –0 is stored as 0 in the database.
- 5 Compatible if the value is in the range of the SQL type and is not “INF”, “–INF” or “NaN”. Where –0 is in the value space of the XML type, –0 is stored as 0 in the database. Loss of significant digits can occur.
- 6 Compatible if the value is not “INF”, “–INF” or “NaN”. Where –0 is in the value space of the XML type, –0 is stored as 0 in the database.
- 7 Compatible, and the value inserted is ‘0’ (for false) or ‘1’ (for true).
- 8 The fractional part of the number is truncated. Compatible if the whole part is in the range of the SQL type. Where –0 is in the value space of the XML type, –0 is stored as 0 in the database.

- 9 The fractional part of the number is truncated. Compatible if the whole part is in the range of the SQL type, and the value is not "INF", "-INF" or "NaN". Where -0 is in the value space of the XML type, -0 is stored as 0 in the database.

Table 78. Compatible XML schema and numeric SQL data types

XML schema type	SQL type					
	SMALLINT	INTEGER	BIGINT	REAL	DECIMAL	DOUBLE
string, normalizedString, token	1	1	1	1	1	1
base64Binary	-	-	-	-	-	-
hexBinary	-	-	-	-	-	-
byte, unsigned byte	2	2	2	2	2	2
integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger	3	3	3	4	4	4
int	3	2	2	4	4	2
unsignedInt	3	3	2	4	4	2
long	3	3	2	4	4	4
unsignedLong	3	3	3	4	4	4
short	2	2	2	2	2	2
unsignedShort	3	2	2	2	2	2
decimal	8	8	8	4	4	4
float	9	9	9	6	5	6
double	9	9	9	5	5	6
boolean	7	7	7	7	7	7
time	-	-	-	-	-	-
dateTime	-	-	-	-	-	-
duration, gMonth, gYear, gDay, gMonthDay, gYearMonth	-	-	-	-	-	-
date	-	-	-	-	-	-
Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY	-	-	-	-	-	-
ENTITIES, NMTOKENS, IDREFS, list types	-	-	-	-	-	-
anyURI	-	-	-	-	-	-
language	-	-	-	-	-	-
anySimpleType, union types	-	-	-	-	-	-
anyType	-	-	-	-	-	-

Related concepts:

- "Annotated XML schema decomposition" on page 235

Related reference:

- "Numbers" in *SQL Reference, Volume 1*

- “XML schema to SQL types compatibility for annotated schema decomposition” on page 307

Annotated XML schema decomposition - Limits and restrictions

Limits:

Table 79. Limits for annotated XML schema decomposition

Condition	Limit value
Maximum size of document to be decomposed	100 MB
Maximum number of tables referred to in a single annotated XML schema	100
Maximum number of \$DECOMP_CONTENT or \$DECOMP_CONTENT instances in an db2-xdb:expression annotation	10
Maximum number of steps in db2-xdb:locationPath	100
Maximum number of namespaces explicitly listed in the "namespace" attribute of <xs:any> or <xs:anyAttribute> (if the list contains the special values ##targetNamespace or ##local, then these also count towards the limit)	25
Maximum string length of the value of db2-xdb:name (table name), db2-xdb:column, db2-xdb:defaultSQLSchema, or db2-xdb:SQLSchema	same as the limit for the corresponding DB2 object
Maximum string length of the value of db2-xdb:rowSet	same as the limit for db2-xdb:name
Maximum string length for value of \$DECOMP_CONTENT	1024 bytes

Restrictions:

Annotated XML schema decomposition does not support the following:

- decomposition of element of attribute wildcards: elements or attributes in the XML document that correspond to the <xs:any> or <xs:anyAttribute> declaration in the XML schema are not decomposed.

If, however, these elements or attributes are children of elements that are decomposed with db2-xdb:contentHandling set to "serializeSubtree" or "stringValue", then the contents of the wildcard elements or attributes will be decomposed as part of the serialized subtree or string value. These wildcard element or attributes must satisfy the namespace constraints specified in the corresponding <xs:any> or <xs:anyAttribute> declaration, however, in order to be part of the serialization.

- substitution groups: an error is generated if a member of a substitution group appears in the XML document where the group head appears in the XML schema, for cases when the substitution group members are used not only as the root element of the document.

As a workaround, the element declarations of the substitution group's head and members can instead be changed to a named model group of type `xs:choice`. For example, these substitution group declarations

```
<xs:element name="head" type="BaseType" />
<xs:element name="member1" type="derived1FromBaseType" substitutionGroup="head"/>
<xs:element name="member2" type="derived2FromBaseType" substitutionGroup="head"/>
<xs:element name="member3" type="derived3FromBaseType" substitutionGroup="head"/>
```

can be changed to an equivalent named model group:

```
<xs:group name="mysubstitutiongrp">
  <xs:choice>
    <xs:element name="head" type="BaseType"/>
    <xs:element name="member1" type="derived1FromBaseType"/>
    <xs:element name="member2" type="derived2FromBaseType"/>
    <xs:element name="member3" type="derived3FromBaseType"/>
  </xs:choice>
</xs:group>
```

Occurrences of the `<head>` element can then be replaced with the newly defined named model group in the XML document.

- runtime substitution using `xsi:type`: an element is decomposed according to the mappings in the schema type associated with the element name in the schema. Specifying a different type for an element in the document through the use of `xsi:type` results in an error being returned during decomposition.

Ensure that the type of an element specified with `xsi:type` in the XML document matches the type specified for that element in the context. If the content of the element or its descendants does not need to be decomposed individually, then the type of the element can be changed to `xs:anyType` in the XML schema. With this change, the XML documents do not need to be modified.

- recursive elements: annotated schemas that contain recursive element declarations cannot be enabled for decomposition.

If the recursive elements do not need to be decomposed, then the recursive element declarations in the XML schema can be changed such that they are declared to be of type `xs:anyType`, instead of the recursive type. With this change, the XML schema can still be enabled for decomposition, and corresponding documents can be decomposed.

For example,

```
<xs:element name="a" type="recursiveType"/>

<xs:complexType name="recursiveType">
  <xs:sequence>
    <xs:element name="b" type="xs:string"/>
    <xs:element ref="a"/>
  </xs:sequence>
</xs:complexType>
```

can be changed to

```
<xs:element name="a" type="recursiveType"/>

<xs:complexType name="recursiveType">
  <xs:sequence>
    <xs:element name="b" type="xs:string"/>
    <xs:element name="a" type="xs:anyType"/>
  </xs:sequence>
</xs:complexType>
```

Note that mapped elements of type `xs:anyType`, must have the specification `db2-xdb:contentHandling="serializeSubtree."`

Note: Apply this workaround only when you can ensure that the document to be decomposed is going to be valid.

- updates to or deletion of existing rows in target tables: decomposition only supports the insertion of new rows. (You can still update or delete rows outside of the XML decomposition process.)
- attributes of simple type derived from NOTATION: decomposition inserts only the notation name.
- attributes of type ENTITY: decomposition inserts only the entity name.
- referential integrity constraints: the order of insertion based on referential integrity constraints is not observed.

To workaround this restriction, the same XML document can be decomposed twice to yield results that satisfy referential integrity constraints. First, maintain two copies of the document's XML schema. Annotate one schema to map only to the table with the primary key, and annotate the other schema to map only to the table with the foreign key. Then register both XML schemas with the XML schema repository (XSR). Decompose the XML document using the schema that maps to the table with the primary key. Then decompose the same XML document with the schema that maps to the table with the foreign key. This method ensures that rows are inserted in the correct order for tables that have referential integrity constraints defined on them.

- multiple mappings to the same rowSet and column with db2-xdb:expression and db2-xdb:condition: where multiple items can be legally mapped to the same rowSet and column, according to mapping rules, the mappings must not contain the db2-xdb:expression or db2-xdb:condition annotations.

Related concepts:

- "Annotated XML schema decomposition" on page 235
- "XML schema, DTD, and external entity management using the XML schema repository (XSR)" on page 133

Related reference:

- "SQL and XQuery limits" in *SQL Reference, Volume 1*
- "db2-xdb:condition decomposition annotation" on page 266
- "db2-xdb:contentHandling decomposition annotation" on page 269
- "db2-xdb:defaultSQLSchema decomposition annotation" on page 247
- "db2-xdb:expression decomposition annotation" on page 263
- "db2-xdb:locationPath decomposition annotation" on page 259
- "db2-xdb:rowSet decomposition annotation" on page 249
- "db2-xdb:table decomposition annotation" on page 254
- "Keywords for annotated XML schema decomposition" on page 284

Troubleshooting considerations for annotated XML schema decomposition

If you find that decomposition is not yielding expected results, consider the following items.

General:

- Check that the XSR object that corresponds to your XML schema appears as enabled in the DECOMPOSITION column of the SYSCAT.XSROBJECTS catalog

view. If the XSR object is not enabled, consider taking the corrective actions described in the disablement documentation.

- Ensure that the limits and restrictions for XML decomposition are not violated.
- Ensure that the XML document is valid according to its XML schema. Validation is not a requirement for decomposition, however, if you expect certain behavior, such as character entity expansion, then perform decomposition with validation.

XML schema:

- Ensure that XML schema does not contain errors such as non-deterministic content models, as these types of errors can cause decomposition to fail when validation is performed, or undefined decomposition results if validation is not performed.
- Ensure that the non-global annotations have been declared only on element or attribute declarations and not on complex types, element/attribute references, model groups or any other xml schema construct. Also check that the annotations are declared in their supported format: as attributes, elements, or global annotations. (Refer to the documentation on each annotation for details of how to specify an annotation.)
- Ensure that complex types derived by extension or restriction are annotated properly.

Specific errors:

Adjusting database configuration parameters can resolve the following errors:

- SQL0954 received when the annotated XML schema contains a large number of rowSets: increase the application heap size using the applheapsz configuration parameter
- SQL0954 received when the annotated XML schema contains complex or many expressions in each rowSet: increase the application heap size using the applheapsz configuration parameter
- SQL0964 received when decomposition results in a large number of rows: increase the number of primary or secondary log files available using the logprimary and logsecond configuration parameters. You can also increase the size of the primary and secondary log files with the logfilesz configuration parameter.

Locking and concurrency:

If you are experiencing lock escalation or deadlock when decomposing documents, adjust concurrency control through your application. If an application makes multiple concurrent invocations of any of the xdbDecompXML stored procedures, where many of the same tables are involved in the multiple decomposition operations, then the application needs to manage the concurrent access to these tables to prevent lock escalation and deadlock.

One way to adjust concurrency control is to explicitly lock all tables involved in the decomposition before invoking the xdbDecompXML stored procedure. Then execute the COMMIT or ROLLBACK statements as appropriate after the stored procedure has returned. Because the decomposition of large documents can result in a large number of rows being inserted, and because each row is locked by default during an insert operation, an application that is inserting many rows can hold many row locks, leading to lock escalation. By obtaining table locks instead, you can avoid the overhead of obtaining row locks and of lock escalation.

If the reduced concurrency associated with obtaining table locks is not appropriate for your application, you can increase either or both of the maxlocks and locklist database configuration parameters, which decreases the likelihood of lock escalation.

Set the locktimeout database configuration parameter to prevent an application from waiting indefinitely to obtain a lock.

Mapping verification in catalog view:

If you are still experiencing problems with decomposition after verifying the conditions above, check that the MAPPINGDESCRIPTION column of the SYSCAT.XDBMAPSHREDTREES catalog view matches the mappings you intended. The MAPPINGDESCRIPTION column contains details on how each item in a rowSet was mapped, including:

- target column name
- target column type
- item's XML schema type
- values specified for db2-xdb:contentHandling, db2-xdb:normalization, db2-xdb:truncate, db2-xdb:expression, and db2-xdb:condition

Note that the columns of SYSCAT.XDBMAPSHREDTREES other than MAPPINGDESCRIPTION are intended for DB2 customer support.

Related concepts:

- "Deadlocks between applications" in *Performance Guide*
- "Annotations of derived complex types for annotated XML schema decomposition" on page 289
- "Locks and concurrency control" in *Performance Guide*
- "Disabling of annotated XML schema decomposition" on page 239
- "rowSets in annotated XML schema decomposition" on page 294
- "Effect of validation on XML decomposition results" on page 286
- "XML decomposition annotations - Overview" on page 244

Related tasks:

- "Registering and enabling XML schemas for decomposition" on page 237

Related reference:

- "SYSCAT.XDBMAPSHREDTREES catalog view" in *SQL Reference, Volume 1*
- "SYSCAT.XSROBJECTS catalog view" in *SQL Reference, Volume 1*
- "applheapsz - Application heap size configuration parameter" in *Performance Guide*
- "COMMIT statement" in *SQL Reference, Volume 2*
- "locklist - Maximum storage for lock list configuration parameter" in *Performance Guide*
- "locktimeout - Lock timeout configuration parameter" in *Performance Guide*
- "logfilsiz - Size of log files configuration parameter" in *Performance Guide*
- "logprimary - Number of primary log files configuration parameter" in *Performance Guide*
- "logsecond - Number of secondary log files configuration parameter" in *Performance Guide*

- “maxlocks - Maximum percent of lock list before escalation configuration parameter” in *Performance Guide*
- “db2-xdb:condition decomposition annotation” on page 266
- “ROLLBACK statement” in *SQL Reference, Volume 2*
- “db2-xdb:contentHandling decomposition annotation” on page 269
- “db2-xdb:expression decomposition annotation” on page 263
- “db2-xdb:truncate decomposition annotation” on page 278
- “Annotated XML schema decomposition - Limits and restrictions” on page 317
- “xdbDecompXML stored procedures for annotated schema decomposition” on page 240

Schema for XML decomposition annotations

Annotated XML schema decomposition supports a set of decomposition annotations that enable you to specify how XML documents are to be decomposed and inserted into database tables. The XML schema for the annotated schema defined by XML decomposition is presented next.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://www.ibm.com/xmlns/prod/db2/xdb1"
           targetNamespace="http://www.ibm.com/xmlns/prod/db2/xdb1"
           elementFormDefault="qualified" >
  <xs:element name="defaultSQLSchema" type="xs:string"/>
  <xs:attribute name="rowSet" type="xs:string"/>
  <xs:attribute name="column" type="xs:string"/>
  <xs:attribute name="locationPath" type="xs:string"/>
  <xs:attribute name="truncate" type="xs:boolean"/>
  <xs:attribute name="contentHandling">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="text"/>
        <xs:enumeration value="serializeSubtree"/>
        <xs:enumeration value="stringValue"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="normalization" >
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="original"/>
        <xs:enumeration value="whitespaceStrip"/>
        <xs:enumeration value="canonical"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="expression" type="xs:string"/>
  <xs:attribute name="condition" type="xs:string"/>
  <xs:element name="table">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="SQLSchema" type="xs:string" minOccurs="0"/>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="rowSet" type="xs:string"
                    maxOccurs="unbounded" form="qualified"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="rowSetMapping">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="rowSet" type="xs:string" />
        <xs:element name="column" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
        <xs:element name="expression" type="xs:string" minOccurs="0" />
        <xs:element name="condition" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute ref="truncate" />
    <xs:attribute ref="locationPath" />
    <xs:attribute ref="normalization" />
    <xs:attribute ref="contentHandling" />
</xs:complexType>
</xs:element>
</xs:schema>
```

Related concepts:

- “Annotated XML schema decomposition” on page 235
- “XML decomposition annotations - Overview” on page 244

Related reference:

- “XML decomposition annotations - Summary” on page 246

Chapter 11. Sample applications

Native XML Samples

The native XML data store support in DB2 enables well-formed XML documents to be stored in their hierarchical format within columns of a table. XML columns are defined with the new XML data type. Because the native XML data store is fully integrated into the DB2 database system, the stored XML data can be accessed and managed by leveraging DB2 functionality. This functionality includes administration support, application development support and efficient search and retrieval of XML via support for XQuery, SQL or a combination of SQL/XML functions.

There are various samples provided to demonstrate the XML support; these are broadly categorized as:

Administration samples

These samples demonstrate the following features:

- XML schema support : Schema registration and validation of XML documents
- XML data indexing support : Indexes on different node types of XML value
- Utility support for XML : Import , export, runstats, db2look, and db2batch support for the XML data type

Application Development samples

These samples demonstrate the following features:

- XML insert, update, and delete : Inserting XML values in XML typed columns, updating and deleting existing values
- XML parsing, validation, and serialization support : Implicit and explicit parsing of compatible data types, validating an XML document, serializing XML data
- Hybrid use of SQL and XQuery : Using SQL/XML functions like XMLTABLE, XMLQUERY and the XMLEXISTS predicate
- XML data type support in SQL and external procedures: Passing XML data to SQL and external procedures by including parameters of data type XML
- Annotated XML schema decomposition support : Decomposing an XML document based on annotated XML schemas
- XML publishing functions : Using functions to construct XML values

XQuery samples

These samples demonstrate the use of axes, FLWOR expressions, and queries written with XQuery and SQL/XML.

These samples can be found in the following location:

- On Windows: %DB2PATH%\sql1lib\samples\xml (where %DB2PATH% is a variable that determines where DB2 is installed)
- On UNIX: \$HOME/sql1lib/samples/xml (where \$HOME is the home directory of the instance owner)

Related concepts:

- “Sample files” in *Samples Topics*

Related reference:

- “Native XML - Administration samples” on page 326
- “Native XML - Application Development samples” on page 328
- “Native XML - Xquery samples” in *Samples Topics*

Native XML - Administration samples

These samples demonstrate native XML data store support for various administration features, including: XML schema support, Utility support, XML data indexing support.

These samples are available in various programming languages and can be found in the language specific sub-directories in the following location:

- On Windows: %DB2PATH%\sql11ib\samples\xml (where %DB2PATH% is a variable that determines where DB2 is installed)
- On UNIX: \$HOME/sql11ib/samples/xml (where \$HOME is the home directory of the instance owner)

Table 80. XML schema support - Samples for schema registration and validation of XML documents

Samples By Language	Sample program name	Program description
C	xmlschema.sqc	Register the XML schema to the database and use the registered schema to validate and insert an XML document.
CLP	xmlschema.db2	Register the XML schema to the database and use the registered schema to validate and insert an XML document.
JDBC	XmlSchema.java	Register the XML schema to the database and use the registered schema to validate and insert an XML document.
SQLJ	XmlSchema.sqlj	Register the XML schema to the database and use the registered schema to validate and insert an XML document.

Table 81. Utility support: Samples for Import, export, runstats, db2look, and db2batch support for XML data type

Samples By Language	Sample program name	Program description
C	xmlrunstats.sqc	Perform RUNSTATS on a table containing XML type columns
	lobstoxml.sqc	Move LOB data into an XML column using IMPORT and EXPORT commands
	impexpxml.sqc	Import and Export XML documents

Table 81. Utility support: Samples for Import, export, runstats, db2look, and db2batch support for XML data type (continued)

Samples By Language	Sample program name	Program description
CLP	xmlrunstats.db2	Perform RUNSTATS on a table containing XML type columns
	xmldb2batch.db2	db2batch support for XML datatype
	xmldb2look.db2	db2look support for XML datatype
	lobstoxml.db2	Move LOB data into an XML column using IMPORT and EXPORT commands
	impexpxml.db2	Import and Export XML documents
JDBC	XmlRunstats.java	Perform RUNSTATS on a table containing XML type columns

Table 82. XML data indexing support: Samples for indices over XML data

Samples By Language	Sample program name	Program description
C	xmlindex.sqc	Create an index and use it in an XQuery query
	xmlconst.sqc	Create an index with UNIQUE and VARCHAR length constraints using XML patterns
CLI	xmlindex.c	Create an index and use it in an XQuery query
	xmlconst.c	Create an index with UNIQUE and VARCHAR length constraints using XML patterns
CLP	xmlindex.db2	Create an index and use it in an XQuery query
	xmlconst.db2	Create an index with UNIQUE and VARCHAR length constraints using XML patterns
JDBC	XmlIndex.java	Create an index and use it in an XQuery query
	XmlConst.java	Create an index with UNIQUE and VARCHAR length constraints using XML patterns
SQLJ	XmlIndex.sqlj	Create an index and use it in an XQuery query
	XmlConst.sqlj	Create an index with UNIQUE and VARCHAR length constraints using XML patterns

Related concepts:

- “Native XML Samples” on page 325
- “Sample files” in *Samples Topics*

Related reference:

- “Native XML - C samples” in *Samples Topics*
- “Native XML - CLI samples” in *Samples Topics*

- “Native XML - CLP samples” in *Samples Topics*
- “Native XML - JDBC samples” in *Samples Topics*
- “Native XML - SQLJ samples” in *Samples Topics*

Native XML - Application Development samples

These samples demonstrate XML support for application development features such as insert, update, and delete, XML parsing, validation, and serialization, hybrid use of SQL/XML, XML data type support in SQL and external stored procedures, XML decomposition, SQL/XML publishing functions.

These samples are available in various programming languages and can be found in the language specific sub-directories in the following location:

- On Windows: %DB2PATH%\sql11ib\samples\xml (where %DB2PATH% is a variable that determines where DB2 is installed)
- On UNIX: \$HOME/sql11ib/samples/xml (where \$HOME is the home directory of the instance owner)

Table 83. Native XML - Application development samples

Samples by language	Sample program name	Program description
CLI	xmlinsert.c	Insert an XML document into a column of XML datatype.
	xmlupdel.c	Update and delete XML documents in the tables.
	xmlread.c	Read XML data stored in tables.
	reltoxml.doc.c	Create an XML document directly from data stored in relational tables using various SQL/XML publishing functions.
	xmltotable.c	Insert the data from an XML document into relational tables using SQL/XML functions like XMLTABLE, XMLQUERY and the XMLEXISTS predicate.
	simple_xmlproc.c	Simple stored procedure with XML type parameters
	simple_xmlproc_client.c	Client program to call the routine in simple_xmlproc.c.
	simple_xmlproc_create.db2	CLP script to register the stored procedure in simple_xmlproc.c
	simple_xmlproc_drop.db2	CLP script to drop the stored procedure in simple_xmlproc.c

Table 83. Native XML - Application development samples (continued)

Samples by language	Sample program name	Program description
C	xmlinsert.sqc	Insert an XML document into a column of XML datatype.
	xmlupdel.sqc	Update and delete XML documents in the tables.
	xmlread.sqc	Read XML data stored in tables.
	reltoxmltype.sqc	Create an XML object from data stored in relational tables using various SQL/XML publishing functions.
	xmldecomposition.sqc	Decompose data stored in an XML file and insert the data into tables.
	simple_xmlproc.sqc	Simple stored procedure with XML type parameters
	simple_xmlproc_client.db2	CLP script to call the routine in simple_xmlproc.sqc
	simple_xmlproc_create.db2	CLP script to register the stored procedure in simple_xmlproc.sqc
	simple_xmlproc_drop.db2	CLP script to drop the stored procedure in simple_xmlproc.sqc
CLP	xmlinsert.db2	Insert an XML document into a column of XML datatype.
	xmlupdel.db2	Update and delete XML documents in the tables.
	reltoxml.doc.db2	Create an XML document directly from data stored in relational tables using various SQL/XML publishing functions.
	reltoxmltype.db2	Create an XML object from data stored in relational tables using various SQL/XML publishing functions
	xmldecomposition.db2	Decompose data stored in an XML file and insert the data into tables.
	simple_xmlproc.db2	Simple stored procedure with XML type parameters
	xmltotable.db2	Insert the data from an XML document into relational tables using SQL/XML functions like XMLTABLE, XMLQUERY and the XMLEXISTS predicate.

Table 83. Native XML - Application development samples (continued)

Samples by language	Sample program name	Program description
JDBC	XmlInsert.java	Insert an XML document into a column of XML datatype.
	XmlUpDel.java	Update and delete XML documents in the tables.
	XmlRead.java	Read XML data stored in tables.
	RelToXmlDoc.java	Create an XML document directly from data stored in relational tables using SQL/XML publishing functions
	RelToXmlType.java	Create an XML object from data stored in relational tables using various SQL/XML publishing functions.
	XmlDecomposition.java	Decompose data stored in an XML file and insert the data into tables.
	Simple_XmlProc.java	Simple stored procedure with XML type parameters
	Simple_XmlProc_Client.java	Client program to call the routine in Simple_XmlProc.java.
	Simple_XmlProc_Create.db2	CLP script to register the stored procedure in Simple_XmlProc.java
	Simple_XmlProc_Drop.db2	CLP script to drop the stored procedure in Simple_XmlProc.java
	XmlToTable.java	Insert the data from an XML document into relational tables using SQL/XML functions like XMLTABLE, XMLQUERY and the XMLEXISTS predicate.
SQLJ	XmlInsert.sqlj	Insert an XML document into a column of XML datatype.
	XmlUpDel.sqlj	Update and delete XML documents in the tables.
	XmlRead.sqlj	Read XML data stored in tables.
	RelToXmlDoc.sqlj	Create an XML document directly from data stored in relational tables using SQL/XML publishing functions
	RelToXmlType.sqlj	Create an XML object from data stored in relational tables using SQL/XML publishing functions.
	XmlToTable.sqlj	Insert the data from an XML document into relational tables using SQL/XML functions like XMLTABLE, XMLQUERY and the XMLEXISTS predicate.

Related concepts:

- “Native XML Samples” on page 325
- “Sample files” in *Samples Topics*

Related reference:

- “Native XML - C samples” in *Samples Topics*
- “Native XML - CLI samples” in *Samples Topics*
- “Native XML - CLP samples” in *Samples Topics*
- “Native XML - JDBC samples” in *Samples Topics*
- “Native XML - SQLJ samples” in *Samples Topics*

Appendix A. Encoding mappings

Mappings of encoding names to effective CCSIDs for stored XML data

If data that you store in an XML column is in a binary application variable, or is an internally encoded XML type, the DB2 database manager examines the data to determine the encoding. If the data has an encoding declaration, the database manager maps the encoding name to a CCSID. Table 84 lists these mappings. If an encoding name is not in Table 84, the database manager returns an error.

The normalized encoding name in the first column of Table 84 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 84. 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 84. 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
CP1140	1140
CP1141	1141
CP1142	1142
CP1143	1143
CP1144	1144
CP1145	1145

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
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
CP424	424
CP437	437
CP500	500
CP5346	5346
CP5347	5347
CP5348	5348
CP5349	5349
CP5350	5350
CP5353	5353
CP813	813
CP819	819
CP838	838

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
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
CP943	943
CP943C	943
CP949	970
CP950	950
CP964	964
CP970	970
CPGR	869
CSASCII	367
CSBIG5	950
CSEBCDICAFR	500
CSEBCDICDKNO	277
CSEBCDICES	284
CSEBCDIFISE	278
CSEBCDICFR	297
CSEBCDIT	280
CSEBCDICPT	37
CSEBCDICUK	285
CSEBCDICUS	37

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
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
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

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
CSISOLATINARABIC	1089
CSISOLATINCYRILLIC	915
CSISOLATINGREEK	813
CSISOLATINHEBREW	62210
CSKOI8R	878
CSKSC56011987	970
CSMACINTOSH	1275
CSMICROSOFTPUBLISHING	1004
CSPC850MULTILINGUAL	850
CSPC862LATINHEBREW	862
CSPC8CODEPAGE437	437
CSPCP852	852
CSSHIFTJIS	943
CSUCS4	1236
CSUNICODE11	1204
CSUNICODE	1204
CSUNICODEASCII	1204
CSUNICODELATIN1	1204
CSVISCI	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
EBCDICPPFR	297
EBCDICCPGB	285
EBCDICCPGR	423
EBCDICPPHE	424
EBCDICPPIS	871
EBCDICPPIT	280
EBCDICPNL	37
EBCDICPPNO	277
EBCDICPROECE	870

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
EBCDICPSE	278
EBCDICPUS	37
EBCDICPWT	37
EBCDICPYU	870
EBCDICDE273EURO	1141
EBCDICDK277EURO	1142
EBCDICDKNO	277
EBCDICES284EURO	1145
EBCDICES	284
EBCDICFI278EURO	1143
EBCDICFISE	278
EBCDICFR297EURO	1147
EBCDICFR	297
EBCDICGB285EURO	1146
EBCDICINTERNATIONAL500EURO	1148
EBCDICIS871EURO	1149
EBCDIT280EURO	1144
EBCDIT	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
HEBREW	62210
HPROMAN8	1051

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
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
IBM500	500
IBM808	808
IBM813	813
IBM819	819
IBM850	850
IBM852	852
IBM855	855
IBM857	857

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
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
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

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
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
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

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
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
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	1250
WINDOWS1251	1251
WINDOWS1252	1252
WINDOWS1253	1253
WINDOWS1254	1254
WINDOWS1255	1255
WINDOWS1256	1256

Table 84. Encoding names and effective CCSIDs (continued)

Normalized encoding name	CCSID
WINDOWS1257	1257
WINDOWS1258	1258
WINDOWS28598	62210
WINDOWS31J	943
WINDOWS936	1386
XEUCTW	964
XMSWIN936	1386
XUTF16BE	1200
XUTF16LE	1202
XWINDOWS949	970

Related concepts:

- “XML data encoding” on page 217
- “XML serialization” on page 86

Related reference:

- “XMLSERIALIZE scalar function” in *SQL Reference, Volume 1*

Mappings of CCSIDs to encoding names for serialized XML output data

As part of an implicit or explicit XMLSERIALIZE operation, the DB2 database manager might add an encoding declaration at the beginning of serialized XML output data. That 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, the DB2 database manager chooses the IANA registry name for the CCSID, as prescribed by the XML standard.

Table 85. CCSIDs and corresponding encoding names

CCSID	Encoding name
37	IBM037
273	IBM273
277	IBM277
278	IBM278

Table 85. CCSIDs and corresponding encoding names (continued)

CCSID	Encoding name
280	IBM280
284	IBM284
285	IBM285
297	IBM297
367	US-ASCII
420	IBM420
423	IBM423
424	IBM424
437	IBM437
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	IBM921
922	IBM922
923	ISO-8859-15

Table 85. CCSIDs and corresponding encoding names (continued)

CCSID	Encoding name
924	IBM00924
932	Shift_JIS
943	Windows-31J
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-16BE
1202	UTF-16LE
1204	UTF-16
1208	UTF-8
1232	UTF-32BE
1234	UTF-32LE
1236	UTF-32
1250	windows-1250

Table 85. CCSIDs and corresponding encoding names (continued)

CCSID	Encoding name
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
13488	UTF-16BE
13490	UTF-16LE
16840	IBM420
17248	IBM864
17584	UTF-16BE
17586	UTF-16LE
62209	IBM862
62210	ISO-8859-8
62211	IBM424

Table 85. CCSIDs and corresponding encoding names (continued)

CCSID	Encoding name
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:

- “Background information on XML internal encoding” on page 217
- “XML data encoding” on page 217

Appendix B. DB2 Database technical information

Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
 - Topics
 - Help for DB2 tools
 - Sample programs
 - Tutorials
- DB2 books
 - PDF files (downloadable)
 - PDF files (from the DB2 PDF CD)
 - printed books
- Command line help
 - Command help
 - Message help
- Sample programs

IBM periodically makes documentation updates available. If you access the online version on the DB2 Information Center at ibm.com[®], you do not need to install documentation updates because this version is kept up-to-date by IBM. If you have installed the DB2 Information Center, it is recommended that you install the documentation updates. Documentation updates allow you to update the information that you installed from the *DB2 Information Center CD* or downloaded from Passport Advantage as new information becomes available.

Note: The DB2 Information Center topics are updated more frequently than either the PDF or the hard-copy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and Redbooks™ online at ibm.com. Access the DB2 Information Management software library site at <http://www.ibm.com/software/data/sw-library/>.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how we can improve the DB2 documentation, send an e-mail to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this e-mail address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

Related concepts:

- “Features of the DB2 Information Center” in *Online DB2 Information Center*
- “Sample files” in *Samples Topics*

Related tasks:

- “Invoking command help from the command line processor” in *Command Reference*
- “Invoking message help from the command line processor” in *Command Reference*
- “Updating the DB2 Information Center installed on your computer or intranet server” on page 355

Related reference:

- “DB2 technical library in hardcopy or PDF format” on page 350

DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/shop/publications/order. DB2 Version 9 manuals in PDF format can be downloaded from www.ibm.com/software/data/db2/udb/support/manualsv9.html.

Although the tables identify books available in print, the books might not be available in your country or region.

The information in these books is fundamental to all DB2 users; you will find this information useful whether you are a programmer, a database administrator, or someone who works with DB2 Connect™ or other DB2 products.

Table 86. DB2 technical information

Name	Form Number	Available in print
<i>Administration Guide: Implementation</i>	SC10-4221	Yes
<i>Administration Guide: Planning</i>	SC10-4223	Yes
<i>Administrative API Reference</i>	SC10-4231	Yes
<i>Administrative SQL Routines and Views</i>	SC10-4293	No
<i>Call Level Interface Guide and Reference, Volume 1</i>	SC10-4224	Yes
<i>Call Level Interface Guide and Reference, Volume 2</i>	SC10-4225	Yes
<i>Command Reference</i>	SC10-4226	No
<i>Data Movement Utilities Guide and Reference</i>	SC10-4227	Yes
<i>Data Recovery and High Availability Guide and Reference</i>	SC10-4228	Yes
<i>Developing ADO.NET and OLE DB Applications</i>	SC10-4230	Yes
<i>Developing Embedded SQL Applications</i>	SC10-4232	Yes

Table 86. DB2 technical information (continued)

Name	Form Number	Available in print
<i>Developing SQL and External Routines</i>	SC10-4373	No
<i>Developing Java Applications</i>	SC10-4233	Yes
<i>Developing Perl and PHP Applications</i>	SC10-4234	No
<i>Getting Started with Database Application Development</i>	SC10-4252	Yes
<i>Getting started with DB2 installation and administration on Linux and Windows</i>	GC10-4247	Yes
<i>Message Reference Volume 1</i>	SC10-4238	No
<i>Message Reference Volume 2</i>	SC10-4239	No
<i>Migration Guide</i>	GC10-4237	Yes
<i>Net Search Extender Administration and User's Guide</i> Note: HTML for this document is not installed from the HTML documentation CD.	SH12-6842	Yes
<i>Performance Guide</i>	SC10-4222	Yes
<i>Query Patroller Administration and User's Guide</i>	GC10-4241	Yes
<i>Quick Beginnings for DB2 Clients</i>	GC10-4242	No
<i>Quick Beginnings for DB2 Servers</i>	GC10-4246	Yes
<i>Spatial Extender and Geodetic Data Management Feature User's Guide and Reference</i>	SC18-9749	Yes
<i>SQL Guide</i>	SC10-4248	Yes
<i>SQL Reference, Volume 1</i>	SC10-4249	Yes
<i>SQL Reference, Volume 2</i>	SC10-4250	Yes
<i>System Monitor Guide and Reference</i>	SC10-4251	Yes
<i>Troubleshooting Guide</i>	GC10-4240	No
<i>Visual Explain Tutorial</i>	SC10-4319	No
<i>What's New</i>	SC10-4253	Yes
<i>XML Extender Administration and Programming</i>	SC18-9750	Yes
<i>XML Guide</i>	SC10-4254	Yes
<i>XQuery Reference</i>	SC18-9796	Yes

Table 87. DB2 Connect-specific technical information

Name	Form Number	Available in print
<i>DB2 Connect User's Guide</i>	SC10-4229	Yes

Table 87. DB2 Connect-specific technical information (continued)

Name	Form Number	Available in print
Quick Beginnings for DB2 Connect Personal Edition	GC10-4244	Yes
Quick Beginnings for DB2 Connect Servers	GC10-4243	Yes

Table 88. WebSphere® Information Integration technical information

Name	Form Number	Available in print
WebSphere Information Integration: Administration Guide for Federated Systems	SC19-1020	Yes
WebSphere Information Integration: ASNCLP Program Reference for Replication and Event Publishing	SC19-1018	Yes
WebSphere Information Integration: Configuration Guide for Federated Data Sources	SC19-1034	No
WebSphere Information Integration: SQL Replication Guide and Reference	SC19-1030	Yes

Note: The DB2 Release Notes provide additional information specific to your product's release and fix pack level. For more information, see the related links.

Related concepts:

- "Overview of the DB2 technical information" on page 349
- "About the Release Notes" in *Release notes*

Related tasks:

- "Ordering printed DB2 books" on page 352

Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the *DB2 PDF Documentation* CD are unavailable in print. For example, neither volume of the *DB2 Message Reference* is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation CD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation CD are available in print.

Note: The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at <http://publib.boulder.ibm.com/infocenter/db2help/>.

Procedure:

To order printed DB2 books:

- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at <http://www.ibm.com/shop/publications/order>. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.
- To order printed DB2 books from your local IBM representative:
 - Locate the contact information for your local representative from one of the following Web sites:
 - The IBM directory of world wide contacts at www.ibm.com/planetwide
 - The IBM Publications Web site at <http://www.ibm.com/shop/publications/order>. You will need to select your country, region, or language to the access appropriate publications home page for your location. From this page, follow the "About this site" link.
 - When you call, specify that you want to order a DB2 publication.
 - Provide your representative with the titles and form numbers of the books that you want to order.

Related concepts:

- "Overview of the DB2 technical information" on page 349

Related reference:

- "DB2 technical library in hardcopy or PDF format" on page 350

Displaying SQL state help from the command line processor

DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure:

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, ? 08003 displays help for the 08003 SQL state, and ? 08 displays help for the 08 class code.

Related tasks:

- "Invoking command help from the command line processor" in *Command Reference*
- "Invoking message help from the command line processor" in *Command Reference*

Accessing different versions of the DB2 Information Center

For DB2 Version 9 topics, the DB2 Information Center URL is <http://publib.boulder.ibm.com/infocenter/db2luw/v9/>.

For DB2 Version 8 topics, go to the Version 8 Information Center URL at: <http://publib.boulder.ibm.com/infocenter/db2luw/v8/>.

Related tasks:

- “Setting up access to DB2 contextual help and documentation” in *Administration Guide: Implementation*

Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

Procedure:

To display topics in your preferred language in the Internet Explorer browser:

1. In Internet Explorer, click the **Tools** → **Internet Options** → **Languages...** button. The Language Preferences window opens.
2. Ensure your preferred language is specified as the first entry in the list of languages.
 - To add a new language to the list, click the **Add...** button.

Note: Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.

- To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

To display topics in your preferred language in a Firefox or Mozilla browser:

1. Select the **Tools** → **Options** → **Languages** button. The Languages panel is displayed in the Preferences window.
2. Ensure your preferred language is specified as the first entry in the list of languages.
 - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
 - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you might have to also change the regional settings of your operating system to the locale and language of your choice.

Related concepts:

- “Overview of the DB2 technical information” on page 349

Updating the DB2 Information Center installed on your computer or intranet server

If you have a locally-installed DB2 Information Center, updated topics can be available for download. The 'Last updated' value found at the bottom of most topics indicates the current level for that topic.

To determine if there is an update available for the entire DB2 Information Center, look for the 'Last updated' value on the Information Center home page. Compare the value in your locally installed home page to the date of the most recent downloadable update at <http://www.ibm.com/software/data/db2/udb/support/icupdate.html>. You can then update your locally-installed Information Center if a more recent downloadable update is available.

Updating your locally-installed DB2 Information Center requires that you:

1. Stop the DB2 Information Center on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to download and apply updates.
2. Use the Update feature to determine if update packages are available from IBM.

Note: Updates are also available on CD. For details on how to configure your Information Center to install updates from CD, see the related links. If update packages are available, use the Update feature to download the packages. (The Update feature is only available in stand-alone mode.)

3. Stop the stand-alone Information Center, and restart the DB2 Information Center service on your computer.

Procedure:

To update the DB2 Information Center installed on your computer or intranet server:

1. Stop the DB2 Information Center service.
 - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click on **DB2 Information Center** service and select **Stop**.
 - On Linux, enter the following command:
`/etc/init.d/db2icdv9 stop`
2. Start the Information Center in stand-alone mode.
 - On Windows:
 - a. Open a command window.
 - b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `C:\Program Files\IBM\DB2 Information Center\Version 9` directory.
 - c. Run the `help_start.bat` file using the fully qualified path for the DB2 Information Center:
`<DB2 Information Center dir>\doc\bin\help_start.bat`
 - On Linux:

- a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the /opt/ibm/db2ic/V9 directory.
- b. Run the help_start script using the fully qualified path for the DB2 Information Center:

```
<DB2 Information Center dir>/doc/bin/help_start
```

The systems default Web browser launches to display the stand-alone Information Center.

3. Click the Update button (🔄). On the right hand panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
4. To initiate the download process, check the selections you want to download, then click **Install Updates**.
5. After the download and installation process has completed, click **Finish**.
6. Stop the stand-alone Information Center.
 - On Windows, run the help_end.bat file using the fully qualified path for the DB2 Information Center:

```
<DB2 Information Center dir>\doc\bin\help_end.bat
```

Note: The help_end batch file contains the commands required to safely terminate the processes that were started with the help_start batch file. Do not use Ctrl-C or any other method to terminate help_start.bat.
 - On Linux, run the help_end script using the fully qualified path for the DB2 Information Center:

```
<DB2 Information Center dir>/doc/bin/help_end
```

Note: The help_end script contains the commands required to safely terminate the processes that were started with the help_start script. Do not use any other method to terminate the help_start script.
7. Restart the DB2 Information Center service.
 - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click on **DB2 Information Center** service and select **Start**.
 - On Linux, enter the following command:

```
/etc/init.d/db2icdv9 start
```

The updated DB2 Information Center displays the new and updated topics.

Related concepts:

- “DB2 Information Center installation options” in *Quick Beginnings for DB2 Servers*

Related tasks:

- “Installing the DB2 Information Center using the DB2 Setup wizard (Linux)” in *Quick Beginnings for DB2 Servers*
- “Installing the DB2 Information Center using the DB2 Setup wizard (Windows)” in *Quick Beginnings for DB2 Servers*

DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

Before you begin:

You can view the XHTML version of the tutorial from the Information Center at <http://publib.boulder.ibm.com/infocenter/db2help/>.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

DB2 tutorials:

To view the tutorial, click on the title.

Native XML data store

Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

Visual Explain Tutorial

Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

Related concepts:

- “Visual Explain overview” in *Administration Guide: Implementation*

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 products.

DB2 documentation

Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Support and Troubleshooting section of the DB2 Information Center. There you will find information on how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 products.

DB2 Technical Support Web site

Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the DB2 Technical Support Web site at <http://www.ibm.com/software/data/db2/udb/support.html>

Related concepts:

- “Introduction to problem determination” in *Troubleshooting Guide*
- “Overview of the DB2 technical information” on page 349

Terms and Conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal use: You may reproduce these Publications for your personal, non commercial 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.

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.

Appendix C. Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs, in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks

Company, product, or service names identified in the documents of the DB2 Version 9 documentation library may be trademarks or service marks of International Business Machines Corporation or other companies. Information on the trademarks of IBM Corporation in the United States, other countries, or both is located at <http://www.ibm.com/legal/copytrade.shtml>.

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 documentation library:

Microsoft, Windows, Windows NT[®], and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel[®], Itanium[®], Pentium[®], and Xeon[®] are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

- ADD XMLSCHEMA DOCUMENT
 - command
 - syntax 149
- annotated XML schema
 - decomposition 235
 - advantage 236
 - annotations
 - considerations 291
 - db2-xdb:column 257
 - db2-xdb:condition 266
 - db2-xdb:contentHandling 269
 - db2-xdb:defaultSQLSchema 247
 - db2-xdb:expression 263
 - db2-xdb:locationPath 259
 - db2-xdb:normalization 274
 - db2-xdb:rowSet 249
 - db2-xdb:rowSetMapping 280
 - db2-xdb:table 254
 - db2-xdb:truncate 278
 - overview 244
 - schema 322
 - specification 244
 - summary 246
 - CDATA sections 287
 - data type compatibility 307
 - character types 308
 - datetime types 310
 - graphic types 311
 - LOB and for bit data types 313
 - numeric types 315
 - decomposing documents 236
 - registering schemas 237
 - derived complex types 289
 - disablement 239
 - empty strings 287
 - enabling schemas 237
 - examples 296, 298, 300, 302, 303, 305
 - keywords 284
 - NULL values 287
 - registering schemas 237
 - restrictions 317
 - results 285
 - rowSets 294
 - schema structuring 292
 - troubleshooting 319
 - validation 286
 - xdbDecompXML stored
 - procedures 240
- archiving XML 89
- ASC clause
 - CREATE INDEX statement 108
- atomic values 10
- attribute nodes 14
- auxiliary storage objects
 - XML data specifier 158

B

- byte order mark (BOM)
 - Unicode 217

C

- casting
 - between data types 56
 - reference types 56
 - user-defined types 56
 - XML values
 - XMLQUERY example 55
- CDATA in decomposition 287
- CLI (call level interface)
 - SQL/XML functions 167
 - XML data 167
 - changing default type 170
 - inserts 168
 - retrieval 169
 - updates 168
 - XQuery expressions 167
- CLI applications
 - XML data 167
- CLI/ODBC keywords
 - MapXMLCDefault 170
 - MapXMLDescribe 170
- CLOSE in CREATE INDEX statement 108
- CLUSTER clause
 - CREATE INDEX statement 108
- columns
 - creating index keys 108
- command line processor
 - registering an XSR object 136
- command line processor (CLP)
 - XML support 90
- commands
 - DECOMPOSE XML DOCUMENT 242
- comment nodes 15
- compatibility
 - data types
 - for decomposition 307
- COMPLETE XMLSCHEMA command
 - syntax 150
- constructing XML 81
 - special character handling 85
- contacting IBM 365
- Control Center
 - XML support 90
- COPY
 - CREATE INDEX statement 108
- CREATE INDEX statement
 - column-names in index keys 108
 - description 108
 - index over XML data
 - examples 124
 - XML column 108
- cursors
 - XQuery 191

D

- data model
 - XQuery and XPath 9
- data types
 - casting between 56
 - XML 17
 - compatibility for
 - decomposition 307
- database managed space (DMS)
 - used to improve performance 131
- DB2 .NET Data Provider
 - XML support 90
- DB2 Developer Workbench
 - XML support 90
- DB2 Information Center
 - updating 355
 - versions 354
 - viewing in different languages 354
- DB2 Net Search Extender (NSE)
 - full-text search of XML data 79
- DB2 XQuery functions
 - sqlquery 47
 - xmlcolumn 46
- DB2 XQuery, overview 45
- DDL statements
 - altering XSR objects 139
- debugging
 - XML decomposition 319
- DECOMP_CONTENT keyword 284
- DECOMP_DOCUMENTID keyword 284
- DECOMP_ELEMENTID keyword 284
- DECOMPOSE XML DOCUMENT
 - command
 - description 242
- decomposing XML 235, 236
 - annotations
 - considerations 291
 - db2-xdb:column 257
 - db2-xdb:condition 266
 - db2-xdb:contentHandling 269
 - db2-xdb:defaultSQLSchema 247
 - db2-xdb:expression 263
 - db2-xdb:locationPath 259
 - db2-xdb:normalization 274
 - db2-xdb:rowSet 249
 - db2-xdb:rowSetMapping 280
 - db2-xdb:table 254
 - db2-xdb:truncate 278
 - keywords 284
 - overview 244
 - schema 322
 - specification 244
 - summary 246
 - CDATA sections 287
 - data type compatibility 307
 - character types 308
 - datetime types 310
 - graphic types 311
 - LOB and for bit data types 313
 - numeric types 315
 - derived complex types 289

- decomposing XML (*continued*)
 - disabling 239
 - empty strings 287
 - enabling schemas 237
 - examples 296, 298, 300, 302, 303, 305
 - NULL values 287
 - registering schemas 237
 - restrictions 317
 - results 285
 - rowSets 294
 - schema structuring 292
 - troubleshooting 319
 - validation 286
 - xdbDecompXML stored procedures 240
- decomposition
 - annotated XML schema advantage 236
- DEFER
 - in CREATE INDEX statement 108
- deleting
 - XML 36
- deregisterDB2XMLObject 137
- DESC clause
 - CREATE INDEX statement 108
- DMS (database managed space)
 - used to improve performance 131
- document nodes
 - description 13
- document order 15
- documentation 349, 350
 - terms and conditions of use 358
- down-level clients
 - retrieval of data from XML columns 80

E

- element nodes 13
- embedded SQL
 - Input and output variables 177
- empty string
 - in decomposition 287
- enabling
 - XML schemas for decomposition 237
- encoding
 - declarations
 - CCSID mappings 344
 - mappings to CCSIDs 333
 - XML data 217
 - background on internal encoding 217
 - CCSIDs to encoding names 344
 - encoding scenarios 226
 - in JDBC, SQLJ and .NET 221
 - input considerations 219
 - names to CCSID mappings 333
 - retrieval 219
 - routine parameter
 - considerations 220
 - scenarios 222, 224, 228
- example
 - deregisterDB2XMLObject 137
 - registerDB2XMLSchema 137
- examples
 - XML decomposition 296, 298, 300, 302, 303, 305

- EXPLAIN
 - XML support 90
- explicit XML parsing 37
- exporting
 - XML data 159
- EXTEND USING clause
 - CREATE INDEX statement 108
- external XML encoding
 - considerations
 - for JDBC, SQLJ, and .NET 221
 - input of XML 219
 - scenarios
 - input 224

F

- FREEPAGE in CREATE INDEX statement 108
- full-text search of XML data 79

G

- GBPCACHE
 - in CREATE INDEX statement 108

H

- help
 - displaying 354
 - for SQL statements 353
- hierarchy, nodes 15

I

- identity of nodes 15
- ignorable whitespace
 - in validation 41
- implicit XML parsing 37
- importing
 - XML data 162
- INCLUDE clause
 - CREATE INDEX statement 108
- index
 - Casting rules for join predicates 77
 - over XML data
 - best practices overview 72
 - common issues 127
 - complex data type 99
 - data type 97
 - data type conversion 102
 - data type of literals 76
 - overview 93
 - recreation 127
 - restrictions 126
 - restrictiveness 74
 - specifying text() 75
 - UNIQUE keyword semantics 105
 - XML namespace declaration 96
 - XML schemas 104
 - XQuery pattern expressions 94
 - XMLEXISTS predicate usage 67
- INDEX clause
 - CREATE INDEX statement 108

- index keys
 - indexes over XML data
 - XQuery pattern expressions 94
- index over XML data
 - best practices
 - overview 72
 - Casting rules for join predicates 77
 - common issues 127
 - CREATE INDEX statement 108
 - examples 124
 - data type
 - complex 99
 - conversion 102
 - of literals 76
 - of XQuery pattern expressions 97
 - database objects associated with 106, 107
 - indeterminate query evaluation 79
 - index keys
 - XQuery pattern expressions 94
 - Logical and physical indexes 106
 - recreation 127
 - restrictions 126
 - restrictiveness 74
 - specifying text() 75
 - Syntax and option descriptions 108
 - UNIQUE keyword semantics 105
 - XML namespace declaration 96
 - XML schemas 104
 - XMLEXISTS predicate usage 67
- Information Center
 - updating 355
 - versions 354
 - viewing in different languages 354
- inserting data
 - XML 32
 - CLI applications 168
- internal XML encoding
 - considerations
 - for JDBC, SQLJ, and .NET 221
 - input of XML 219
 - scenarios
 - input 222
- items in sequences 9

J

- JDBC
 - XML data
 - encoding 221
- JDBC application
 - XML column updates 178
 - XML data 177
 - XML data retrieval 180

L

- LOB (large object) data types
 - importing and exporting 157

M

- managing
 - XML result sets 70

moving data
 considerations for moving XML
 data 156

N

native XML data store 1
 adding XML columns 32
 administration
 samples 326
 constructing XML 81
 deleting XML 36
 encoding 217
 full-text search 79
 inserting XML 32
 parsing 37
 publishing XML 81
 special character handling 85
 querying 45
 comparison of methods 49
 with SQL 48
 XMLEXISTS predicate 66
 XMLQUERY overview 51
 XMLTABLE overview 61
 restrictions 233
 sample programs 325
 samples 326, 328
 serialization 86
 SQL/XML functions
 publishing 81
 XMLQUERY overview 51
 XMLTABLE overview 61
 table creation 31
 tools support 90
 triggers 37
 tutorial 18
 validation 41
 XML documents
 XML schema retrieval 140
 XML schema repository 133
 XML schemas
 component retrieval 140
 listing all registered 139
 XSR objects 133
 registration 134
nodes
 attribute 14
 comment
 description 15
 document
 description 13
 duplicate 15
 element 13
 hierarchy 15
 identity 15
 overview 10, 13
 processing instruction
 description 15
 properties 12
 string values 16
 text
 description 14
 typed values 16
notices 359
NULL value
 decomposition 287

O

objects
 associated with XML columns 107
ON clause
 CREATE INDEX statement 108
ordering DB2 books 352
overview
 XML data movement 155

P

parsing
 explicit 37
 CLI applications 168
 implicit 37
 CLI applications 168
PCTFREE clause
 CREATE INDEX statement 108
performance
 routines 208
PIECESIZE, in CREATE INDEX
 statement 108
printed books
 ordering 352
problem determination
 online information 357
 tutorials 357
 XML decomposition 319
procedures
 effects of commits and rollbacks on
 XML parameters and variables 193
 XML parameters 189
 XML variables 189
processing instruction nodes
 description 15
programming languages
 supported
 for XML 165
publishing XML 81
 special character handling 85

Q

queries
 index over XML data
 indeterminate query
 evaluation 79
 structure 46
query languages
 XML data 49
query performance
 impact of system managed space 131
querying XML 45
 comparison of methods 49
 with SQL 48
 constant and parameter marker
 passing 69
 XMLEXISTS 66
 XMLQUERY 51
 XMLTABLE 61

R

reference types
 casting 56

REGISTER XMLSCHEMA command
 syntax 147
REGISTER XSROBJECT command
 syntax 151
registerDB2XMLSchema 137
registering
 XML schemas for decomposition 237
restrictions
 native XML data store 233
result sets
 XML 70
retrieving data
 XML 45
 CLI applications 169
 encoding considerations 219
 encoding scenarios 226, 228
routines
 C/C++
 xml data type support 194
 COBOL
 xml data type support 194
 common language runtime
 xml data type support 194
 external
 xml data type support 194
 invocation from Java programs
 XML parameters 182
 Java
 xml data type support 194
 performance 208
 XML support
 encoding considerations 220
rows
 index keys with UNIQUE clause 108
 indexes 108
rowSets in decomposition 294

S

schema
 repository 133
sequences
 description 9
serialization
 CCSID to encoding name
 mappings 344
 differences in XML document 88
 explicit 86
 CLI applications 169
 implicit 86
 CLI applications 167, 169
shredding XML 235, 236
SQL statements
 CREATE INDEX 108
 displaying help 353
SQL/XML
 CREATE INDEX statement 108
 functions
 XMLQUERY overview 51
 XMLTABLE overview 61
SQLJ
 XML data
 encoding 221
SQLJ application
 XML column updates 184
 XML data 183
 XML data retrieval 186

- sqlquery function 47
- storage
 - native XML data store 1
 - XML data specifier 158
- storage requirements
 - XML documents 29
- stored procedures
 - registering an XSR object 135
 - XSR_COMPLETE 143
 - XSR_DTD 144
 - XSR_ENTITY 145
 - XSR_REGISTER 141
- storing XML data
 - encoding 217
 - considerations 219
 - name to CCSID mappings 333
 - inserting 32
 - native XML data store 1
 - updating 34
- string values of nodes 16

T

- tables
 - creating
 - with XML columns 31
 - indexes 108
- terms and conditions
 - use of publications 358
- text nodes
 - description 14
- text search
 - full-text search of XML data 79
- triggers
 - XML support 37
- troubleshooting
 - online information 357
 - tutorials 357
 - XML decomposition 319
- tutorials
 - native XML data store 18
 - troubleshooting and problem determination 357
 - Visual Explain 357
- type 2 indexes 108
- typed values of nodes 16

U

- UNIQUE clause
 - CREATE INDEX statement 108
- updates
 - DB2 Information Center 355
 - Information Center 355
 - of XML columns 34
- updating
 - XML columns 34
- user-defined types (UDTs)
 - casting 56
- USING clause
 - CREATE INDEX statement 108

V

- validating
 - XML data 41

- validating (*continued*)
 - decomposition 286
- values, atomic 10
- Visual Explain
 - tutorial 357
 - XML support 90

W

- whitespace
 - boundary 37
 - in validation 41
 - parsing 37

X

- xdbDecompXML stored procedures 240
- XML
 - adding XML columns 32
 - application development 165
 - samples 328
 - archival data types 89
 - C/C++ applications
 - Executing XQuery expressions 174
 - COBOL applications
 - Executing XQuery expressions 174
 - constructing 81
 - special character handling 85
 - CREATE INDEX statement 108
 - data encoding 171
 - data retrieval 45
 - C applications 173
 - COBOL applications 173
 - comparison of methods 49
 - document differences 88
 - result sets 70
 - data type 17, 171
 - declarations 171
 - CLI applications 167
 - decomposition 235
 - advantage 236
 - annotations 244, 246, 247, 249, 254, 257, 259, 263, 266, 269, 274, 278, 280, 291, 322
 - CDATA sections 287
 - data type compatibility 307
 - decomposing documents 236
 - derived complex types 289
 - disablement 239
 - empty strings 287
 - enabling schemas 237
 - examples 296, 298, 300, 302, 303, 305
 - keywords 284
 - limits 317
 - NULL values 287
 - registering schemas 237
 - restrictions 317
 - results 285
 - rowSets 294
 - schema structuring 292
 - troubleshooting 319
 - validation 286

- XML (*continued*)
 - decomposition (*continued*)
 - xdbDecompXML stored procedures 240
 - deleting 36
 - encoding 217
 - internal 217
 - inserting 32
 - model comparison 7
 - native XML data store 1
 - native XML tutorial 18
 - parameters in procedures 189
 - parsing 37
 - programming language support 165
 - publishing 81
 - special character handling 85
 - querying with SQL 48
 - constant and parameter marker passing 69
 - XMLEXISTS predicate 66
 - XMLQUERY 51, 52, 53
 - XMLTABLE 61, 63, 65
 - serialization 86
 - storage
 - document differences 88
 - encoding name to CCSID mappings 333
 - support in triggers 37
 - tables
 - creation for native XML data 31
 - tools support 90
 - updating columns 34
 - validation 41
 - variables in procedures 189
 - XMLQUERY function 174, 176
 - XQuery expressions 174, 176
- XML columns
 - adding 32
 - CREATE INDEX statement 108
 - data type 17
 - defining 31
 - indexing
 - overview 93
 - inserting into 32
 - updates
 - examples 34
 - JDBC applications 178
 - SQLJ applications 184
- XML data
 - CLI applications 167
 - inserting 168
 - retrieval 169
 - updating 168
 - considerations for moving 156
 - CREATE INDEX statement 108
 - creating tables 31
 - deleting 36
 - encoding 217
 - CCSIDs to encoding names 344
 - names to CCSID mappings 333
 - exporting 159
 - importing 162
 - indeterminate query evaluation 79
 - indexing
 - overview 93
 - inserting 32
 - JDBC applications 177

- XML data (*continued*)
 - model 7
 - querying in DB2 database 49
 - SQLJ applications 183
 - updating 34
 - XQuery data model 156
- XML data movement
 - overview 155
- XML data retrieval
 - CLI applications 169
 - comparison of methods 49
 - down-level clients 80
 - introduction 45
 - JDBC applications 180
 - result sets 70
 - SQLJ application 186
 - XMLEXISTS 66
 - XMLQUERY 51
 - XMLTABLE 61
- XML data store 1
- XML data type 17, 194
 - CLI applications 167
 - importing and exporting 157
 - indexing
 - overview 93
- XML declarations
 - CLI applications 167
 - encoding declarations 217
- XML documents
 - archival data types 89
 - differences after storage and retrieval 88
 - storage 29
 - storage requirements 29
- XML encoding 217
 - considerations
 - for routine parameters 220
 - in JDBC, SQLJ, and .NET 221
 - input of XML 219
 - retrieval of XML 219
 - internal
 - background 217
 - scenarios
 - input of externally encoded data 224
 - input of internally encoded data 222
 - retrieval with explicit serialization 228
 - retrieval with implicit serialization 226
- XML parameters
 - effects of commits and rollbacks on 193
 - invocation of routines from Java programs 182
- XML parsing 37
 - CLI applications 168
- XML schema registration
 - Java 137
- XML schema removal
 - Java 137
- XML schema repository
 - ADD XMLSCHEMA DOCUMENT
 - command 149
 - altering objects 139
 - XML schema repository (*continued*)
 - COMPLETE XMLSCHEMA
 - command 150
 - overview 133
 - REGISTER XMLSCHEMA
 - command 147
 - REGISTER XSROBJECT
 - command 151
 - registration with 134
 - for decomposition 237
 - Uniform Resource Identifier (URI)
 - location reference 133
 - validation 41
 - XML schema retrieval 140
 - XML schemas
 - component retrieval 140
 - listing all registered 139
 - XSR objects 133, 139
- XML schemas
 - component retrieval 140
 - enabling for decomposition 237
 - indexes over XML data 104
 - listing all registered 139
 - registering for decomposition 237
 - retrieval 140
 - structuring for decomposition 292
 - validation 41
- XML serialization 86
 - CLI applications 167, 169
- XML storage object
 - overview 29
- XML type 177
- XMLAGG aggregate function
 - publishing XML 81
- XMLATTRIBUTES scalar function
 - publishing XML 81
- xmlcolumn function 46
- XMLCOMMENT scalar function
 - publishing XML 81
- XMLDOCUMENT scalar function
 - publishing XML 81
- XMLELEMENT scalar function
 - publishing XML 81
- XMLEXISTS function 49
- XMLEXISTS predicate
 - querying with 66
 - passing constants 69
 - passing parameter markers 69
 - type casting 69
- XMLFOREST scalar function
 - publishing XML 81
- XMLNAMESPACES declaration
 - publishing XML 81
- XMLPARSE scalar function
 - parsing overview 37
- XMLPI scalar function
 - publishing XML 81
- XMLQUERY function 49
- XMLQUERY scalar function
 - overview 51
 - querying with
 - passing constants 69
 - passing parameter markers 69
 - results
 - casting to non-XML types 55
 - empty sequences 53
 - non-empty sequences 52
- XMLSERIALIZE scalar function
 - serialization overview 86
- XMLTABLE function 49
- XMLTABLE table function
 - example 63, 65
 - overview 61
- XMLTEXT scalar function
 - publishing XML 81
- XMLVALIDATE scalar function
 - validation overview 41
- XQuery 171
 - calling from SQL 191
 - comparison to SQL 49
 - invoking from SQL 49
 - XMLEXISTS 66
 - XMLQUERY 51
 - XMLTABLE 61
 - overview 45
 - results 70
 - XQuery data model 156
- XQuery and XPath data model 9
- XQuery statements
 - pattern expressions
 - used for index keys 94
 - specifying in the CLP 90
- XSR
 - altering objects 139
- XSR objects 133
 - registration 134
 - through command line processor 136
 - through stored procedure 135
- XSR_COMPLETE stored procedure 143
- XSR_DTD stored procedure 144
- XSR_ENTITY stored procedure 145
- XSR_REGISTER stored procedure 141

Contacting IBM

To contact IBM in your country or region, check the IBM Directory of Worldwide Contacts at <http://www.ibm.com/planetwide>

To learn more about DB2 products, go to <http://www.ibm.com/software/data/db2/>.



Printed in USA

SC10-4254-00



Spine information:

IBM DB2 DB2 Version 9

XML Guide

