# IBM

IBM i

# Database
# SQL XML Programming

*Version 7.1*

# IBM

IBM i

# Database
# SQL XML Programming

*Version 7.1*

# Note

Before using this information and the product it supports, read the information in "Notices" on page 223.

This edition applies to IBM i 7.1 (product number 5770-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

This document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

# Contents

# SQL XML programming

DB2® for IBM® i provides support to store and retrieve XML data using Structured Query Language (SQL). Objects defined using SQL such as tables, functions, and procedures can use the XML data type for column, parameter, and variable definitions. In addition to an XML data type, there are built-in functions and procedures that can be used to generate XML documents and to retrieve all or part of an XML document.

**Note:** By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 222.

## How to read the syntax diagrams

The following rules apply to the syntax diagrams used in this book.

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

  The ►►── symbol indicates the beginning of the syntax diagram.

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

  The ►── symbol indicates that the syntax is continued from the previous line.

  The ──►◄ symbol indicates the end of the syntax diagram.

  Diagrams of syntactical units start with the |── symbol and end with the ──| symbol.

- Required items appear on the horizontal line (the main path).

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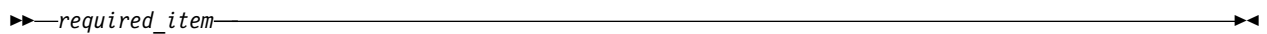
- Optional items appear below the main path.

  ```
  ►►──required_item─────────────────────────────────────────────►◄
                    └─optional_item─┘
  ```
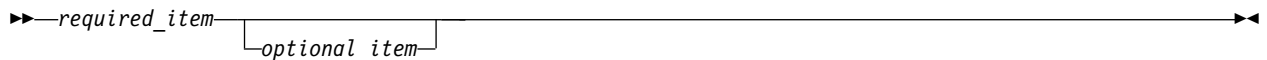
  If an item appears above the main path, that item is optional, and has no effect on the execution of the statement and is used only for readability.

  ```
                    ┌─optional_item─┐
  ►►──required_item──┴───────────────┴──────────────────────────►◄
  ```

- If more than one item can be chosen, they appear vertically, in a stack.

  If one of the items must be chosen, one item of the stack appears on the main path.

  ```
  ►►──required_item──┬─required_choice─┬─────────────────────────►◄
                     └─required_choice─┘
  ```

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

  ```
  ►►──required_item─┬─────────────────┬──────────────────────────►◄
                    ├─optional_choice─┤
                    └─optional_choice─┘
  ```
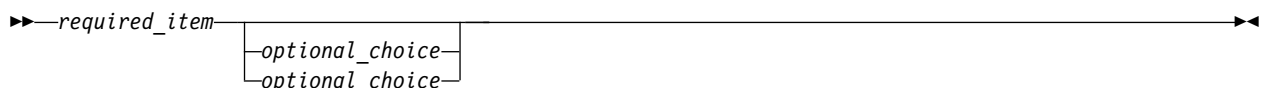
  If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.

```
        ┌─default_choice─┐
►►──required_item──┼────────────────┼──────────────────────────────►◄
        ├─optional_choice─┤
        └─optional_choice─┘
```

If an optional item has a default when it is not specified, the default appears above the main path.

```
►►──required_item──────────────────────────────────────────────────►◄
        │                    ┌─default_choice──┐  │
        ├─optional_choice──┼─required_choice─┤─┤
        │                    └─required_choice─┘  │
        └─optional_choice──────────────────────┘
```
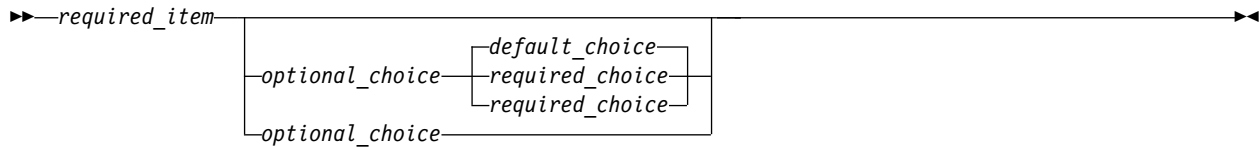
- An arrow returning to the left, above the main line, indicates an item that can be repeated.

```
                 ┌──────────────┐
                 ▼              │
►►──required_item───repeatable_item──────────────────────────────────►◄
```
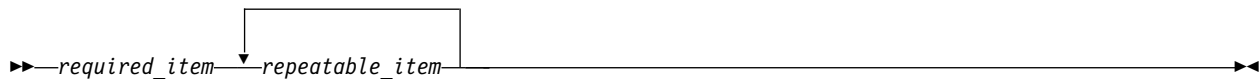
If the repeat arrow contains a comma, you must separate repeated items with a comma.

```
                 ┌──────,───────┐
                 ▼              │
►►──required_item───repeatable_item──────────────────────────────────►◄
```
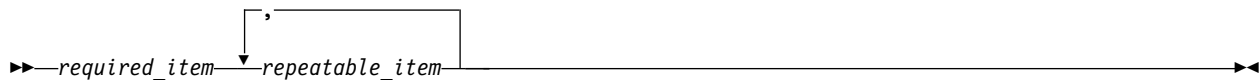
A repeat arrow above a stack indicates that the items in the stack can be repeated.
- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

# PDF file for SQL XML programming

You can view and print a PDF file of this information.

| To view or download the PDF version of this document, select SQL XML programming (about 935 KB).

## Saving PDF files

To save a PDF on your workstation for viewing or printing:
1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

## Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (http://get.adobe.com/reader/).

# SQL statements and SQL/XML functions

Many SQL statements support the XML data type. This enables you to perform many common database operations with XML data, such as creating tables with XML columns, adding XML columns to existing tables, creating triggers on tables with XML columns, and inserting, updating, or deleting XML documents. A set of SQL/XML functions, expressions, and specifications supported by DB2 database server takes full advantage of the XML data type.

## XML data type

The XML data type can store an XML value up to 2 GB. A CCSID can be specified for the XML data type. If a CCSID is not specified, the value of the SQL_XML_DATA_CCSID QAQQINI option will be used. The default for this option is 1208 (UTF-8). The XML data type can store single byte and Unicode double byte characters.

A single row in a table that contains one or more XML or LOB values cannot exceed 3.5 GB. The XML data type can be specified in a partitioned table.

XML host variables and XML locators can be declared in application programs.

XML locators can be used to refer to XML values. An XML value can be fetched into an XML locator. An XML locator can be passed to a procedure or function. The locator can be specified as a value on an INSERT or UPDATE statement.

Journal entries for XML columns are the same as for LOBs. See "Journal entry layout of LOB columns" in the SQL programming topic collection.

## Application development

Support for application development is provided by several programming languages, and through SQL and external functions and procedures:

**Programming language support**
> Application development support of XML enables applications to combine XML and relational data access and storage. The following programming languages support the XML data type for SQL:
> - ILE RPG
> - ILE COBOL
> - C or C++ (embedded SQL or DB2 CLI)
> - Java™ (JDBC or SQLJ)

**SQL and external functions and procedures**
> XML data can be passed to SQL procedures and external procedures by including parameters of data type XML in CREATE PROCEDURE parameter signatures. XML data can also be passed to SQL functions and external functions by including parameters of data type XML in CREATE FUNCTION parameter signatures. Existing SQL routine 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

The XML features provide a repository for managing the URI dependencies of XML documents:

**XML schema repository (XSR)**
> The XML schema repository (XSR) is a repository for all XML artifacts required to process XML

instance documents stored in XML columns. It stores XML schemas, referenced in XML documents. It can be used to validate or decompose XML instance documents.

## Tooling

Support for the XML data type is available with the IBM i Navigator.

## Annotated XML schema decomposition

The XML features enable you to store and access XML data as XML documents. There can be cases where accessing XML data as relational data is required. Annotated XML schema decomposition decomposes documents based on annotations specified in an XML schema.

# XML input and output overview

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

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 XML document form.

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

The method that you use to put XML data into the database system depends on the task you want to accomplish:

**Insert or update**
Well-formed documents are inserted into XML columns using the SQL INSERT statement. A document is well-formed when it can be parsed successfully. 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). Documents are updated using the SQL UPDATE statement.

**Annotated XML schema decomposition**
Data from XML documents can be decomposed or stored into relational and XML columns using annotated XML schema decomposition. Decomposition stores data in 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).

**XML schema repository (XSR) registration**
The XML schema repository (XSR) stores XML schemas that are used for the validation or decomposition of XML documents. Registration of XML schemas is usually a prerequisite for other tasks that are performed on XML documents which have a dependency on these schemas. XML schemas are registered with the XSR using stored procedures provided by DB2.

## Output

SQL is used to retrieve the XML data from the database system.

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. The XMLTABLE built in table function can be used to retrieve fragments of an XML document in an SQL query.

A number of publishing functions are also available to construct XML values from relational data stored in DB2 database server. XML values constructed with these publishing functions do not have to be well-formed XML documents.

# Comparison of XML and relational models

When you design your databases, you need to decide whether your data is better suited to the XML model or the relational model. Your design can also take advantage of the nature of a DB2 database - the ability to support both relational and XML data in a single database.

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 to assess the factors that can impact your specific implementation.

## Major differences between XML data and relational data

**XML data is hierarchical; relational data is represented in a model of logical relationships**

> 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 for one or more columns in a fullselect.

## Factors influencing data model choice

What kind of data you store can help you determine how you store it. For example, if the data is naturally hierarchical and self-describing, you might store it as XML data. However, there are other factors that might influence your decision about which model to use:

**When you need maximum flexibility**

> Relational tables follow a fairly rigid model. 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. XML schemas can be evolved over time, for example.

**When you need maximum performance for data retrieval**

> 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.

**When 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.

**When 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.

**When 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.

**When 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.

**When the data needs to be updated often**
You 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.

# Tutorial for XML

The XML data type 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 XML features.

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

* Creating a table that can store XML data
* Inserting XML documents into XML typed columns
* Updating XML documents stored in an XML column
* Validating XML documents against XML schemas
* Transforming with XSLT stylesheets

## Preparation

The examples in the exercises should be entered at or copied and pasted into the IBM i Navigator Run SQL Scripts tool. Using Interactive SQL will not show the XML result data as serialized data.

## Exercise 1: Creating a table that can store XML data

This exercise shows how to create a table that contains an XML column.

Create a table named `Customer` that contains an XML column:

```
CREATE SCHEMA POSAMPLE;

SET CURRENT SCHEMA POSAMPLE;

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 XML.

You can also add one or more XML columns to existing tables with the ALTER TABLE SQL statement.

Return to the tutorial

## Exercise 2: Inserting XML documents into XML typed columns

Well-formed XML documents are inserted into XML typed columns using the SQL INSERT statement. This exercise shows you how to insert XML documents into XML columns.

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 if XML documents from many sources are processed with different encodings.

This exercise shows how to insert XML documents into XML typed columns manually in Run SQL Scripts, 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 or UPDATE operations, however, string data can be directly assigned to XML columns without an explicit call to the XMLPARSE function. In these two 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;
```

If you are running this exercise in Interactive SQL, the XML values will not be serialized for you. You must explicitly use the XMLSERIALIZE function to see the inserted data.

Return to the tutorial

## Exercise 3: Updating XML documents stored in an XML column

This exercise shows you how to update XML documents with SQL statements.

**Updating with SQL**

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

Update one of the documents inserted in Exercise 2 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>'
WHERE Cid = 1002;
```

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

```
SELECT * from Customer;
```

If you are running this exercise in Interactive SQL, the XML values will not be serialized for you. You must explicitly use the XMLSERIALIZE function to see the updated data.

The row where Cid="1002" contains the changed <street>, <city>, and <pcode-zip> values.

XML documents can be identified by values in the non-XML columns of the same table.

Return to the tutorial

## Exercise 4: Validating XML documents against XML schemas

This exercise shows you how to validate XML documents. 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.

Register and complete registration of the posample.customer XML schema as follows:

```
CREATE PROCEDURE SAMPLE_REGISTER
LANGUAGE SQL
BEGIN
  DECLARE CONTENT BLOB(1M);
  VALUES BLOB('<?xml version="1.0"?>
<xs:schema targetNamespace="http://posample.org"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="customerinfo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string" minOccurs="1" />
```

```
              <xs:element name="addr" minOccurs="1" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="street" type="xs:string" minOccurs="1" />
                    <xs:element name="city" type="xs:string" minOccurs="1" />
                    <xs:element name="prov-state" type="xs:string" minOccurs="1" />
                    <xs:element name="pcode-zip" type="xs:string" minOccurs="1" />
                  </xs:sequence>
                  <xs:attribute name="country" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="phone" nillable="true" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" form="unqualified" type="xs:string" />
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element name="assistant" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="name" type="xs:string" minOccurs="0" />
                    <xs:element name="phone" nillable="true" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:simpleContent  >
                          <xs:extension base="xs:string">
                            <xs:attribute name="type" type="xs:string" />
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="Cid" type="xs:integer" />
          </xs:complexType>
        </xs:element>
</xs:schema>') INTO CONTENT;

  CALL SYSPROC.XSR_REGISTER('POSAMPLE', 'CUSTOMER', 'http://posample.org', CONTENT, null);
END;

SET PATH POSAMPLE;

CALL SAMPLE_REGISTER;

CALL SYSPROC.XSR_COMPLETE('POSAMPLE', 'CUSTOMER', null, 0);
```

You can verify that the XML schema was successfully registered by querying the QSYS2.XSROBJECTS catalog view, which contains information about objects stored in the XSR. This query and its result (formatted for clarity) are as follows:

```
SELECT XSROBJECTSCHEMA, XSROBJECTNAME FROM QSYS2.XSROBJECTS
  WHERE XSROBJECTSCHEMA = 'POSAMPLE';

XSROBJECTSCHEMA        XSROBJECTNAME
--------------------   --------------------
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 occur only 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 (1004, XMLVALIDATE (XMLPARSE (DOCUMENT
'<customerinfo xmlns="http://posample.org" Cid="1004">
  <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, XMLVALIDATE must be used in conjunction with the XMLPARSE function. Note that character data can be assigned directly to XML only in INSERT, UPDATE, or MERGE statements. Here, an INSERT statement is used. 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.

Return to the tutorial

## Exercise 5: Transforming with XSLT stylesheets

You can use the XSLTRANSFORM function to convert XML data within the database into other formats.

This example illustrates how to use the XSLTRANSFORM built-in function to transform XML documents that are stored in the database. In this case the XML document contains an arbitrary number of university student records. Each student element contains a student's ID, first name, last name, age, and the university he is attending, as follows:

```
<?xml version="1.0"?>
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <student studentID="1" firstName="Steffen" lastName="Siegmund"
    age="23" university="Rostock"/>
</students>
```

The intent of the XSLT transformation is to extract the information in the XML records and create an HTML web page that can be viewed in a browser. For that purpose we will use the following XSLT stylesheet, which is also stored in the database.

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="headline"/>
<xsl:param name="showUniversity"/>
<xsl:template match="students">
    <html>
    <head/>
    <body>
    <h1><xsl:value-of select="$headline"/></h1>
    <table border="1">
    <th>
```

```
<tr>
<td width="80">StudentID</td>
<td width="200">First Name</td>
<td width="200">Last Name</td>
<td width="50">Age</td>
<xsl:choose>
    <xsl:when test="$showUniversity =''true''">
        <td width="200">University</td>
    </xsl:when>
</xsl:choose>
</tr>
</th>
<xsl:apply-templates/>
</table>
</body>
</html>
</xsl:template>
    <xsl:template match="student">
        <tr>
        <td><xsl:value-of select="@studentID"/></td>
        <td><xsl:value-of select="@firstName"/></td>
        <td><xsl:value-of select="@lastName"/></td>
        <td><xsl:value-of select="@age"/></td>
        <xsl:choose>
            <xsl:when test="$showUniversity = ''true'' ">
                <td><xsl:value-of select="@university"/></td>
            </xsl:when>
        </xsl:choose>
        </tr>
    </xsl:template>
</xsl:stylesheet>
```

This stylesheet will work both with a standard XSLT transform, and using a supplied parameter file to control its behavior at runtime.

1. Create the table into which you can store your XML document and stylesheet document.

   ```
   SET CURRENT SCHEMA USER;
   CREATE TABLE XML_TAB (DOCID INTEGER, XML_DOC XML, XSL_DOC CLOB(1M));
   ```

2. Insert your documents into the tables. In this example the XML document and XSLT stylesheet can be loaded into the same table as separate column values. The INSERT statement uses a truncated version of the XSLT stylesheet as the third value. To use the following INSERT statement, replace the truncated stylesheet value with the XSLT stylesheet listed previously in this exercise.

   ```
   INSERT INTO XML_TAB VALUES
       (1,
        '<?xml version="1.0"?>
        <students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <student studentID="1" firstName="Steffen" lastName="Siegmund"
      age="23" university="Rostock"/>
        </students>',

        '<?xml version="1.0" encoding="UTF-8"?>
        <xsl:stylesheet version="1.0"
            ...
        </xsl:stylesheet>'
   );
   ```

3. Call the XSLTRANSFORM built-in function to transform the XML document.

   ```
   SELECT XSLTRANSFORM (XML_DOC USING XSL_DOC AS CLOB(1M)) FROM XML_TAB;
   ```

The output of this process will be the following HTML file:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
```

```
<body>
<h1></h1>
<table border="1">
<th>
<tr>
<td width="80">StudentID</td>
<td width="200">First Name</td>
<td width="200">Last Name</td>
<td width="50">Age</td>
</tr>
</th>
<tr>
<td>1</td>
<td>Steffen</td><td>Siegmund</td>
<td>23</td>
</tr>
</table>
</body>
</html>
```

While this is straightforward, there may be occasions when you want to alter the behavior of the XSLT stylesheet at runtime, either to add information not contained in the XML records or to change the nature of the output itself (to XHTML instead of standard HTML, for instance). You can pass parameters to the XSLT process at runtime by using a separate parameter file. The parameter file is itself an XML document and contains param statements that correspond to similar statements in the XSLT stylesheet file.

For instance, two parameters are defined in the stylesheet above as follows:

```
<xsl:param name="showUniversity"/>
<xsl:param name="headline"/>
```

These parameters were not used in the first transform as described above. To see how parameter-passing works, create a parameter file as follows:

```
CREATE TABLE PARAM_TAB (DOCID INTEGER, PARAM VARCHAR(1000));

INSERT INTO PARAM_TAB VALUES
    (1,
    '<?xml version="1.0"?>
    <params xmlns="http://www.ibm.com/XSLTransformParameters">
        <param name="showUniversity" value="true"/>
        <param name="headline">The student list ...</param>
    </params>'
    );
```

Examine this query:

```
SELECT XSLTRANSFORM (
    XML_DOC USING XSL_DOC WITH PARAM AS CLOB(1M)) FROM XML_TAB X, PARAM_TAB P
WHERE X.DOCID=P.DOCID;
```

The above query generates the following HTML:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1>The student's list ...</h1>
<table border="1">
<th>
<tr>
<td width="80">StudentID</td>
<td width="200">First Name</td>
<td width="200">Last Name</td>
<td width="50">Age</td>
<td width="200">University</td>
```

```
</tr>
</th>
<tr>
<td>1</td>
<td>Steffen</td>
<td>Siegmund</td><td>23</td><td>Rostock</td>
</tr>
</table>
</body>
</html>
```

Return to the tutorial

# Inserting XML data

Before you can insert XML documents, you must create a table that contains an XML column, or add an XML column to an existing table.

## 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. A table can have one or more XML columns.

**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:

```
SET CURRENT SCHEMA USER;

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

## 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.

You should insert XML data from host variables, rather than literals, so that the DB2 database server can use the host variable data type to determine some of the encoding information.

XML data in an application is in its serialized string format. When you insert the data into an XML column, it must be converted to the stored XML document 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 document 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);
```

## XML parsing

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

You can let the DB2 database server 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, or MERGE statement. The implicit parsing occurs when the statement is executed.

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

The XMLPARSE function takes a non-XML character, binary, or Unicode graphic 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 <b> and between </b> and </a> 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 server 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 server 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.

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 <b> are always preserved, regardless of any XML parsing options, because the spaces are contained within an element which is defined with the attribute xml:space="preserve":

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

However, in the following document, the spaces immediately before and after <b> can be controlled by the XML parsing options, because the spaces are contained within an element which is defined with the attribute xml:space="default":

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

# SQL/XML publishing functions for constructing XML values

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

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.

The following SQL/XML publishing functions can be used 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 be used only as an argument of the XMLELEMENT function.

**XMLCOMMENT scalar function**
    Returns an XML value 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 that is a well-formed XML document. Every XML value that is stored in a DB2 table must be a document. This function forms an XML document from the XML value.

**XMLELEMENT scalar function**
    Returns an XML value that is an XML element. Every XML value that is stored in a DB2 table must be a document. The XMLELEMENT function does not create a document, only an element. The stored XML value must be a document formed by the XMLDOCUMENT function.

**XMLFOREST scalar function**
    Returns an XML value that is a sequence of XML elements.

**XMLGROUP aggregate function**
    Returns a single top-level element to represent a table or the result of a query. By default each row in the result set is mapped to a row subelement and each input expression is mapped to a subelement of the row subelement. Optionally, each row in the result can be mapped to a row subelement and each input expression to be mapped to an attribute of the row subelement.

**XMLNAMESPACES declaration**
    Constructs namespace declarations from the arguments. This declaration can be used only as an argument of the XMLELEMENT and XMLFOREST functions.

**XMLPI scalar function**
    Returns an XML value with a single processing instruction.

**XMLROW scalar function**
    Returns a sequence of row elements to represent a table or the result of a query. By default each input expression is transformed into a subelement of a row element. Optionally, each input expression can be transformed into an attribute of a row element.

**XMLTEXT scalar function**
    Returns an XML value that contains the value of the input argument.

**XSLTRANSFORM scalar function**
    Converts XML data into other formats, including other XML schemas.

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

This example shows how you can construct XML values suitable for publishing from a single table with SQL/XML publishing functions.

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 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
```

This query returns the following XML value. It is formatted here to improve readability.

```
<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>
```

You can construct a similar XML document that contains a sequence of row elements by using the XMLROW function instead of aggregating the elements with XMLAGG. Item elements are also given a namespace prefix:

```
SELECT XMLELEMENT (NAME "products",
                   XMLNAMESPACES ('http://posample.org' AS "po"),
                   XMLROW(NAME AS "po:item"))
FROM Product
```

The resulting output is as follows:

```
<products xmlns:po="http://posample.org">
    <row>
        <po:item>Snow Shovel, Basic 22 inch</po:item>
    </row>
</products>
<products xmlns:po="http://posample.org">
    <row>
        <po:item>Snow Shovel, Deluxe 24 inch</po:item>
    </row>
</products>
<products xmlns:po="http://posample.org">
        <row><po:item>Snow Shovel, Super Deluxe 26 inch</po:item>
    </row>
</products>
<products xmlns:po="http://posample.org">
        <row><po:item>Ice Scraper, Windshield 4 inch</po:item>
    </row>
</products>
```

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

This example shows how you can construct XML values suitable for publishing from multiple tables with SQL/XML publishing functions.

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 *NAME* column in the PRODUCT table and the *QUANTITY* column in the INVENTORY table.
- 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>
```

## Example: Construct an XML document with values from table rows that contain null elements

This example shows how you can construct XML values suitable for publishing from table rows that contain null elements with SQL/XML publishing functions.

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.

This example assumes 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
```

In the result value, there is no <loc> element for the row that contains the null value.

```
<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
```

In the result value, there is an empty <loc> element.

```
<newElem prodID="100-100-01">
  <quantity>5</quantity>
  <loc/>
</newElem>
```

## Example: Transforming with XSLT stylesheets

The standard way to transform XML data into other formats is by Extensible Stylesheet Language
Transformations (XSLT). You can use the built-in XSLTRANSFORM function to convert XML documents
into HTML, plain text, or different XML schemas.

XSLT uses stylesheets to convert XML into other data formats. You can convert part or all of an XML
document and select or rearrange the data using the XPath query language and the built-in functions of
XSLT. XSLT is commonly used to convert XML to HTML, but can also be used to transform XML
documents that comply with one XML schema into documents that comply with another schema. XSLT
can also be used to convert XML data into unrelated formats, like comma-delimited text or formatting
languages such as troff. XSLT has two main areas of applicability:

- Formatting (conversion of XML into HTML)
- Data exchange (querying, reorganizing and converting data from one XML schema to another, or into a
  data exchange format such as SOAP)

Both cases may require that an entire XML document or only selected parts of it be transformed. XSLT
incorporates the XPath specification, permitting query and retrieval of arbitrary data from the source
XML document. An XSLT template may also contain or create additional information such as file headers
and instruction blocks that will be added to the output file.

### How XSLT Works

XSLT stylesheets are written in Extensible Stylesheet Language (XSL), an XML schema. XSL is a template
language rather than an algorithmic language such as C or Perl, a feature that limits XSL's power but
makes it uniquely suited to its purpose. XSL stylesheets contain one or more `template` elements, which
describe what action to take when a given XML element or query is encountered in the target file. A
typical XSLT template element will start by specifying which element it applies to. For instance,

```
<xsl:template match="product">
```

declares that the contents of this template will be used to replace the content of any `<product>` tag
encountered in the target XML file. An XSLT file consists of a list of such templates, in no necessary
order.

The following example shows typical elements of an XSLT template. In this case the target will be XML
documents containing inventory information, such as this record describing an ice scraper:

```
<?xml version="1.0"?>
<product pid="100-201-01">
    <description>
        <name>Ice Scraper, Windshield 4 inch</name>
        <details>Basic Ice Scraper 4 inches wide, foam handle</details>
        <price>3.99</price>
    </description>
</product>
```

This record includes such information as the part number, description, and price of a windshield ice scraper. Some of this information is contained within elements, such as <name>. Some, like the part number, are contained in attributes (in this case the pid attribute of the <product> element). To display this information as a web page, you could apply the following XSLT template:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
    <xsl:template match="/">
        <html>
            <body>
                <h1><xsl:value-of select="/product/description/name"/></h1>
                <table border="1">
                        <th>
        <xsl:apply-templates select="product"/>
                        </th>
                    </table>
            </body>
        </html>
    </xsl:template>
      <xsl:template match="product">
        <tr>
                <td width="80">product ID</td>
                <td><xsl:value-of select="@pid"/></td>
            </tr>
            <tr>
                <td width="200">product name</td>
                <td><xsl:value-of select="/product/description/name"/></td>
            </tr>
            <tr>
                <td width="200">price</td>
                <td>$<xsl:value-of select="/product/description/price"/></td>
            </tr>
            <tr>
                <td width="50">details</td>
                <td><xsl:value-of select="/product/description/details"/></td>
            </tr>
    </xsl:template>
</xsl:stylesheet>
```

When an XSLT processor receives as input both the template and target documents above, it will output the following HTML document:

```
<html>
<body>
<h1>Ice Scraper, Windshield 4 inch</h1>
<table border="1">
<th>
<tr>
<td width="80">product ID</td><td>100-201-01</td>
</tr>
<tr>
<td width="200">product name</td><td>Ice Scraper, Windshield 4 inch</td>
</tr>
<tr>
<td width="200">price</td><td>$3.99</td>
</tr>
<tr>
<td width="50">details</td><td>Basic Ice Scraper 4 inches wide, foam handle</td>
</tr>
</th>
</table>
</body>
</html>
```

The XSLT processor tests the incoming XML document for given conditions (typically one condition per template). If a condition is true the template contents are inserted into the output, and if they are false the template is passed over by the processor. The stylesheet may add its own data to the output, for example in the HTML table tagging and strings such as "product ID."

XPath can be used both to define template conditions, as in `<xsl:template match="product">` and to select and insert data from anywhere in the XML stream, as in `<h1><xsl:value-of select="/product/description/name"/></h1>`.

## Using XSLTRANSFORM

You can use the XSLTRANSFORM function to apply XSLT stylesheets to XML data. If you supply the function with the name of an XML document and an XSLT stylesheet, the function will apply the stylesheet to the document and return the result.

## Example: Using XSLT as a formatting engine

The following example illustrates how to use the built-in XSLTRANSFORM function as a formatting engine.

To get set up, first insert the two example documents below into the database.

```
INSERT INTO XML_TAB VALUES
(1,
    '<?xml version="1.0"?>
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation = "/home/steffen/xsd/xslt.xsd">
<student studentID="1" firstName="Steffen" lastName="Siegmund"
    age="23" university="Rostock"/>
</students>',

    '<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="headline"/>
<xsl:param name="showUniversity"/>
<xsl:template match="students">
    <html>
            <head/>
              <body>
               <h1><xsl:value-of select="$headline"/></h1>
               <table border="1">
                 <th>
                     <tr>
                             <td width="80">StudentID</td>
                             <td width="200">First Name</td>
                             <td width="200">Last Name</td>
                             <td width="50">Age</td>
                             <xsl:choose>
        <xsl:when test="$showUniversity ='true'">
                                 <td width="200">University</td>
                                  </xsl:when>
        </xsl:choose>
                         </tr>
                     </th>
                     <xsl:apply-templates/>
                 </table>
               </body>
             </html>
         </xsl:template>
       <xsl:template match="student">
            <tr>
               <td><xsl:value-of select="@studentID"/></td>
               <td><xsl:value-of select="@firstName"/></td>
               <td><xsl:value-of select="@lastName"/></td>
```

```
                    <td><xsl:value-of select="@age"/></td>
                    <xsl:choose>
                        <xsl:when test="$showUniversity = 'true' ">
                          <td><xsl:value-of select="@university"/></td>
                        </xsl:when>
                    </xsl:choose>
        </tr>
         </xsl:template>
</xsl:stylesheet>'
);
```

Next, call the XSLTRANSFORM function to convert the XML data into HTML and display it.

```
SELECT XSLTRANSFORM (XML_DOC USING XSL_DOC AS CLOB(1M)) FROM XML_TAB;
```

The result is this document:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1></h1>
<table border="1">
<th>
<tr>
<td width="80">StudentID</td>
<td width="200">First Name</td>
<td width="200">Last Name</td>
<td width="50">Age</td>
</tr>
</th>
 <tr>
<td>1</td>
<td>Steffen</td><td>Siegmund</td>
<td>23</td>
</tr>
  </table>
</body>
</html>
```

In this example, the output is HTML and the parameters influence only what HTML is produced and what data is brought over to it. As such it illustrates the use of XSLT as a formatting engine for end-user output.

## **Example: Using XSLT for data exchange**

This example illustrates how to use the built-in XSLTRANSFORM function to convert XML documents for data exchange.

This example uses parameters with the stylesheet to produce different data exchange formats at runtime.

We use a stylesheet that incorporates xsl:param elements to capture data from a parameter file.

```
INSERT INTO Display_productdetails values(1, '<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="headline"/>
<xsl:param name="supermarketname"/>
<xsl:template match="product">
    <html>
            <head/>
                <body>
                <h1><xsl:value-of select="$headline"/></h1>
                <table border="1">
                        <th>
                        <tr>
```

```
                              <td width="80">product ID</td>
                              <td width="200">product name</td>
                              <td width="200">price</td>
                              <td width="50">details</td>
                              <xsl:choose>
        <xsl:when test="$supermarket =''true'' ">
                                    <td width="200">BIG BAZAAR super market</td>
                                    </xsl:when>
         </xsl:choose>
                                   </tr>
                                   </th>
                              <xsl:apply-templates/>
                    </table>
                    </body>
                    </html>
           </xsl:template>
       <xsl:template match="product">
              <tr>
                    <td><xsl:value-of select="@pid"/></td>
                    <td><xsl:value-of select="/product/description/name"/></td>
                    <td><xsl:value-of select="/product/description/price"/></td>
                    <td><xsl:value-of select="/product/description/details"/></td>
                    </tr>
        </xsl:template>
     </xsl:stylesheet>'
);
```

The parameter file contains parameters corresponding to the ones in the XSLT template, with content:

```
CREATE TABLE PARAM_TAB (DOCID INTEGER, PARAM VARCHAR (10K));

INSERT INTO PARAM_TAB VALUES
(1,
'<?xml version="1.0"?>
<params xmlns="http://www.ibm.com/XSLTransformParameters">
        <param name="supermarketname" value="true"/>
        <param name="headline">BIG BAZAAR super market</param>
</params>'
);
```

You can then apply the parameter file at runtime using the following command:

```
SELECT XSLTRANSFORM (XML_DOC USING XSL_DOC WITH PARAM AS CLOB (1M))
    FROM product_details X, PARM_TAB P WHERE X.DOCID=P.DOCID;
```

The result is HTML, but with content determined by the parameter file and tests done against the content of the XML document:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1></h1>
<table border="1">
<th>
<tr>
<td width="80">product ID</td>
<td width="200">product Name</td>
<td width="200">price</td>
<td width="50">Details</td>
</tr>
</th>
</table>
</body>
</html>
```

In other applications, the output of XSLTRANSFORM might not be HTML but rather another XML document or a file using a different data format, such as an EDI file.

For data exchange applications, the parameter file could contain EDI or SOAP file header information such as e-mail or port addresses, or other critical data unique to a particular transaction. Since the XML used in the above examples is an inventory record, it is easy to imagine using XSLT to repackage this record for exchange with a client's purchasing system.

## Example: Using XSLT to remove namespaces

XML documents you receive might contain unneeded or incorrect namespace information. You can use XSLT style sheets to remove or manipulate the namespace information in the documents.

The following examples show how to use XSLT to remove namespace information from an XML document. The examples store the XML document and the XSLT stylesheets in XML columns and use the XSLTRANSFORM function to convert the XML document using one of the XSLT stylesheets.

The following CREATE statements create the tables XMLDATA and XMLTRANS. XMLDATA contains a sample XML document, and XMLTRANS contains XSLT stylesheets.

```
CREATE TABLE XMLDATA (ID BIGINT NOT NULL PRIMARY KEY, XMLDOC XML );
CREATE TABLE XMLTRANS (XSLID BIGINT NOT NULL PRIMARY KEY, XSLT XML );
```

Add the sample XML document to the XMLDATA table using the following INSERT statement.

```
insert into XMLDATA (ID, XMLDOC) values ( 1, '
<newinfo xmlns="http://mycompany.com">
<!-- merged customer information -->
  <customerinfo xmlns="http://oldcompany.com" xmlns:d="http://test" Cid="1004">
     <name>Matt Foreman</name>
  <addr country="Canada">
     <street>1596 Baseline</street>
     <city>Toronto</city>
     <prov-state>Ontario</prov-state>
     <pcode-zip>M3Z 5H9</pcode-zip>
  </addr >
  <phone type="work" >905-555-4789</phone>
  <h:phone xmlns:h="http://test1" type="home">416-555-3376</h:phone>
  <d:assistant>
     <name>Gopher Runner</name>
     <h:phone xmlns:h="http://test1" type="home">416-555-3426</h:phone>
  </d:assistant>
  </customerinfo>
</newinfo>
');
```

## Example XSLT stylesheet that removes all namespaces

The following example uses an XSLT stylesheet to remove all namespace information from the XML document stored in the table XMLDATA. The examples stores the stylesheet in the table XMLTRANS and uses a SELECT statement to apply the stylesheet to the XML document.

Add the stylesheet to the XMLTRANS table using the INSERT statement.

```
insert into XMLTRANS (XSLID, XSLT) values ( 1, '
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

 <!-- keep comments -->
 <xsl:template match="comment()">
    <xsl:copy>
       <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
```

```
|      <xsl:template match="*">
|        <!-- remove element prefix -->
|        <xsl:element name="{local-name()}">
|          <!-- process attributes -->
|          <xsl:for-each select="@*">
|            <!-- remove attribute prefix -->
|            <xsl:attribute name="{local-name()}">
|              <xsl:value-of select="."/>
|            </xsl:attribute>
|          </xsl:for-each>
|          <xsl:apply-templates/>
|        </xsl:element>
|      </xsl:template>
|    </xsl:stylesheet>
|    ') ;
```

The following SELECT statement converts the sample XML document using the XSLT stylesheet.

```
SELECT XSLTRANSFORM (XMLDOC USING XSLT )
  FROM XMLDATA, XMLTRANS
  where ID = 1 and XSLID = 1
```

The XSLTRANSFORM command converts the XML document using the first XSLT stylesheet and returns the following XML with all the namespace information removed.

```
<?xml version="1.0" encoding="UTF-8"?>
<newinfo>
 <!-- merged customer information -->
 <customerinfo Cid="1004">
  <name>Matt Foreman</name>
  <addr country="Canada">
   <street>1596 Baseline</street>
   <city>Toronto</city>
   <prov-state>Ontario</prov-state>
   <pcode-zip>M3Z 5H9</pcode-zip>
  </addr>
  <phone type="work">905-555-4789</phone>
  <phone type="home">416-555-3376</phone>
  <assistant>
   <name>Gopher Runner</name>
   <phone type="home">416-555-3426</phone>
  </assistant>
 </customerinfo>
</newinfo>
```

## Example XSLT stylesheet that keeps the namespace binding for an element

The following example uses an XSLT stylesheet keeps the namespace binding for only the phone elements. The name of the element is specified in the XSLT variable mynode. The example stores the stylesheet in the table XMLTRANS and uses a SELECT statement to apply the stylesheet to the XML document.

Add the stylesheet to the XMLTRANS table using the following INSERT statement.

```
insert into XMLTRANS (XSLID, XSLT) values ( 2, '
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
  <xsl:variable name ="mynode">phone</xsl:variable>

  <!-- keep comments -->
  <xsl:template match="comment()">
      <xsl:copy>
        <xsl:apply-templates/>
      </xsl:copy>
  </xsl:template>
```

```
|    <xsl:template xmlns:d="http://test" xmlns:h="http://test1" match="*">
|    <xsl:choose>
|
|    <!-- keep namespace prefix for node names $mynode -->
|      <xsl:when test="local-name() = $mynode " >
|       <xsl:element name="{name()}">
|         <!-- process node attributes -->
|         <xsl:for-each select="@*">
|         <!-- remove attribute prefix  -->
|           <xsl:attribute name="{local-name()}">
|             <xsl:value-of select="."/>
|           </xsl:attribute>
|         </xsl:for-each>
|        <xsl:apply-templates/>
|       </xsl:element>
|      </xsl:when>
|
|    <!-- remove namespace prefix from node -->
|      <xsl:otherwise>
|       <xsl:element name="{local-name()}">
|         <!-- process node attributes -->
|         <xsl:for-each select="@*">
|         <!-- remove attribute prefix  -->
|           <xsl:attribute name="{local-name()}">
|             <xsl:value-of select="."/>
|           </xsl:attribute>
|         </xsl:for-each>
|        <xsl:apply-templates/>
|       </xsl:element>
|      </xsl:otherwise>
|    </xsl:choose>
|    </xsl:template>
|  </xsl:stylesheet>
|  ');
```

The following SELECT statement converts the sample XML document using the second XSLT stylesheet since XSLID = 2 is specified.

```
SELECT XSLTRANSFORM (XMLDOC USING XSLT)
  FROM XMLDATA, XMLTRANS
  where ID = 1 and XSLID = 2 ;
```

The XSLTRANSFORM command converts the XML document using the second XSLT stylesheet and returns the following XML with the namespaces for only the phone elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<newinfo>
 <!-- merged customer information -->
 <customerinfo Cid="1004">
  <name>Matt Foreman</name>
  <addr country="Canada">
   <street>1596 Baseline</street>
   <city>Toronto</city>
   <prov-state>Ontario</prov-state>
   <pcode-zip>M3Z 5H9</pcode-zip>
  </addr>
  <phone type="work">905-555-4789</phone>
  <h:phone xmlns:h="http://test1" type="home">
   416-555-3376
  </h:phone>
  <assistant>
   <name>Gopher Runner</name>
   <h:phone xmlns:h="http://test1" type="home">
    416-555-3426
```

```
    </h:phone>
   </assistant>
  </customerinfo>
</newinfo>
```

## Important considerations for transforming XML documents

When using the built-in XSLTRANSFORM function to convert XML documents some important
considerations and restrictions apply.

Note the following when transforming XML documents:

- Source XML documents must be single-rooted and well-formed.
- Because XSLT transformation by default produces UTF-8 characters, the output stream might lose
  characters if inserted into a character column that is not Unicode.

### Restrictions

- Only the W3C XSLT Version 1.10 Recommendation is supported.
- All parameters and the result type must be SQL types; they cannot be file names.
- Transformation with more than one stylesheet document (using an xsl:include declaration) is not
  supported.

## Special character handling in SQL/XML publishing functions

SQL/XML publishing functions have a 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 1. Special characters and their entity representations*

| Special character | Entity representation |
|---|---|
| < | &lt; |
| > | &gt; |
| & | &amp; |
| " | &quot; |

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&#x0040;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 a name 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.

# XML serialization

XML serialization is the process of converting XML data from the format that 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.

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.

In general, implicit serialization is preferable because it is more efficient to send data to the client as XML data. However, under certain circumstances (described later), it is better to do an explicit 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.

## Implicit XML serialization

With implicit serialization for DB2 CLI and embedded SQL applications, the DB2 database server adds an XML declaration with the appropriate encoding specified to the data. For .NET applications, the DB2 database server also adds an XML declaration. For Java applications, depending on the SQLXML object methods that are called to retrieve the data from the SQLXML object, the data with an XML declaration added by the DB2 database server will be returned.

**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;
```

## Explicit XML 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.

The XMLSERIALIZE scalar function lets you specify:
- The SQL data type to which the data is converted when it is serialized
  The data type is a character, graphic, or binary data type.
- Whether the output data should include the explicit encoding specification (EXCLUDING XMLDECLARATION or INCLUDING XMLDECLARATION).

The output from XMLSERIALIZE is Unicode, character, or graphic data.

If you retrieve the serialized data into an 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, when the client does not support XML data, it is better to do an explicit XMLSERIALIZE:

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 a CLOB or DBCLOB before sending the data to the client.

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

**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)) into :hostvar from Customer
  WHERE CID=1000
```

## Differences in an XML document after storage and retrieval

When you store 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.

Some of the changes to the document occur when the document is stored. Those changes are:
- If you execute XMLVALIDATE, the database server:
  - 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 specified to the data. For .NET applications, the DB2 database server also adds an XML declaration. For Java applications, depending on the SQLXML object methods that are called to retrieve the data from the SQLXML object, the data with an XML declaration added by the DB2 database server will be returned.

  If you execute the XMLSERIALIZE function, the DB2 database server adds an XML declaration with an encoding specification 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:

SQL XML programming    **29**

| Character | Unicode value | Entity representation |
|---|---|---|
| AMPERSAND | U+0026 | &amp; |
| LESS-THAN SIGN | U+003C | &lt; |
| GREATER-THAN SIGN | U+003E | &gt; |

- Within attribute values, the QUOTATION MARK (U+0022) character is replaced with its predefined XML entity &quot;.
- 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.

## 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 CCSID conversions, which can make a document inconsistent with its original form.

# Using XMLTABLE to reference XML content as a relational table

The XMLTABLE built-in table function can be used to retrieve the content of an XML document as a result set that can be referenced in SQL.

Assume you have a table called EMP with an XML column defined like this:

```
CREATE TABLE EMP (DOC XML)
```

The table contains 2 rows, which look like this:

```
<dept bldg="101">
  <employee id="901">
    <name>
      <first>John</first>
      <last>Doe</last>
    </name>
    <office>344</office>
    <salary currency="USD">55000</salary>
  </employee>
  <employee id="902">
    <name>
      <first>Peter</first>
      <last>Pan</last>
    </name>
    <office>216</office>
    <phone>905-416-5004</phone>
  </employee>
</dept>
<dept bldg="114">
  <employee id="903">
    <name>
      <first>Mary</first>
      <last>Jones</last>
    </name>
    <office>415</office>
    <phone>905-403-6112</phone>
    <phone>647-504-4546</phone>
    <salary currency="USD">64000</salary>
  </employee>
</dept>
```

In the XMLTABLE function invocation, you specify a row-generating XPath expression and, in the columns clause, one or more column-generating expressions. In this example, the row-generating expression is the XPath expression $d/dept/employee. The passing clause indicates that the variable $d refers to the XML column doc of the table emp.

```
SELECT X.*
  FROM emp,
       XMLTABLE ('$d/dept/employee' PASSING emp.doc AS "d"
                 COLUMNS
                       empID     INTEGER     PATH '@id',
                       firstname VARCHAR(20) PATH 'name/first',
                       lastname  VARCHAR(25) PATH 'name/last') AS X
```

The row-generating expression is applied to each XML document in the XML column and produces one or multiple employee elements (sub-trees) per document. The output of the XMLTABLE function contains one row for each employee element. Hence, the output produced by the row-generating XPath expression determines the cardinality of the result set of the SELECT statement.

The COLUMNS clause is used to transform XML data into relational data. Each of the entries in this clause defines a column with a column name and an SQL data type. In the example above, the returned rows have 3 columns named empID, firstname, and lastname of data types Integer, Varchar(20), and Varchar(25), respectively. The values for each column are extracted from the employee elements, which are produced by the row-generating XPath expression, and cast to the SQL data types. For example, the path name/first is applied to each employee element to obtain the value for the column firstname. The row-generating expression provides the context for the column-generating expressions. In other words, you can typically append a column-generating expression to the row-generating expression to get an idea of what a given XMLTABLE function returns for a column.

The result of the previous query is:

```
EMPID       FIRSTNAME            LASTNAME
----------- -------------------- ------------------------
        901 John                 Doe
        902 Peter                Pan
        903 Mary                 Jones
```

Be aware that the path expressions in the COLUMNS clause must not return more than one item per row. If a path expression returns a sequence of two or more items, the XMLTABLE execution will typically fail, as it is not possible to convert a sequence of XML values into an atomic SQL value.

## Example: Use XMLTABLE to handle missing elements

XML data can contain optional elements that are not present in all of your documents

For example, employee Peter Pan does not have a salary element since it is not a required data field. It's easy to deal with that because the XMLTABLE function produces NULL values for missing elements. You can write XMLTABLE queries as if the salary element is always present.

```
SELECT X.*
  FROM emp,
       XMLTABLE ('$d/dept/employee' PASSING doc AS "d"
                 COLUMNS
                       empID     INTEGER     PATH '@id',
                       firstname VARCHAR(20) PATH 'name/first',
                       lastname  VARCHAR(25) PATH 'name/last',
                       salary    INTEGER     PATH 'salary') AS X
```

This query returns the following result. Note that the salary column for Peter Pan has the NULL value since the XML document contains no salary value.

```
| EMPID       FIRSTNAME            LASTNAME                 SALARY
| -----------  -------------------- ------------------------ ----------
|        901 John                 Doe                      55000
|        902 Peter                Pan                      -
|        903 Mary                 Jones                    64000
```

If you want a value other than NULL to appear for a missing element, you can define a default value to
use when the expected element is missing. Here, we define the salary result column to return 0 instead of
NULL.

```
SELECT X.*
   FROM emp,
        XMLTABLE ('$d/dept/employee' PASSING doc AS "d"
                   COLUMNS
                         empID    INTEGER            PATH '@id',
                         firstname VARCHAR(20)       PATH 'name/first',
                         lastname VARCHAR(25)        PATH 'name/last',
                         salary   INTEGER DEFAULT 0  PATH 'salary') AS X
```

## Example: Use XMLTABLE to subset result data

Often you want to produce a result containing a subset of the possible rows based on some filtering
predicate.

There are several ways to produce a subset of rows. One solution is to add a WHERE clause to the query
to filter using an output column. This requires all the rows to be generated only to be immediately
discarded. Another solution is to use filtering predicates in the row-generating expression of the
XMLTABLE function.

Suppose you need to produce rows only for employees in building 114. You can add a corresponding
condition to the XMLTABLE like this:

```
SELECT X.*
   FROM emp,
        XMLTABLE ('$d/dept[@bldg="114"]/employee' PASSING doc AS "d"
                   COLUMNS
                         empID    INTEGER            PATH '@id',
                         firstname VARCHAR(20)       PATH 'name/first',
                         lastname VARCHAR(25)        PATH 'name/last',
                         salary   INTEGER DEFAULT 0  PATH 'salary') AS X
```

This query returns a single row for Mary Jones, who is the only employee in building 114.

## Example: Use XMLTABLE to handle multiple values

Sometimes a path expression refers to an item that has multiple values.

The path expressions in the COLUMNS clause must not produce more than one item per row. In the
sample documents, notice that the employee Mary Jones has two phone numbers. If you need to query
this data and return a relational table with each employee's name and phone number, the query you
would write might look like this:

```
SELECT X.*
   FROM emp,
        XMLTABLE ('$d/dept/employee' PASSING doc AS "d"
                   COLUMNS
                         firstname VARCHAR(20)  PATH 'name/first',
                         lastname VARCHAR(25)   PATH 'name/last',
                         phone    VARCHAR(12)   PATH 'phone') AS X
```

When run against the sample documents, this query fails since there are two values for phone. A different
solution is needed.

**Return only first value**

One way to deal with this issue is to return only one of the multiple phone numbers. If you need summarized information for each employee, having just one phone number might be enough. Returning only one occurrence of the phone element can be done with a positional predicate in the XPath expression for the column `phone`.

Square brackets in XPath are used to specify predicates. To obtain the first phone element for an employee, use a positional predicate, written either as [1] or [fn:position()=1]. The first notation of [1] is an abbreviated version of the second.

```
SELECT X.*
   FROM emp,
        XMLTABLE ('$d/dept/employee' PASSING doc AS "d"
                  COLUMNS
                        firstname VARCHAR(20)  PATH 'name/first',
                        lastname  VARCHAR(25)  PATH 'name/last',
                        phone     VARCHAR(12)  PATH 'phone[1]') AS X
```

**Return multiple values as XML**

Another option to return multiple phone numbers for a single employee is to return an XML sequence of phone elements. To achieve this, the generated `phone` column needs to be of type XML, which allows you to return an XML value as the result of the XPath expression.

```
SELECT X.*
   FROM emp,
        XMLTABLE ('$d/dept/employee' PASSING doc AS "d"
                  COLUMNS
                        firstname VARCHAR(20)  PATH 'name/first',
                        lastname  VARCHAR(25)  PATH 'name/last',
                        phone     XML          PATH 'phone') AS X
```

The result of this query is:

```
FIRSTNAME    LASTNAME   PHONE
----------- ---------- ------------------
John         Doe        -
Peter        Pan        <phone>905-416-5004</phone>
Mary         Jones      <phone>905-403-6112</phone><phone>647-504-4546</phone>
```

The XML value returned in the phone column for Mary Jones is not a well-formed XML document since there is no single root element. This value can still be processed by DB2, but you won't be able to insert it into an XML column or parse it with an XML parser. Combining multiple phone numbers into a single VARCHAR or XML value may require additional code in your application to use the individual numbers.

**Return multiple columns**

Another solution is to return each phone number as a separate VARCHAR value by producing a fixed number of result phone columns. This example uses positional predicates to return phone numbers in two columns.

```
SELECT X.*
   FROM emp,
        XMLTABLE ('$d/dept/employee' PASSING doc AS "d"
                  COLUMNS
                        firstname VARCHAR(20)  PATH 'name/first',
                        lastname  VARCHAR(25)  PATH 'name/last',
                        phone     VARCHAR(12)  PATH 'phone[1]',
                        phone2    VARCHAR(12)  PATH 'phone[2]') AS X
```

An obvious drawback to this approach is that a variable number of items is being mapped to a fixed number of columns. One employee may have more phone numbers than anticipated. Others may have fewer which results in null values. If every employee has exactly one office phone and one cell phone, then producing two columns with corresponding names might be very useful.

**Return one row for each value**

Instead of returning the phone numbers in separate columns, you can also use XMLTABLE to return them in separate rows. In this case, you need to return one row for each phone number instead of one row for each employee. This may result in repeated information in the columns for the first and last names.

```
SELECT X.*
  FROM emp,
      XMLTABLE ('$d/dept/employee/phone' PASSING doc AS "d"
                COLUMNS
                      firstname VARCHAR(20)  PATH '../name/first',
                      lastname  VARCHAR(25)  PATH '../name/last',
                      phone     VARCHAR(12)  PATH '.') AS X
```

The result of this query is:

```
FIRSTNAME    LASTNAME    PHONE
 ----------- ---------- ------------------
Peter        Pan         905-416-5004
Mary         Jones       905-403-6112
Mary         Jones       647-504-4546
```

In this result, there is no row for John Doe since he has no phone number.

**Handling non-existent path values**

The previous example did not return a row for employee John Doe because the row-xquery expression iterates over all the phone elements and there is no phone element for the employee John Doe. As a result, the employee element for John Doe is never processed.

To resolve this issue, you need to use an SQL UNION of two XMLTABLE functions.

```
SELECT X.*
  FROM emp,
      XMLTABLE ('$d/dept/employee/phone' PASSING doc AS "d"
                COLUMNS
                      firstname VARCHAR(20)  PATH '../name/first',
                      lastname  VARCHAR(25)  PATH '../name/last',
                      phone     VARCHAR(12)  PATH '.') AS X
UNION
SELECT Y.*, CAST(NULL AS VARCHAR(12))
  FROM emp,
      XMLTABLE ('$d/dept/employee[fn:not(phone)]' PASSING doc AS "d"
                COLUMNS
                      firstname VARCHAR(20)  PATH 'name/first',
                      lastname  VARCHAR(25)  PATH 'name/last') AS Y
```

The `$d/dept/employee[fn:not(phone)]` row expression in the second XMLTABLE returns all employees with no phone numbers, adding the employee rows that were omitted in the first XMLTABLE.

## Example: Use XMLTABLE with namespaces

XML namespaces are a W3C XML standard for providing uniquely named elements and attributes in an XML document. XML documents may contain elements and attributes from different vocabularies but have the same name. By giving a namespace to each vocabulary, the ambiguity is resolved between identical element or attribute names.

In XML documents, you declare XML namespaces with the reserved attribute `xmlns`, whose value must contain an Universal Resource Identifier (URI). URIs are used as identifiers; they typically look like a URL but they don't have to point to an existing web page. A namespace declaration can also contain a prefix, used to identify elements and attributes. Below is an example of a namespace declaration with and without prefix:

```
xmlns:ibm = "http://www.ibm.com/xmltable/"
xmlns = "http://www.ibm.com/xmltable/"
```

To demonstrate the use of namespaces with XMLTABLE, a sample document is added to the previous example, so we are working with the following three rows:

```
<dept bldg="101">
   <employee id="901">
      <name>
         <first>John</first>
         <last>Doe</last>
      </name>
      <office>344</office>
      <salary currency="USD">55000</salary>
   </employee>
   <employee id="902">
      <name>
         <first>Peter</first>
         <last>Pan</last>
      </name>
      <office>216</office>
      <phone>905-416-5004</phone>
   </employee>
</dept>
<dept bldg="114">
   <employee id="903">
      <name>
         <first>Mary</first>
         <last>Jones</last>
      </name>
      <office>415</office>
      <phone>905-403-6112</phone>
      <phone>647-504-4546</phone>
      <salary currency="USD">64000</salary>
   </employee>
</dept>
<ibm:dept bldg="123" xmlns:ibm="http://www.ibm.com/xmltable">
   <ibm:employee id="144">
      <ibm:name>
         <ibm:first>James</ibm:first>
         <ibm:last>Bond</ibm:last>
      </ibm:name>
      <ibm:office>007</ibm:office>
      <ibm:phone>905-007-1007</ibm:phone>
      <ibm:salary currency="USD">77007</ibm:salary>
   </ibm:employee>
</ibm:dept>
```

In order to return all the employees in the database, you can use the * wildcard for the namespace prefix in the path expressions. This causes all elements to be considered, regardless of namespaces, because this wildcard (*) matches any namespace including no namespace.

```
SELECT X.*
   FROM emp,
        XMLTABLE ('$d/*:dept/*:employee' PASSING doc AS "d"
                   COLUMNS
                         empID     INTEGER    PATH '@*:id',
                         firstname VARCHAR(20) PATH '*:name/*:first',
                         lastname  VARCHAR(25) PATH '*:name/*:last') AS X
```

The result of the query is:

```
| EMPID        FIRSTNAME            LASTNAME
| ----------- -------------------- -------------------------
|         901 John                 Doe
|         902 Peter                Pan
|         903 Mary                 Jones
|         144 James                Bond
```

For this specific data, the namespace wildcard for the attribute @id was not strictly necessary. The reason is that the @id attribute employee James Bond has no namespace. Attributes never inherit namespaces from their element and also never assume the default namespace. So, unless the attribute name has a prefix, it doesn't belong to any namespace.

The use of the wildcard expression is the simplest way to return all employees, regardless of namespace.

**Declaring a default element namespace**

When all the elements you want to query belong to the same namespace, declaring a default element namespace can be the simplest way to write your queries. You just need to declare the default namespace in the beginning of your XPath expression and, after that, all unqualified elements you reference are tied to that namespace.

```
SELECT X.*
  FROM emp,
       XMLTABLE ('declare default element namespace "http://www.ibm.com/xmltable";
                  $d/dept/employee' PASSING doc AS "d"
                  COLUMNS
                      empID    INTEGER     PATH '@id',
                      firstname VARCHAR(20) PATH
                  'declare default element namespace "http://www.ibm.com/xmltable"; name/first',
                      lastname  VARCHAR(25) PATH
                  'declare default element namespace "http://www.ibm.com/xmltable"; name/last') AS X
```

The result is:
```
EMPID        FIRSTNAME            LASTNAME
----------- -------------------- -------------------------
        144 James                Bond
```

The column-generating expressions do not inherit the namespace declaration from the row-generating expression. Each column-generating expression is a separate XPath query and needs its own namespace declaration. These namespace declarations may differ from each other, for example, if your document contains multiple namespaces.

Often there is only one namespace, in which case it would be convenient to declare a single namespace for all expressions in the XMLTABLE function. This can be achieved by using the function XMLNAMESPACES(). This function allows you to declare a default element namespace and/or several namespace prefixes to be used within the XMLTABLE function. The advantage of using the XMLNAMESPACES function is that the declared namespaces are global for all expressions in the XMLTABLE context, so all the XPath expressions will be aware of these namespaces declarations and repeated namespace declarations are not required.

The default namespace declared by the XMLNAMESPACES function applies to both the row-generating expression and all the column-generating expressions. This way only one namespace declaration is needed for all XPath expressions in an XMLTABLE function. The result of the following query is exactly the same as the previous example.

```
SELECT X.*
  FROM emp,
       XMLTABLE (XMLNAMESPACES(DEFAULT 'http://www.ibm.com/xmltable'),
                  '$d/dept/employee' PASSING doc AS "d"
                  COLUMNS
```

```
                      empID    INTEGER    PATH '@id',
                      firstname VARCHAR(20) PATH 'name/first',
                      lastname  VARCHAR(25) PATH 'name/last') AS X
```

**Declaring a namespace prefix with XMLNAMESPACES**

If you want to select elements and attributes from multiple specific namespaces, then using namespace prefixes can be your best option. Unless you use the XMLNAMESPACES function, the namespaces prefixes need to be declared for every expression. But, just like for default element namespaces, you can use the XMLNAMESPACES function to avoid repeated namespace declarations.

```
SELECT X.*
  FROM emp,
       XMLTABLE (XMLNAMESPACES('http://www.ibm.com/xmltable' AS "ibm"),
                 '$d/ibm:dept/ibm:employee' PASSING doc AS "d"
                 COLUMNS
                      empID    INTEGER    PATH '@id',
                      firstname VARCHAR(20) PATH 'ibm:name/ibm:first',
                      lastname  VARCHAR(25) PATH 'ibm:name/ibm:last') AS X
```

## Example: Number result rows for XMLTABLE

In some cases, you may want to generate a column that numbers the rows that XMLTABLE produces for any given document. This can help your application to remember the order in which the values appeared in each document.

To number the result rows, use the FOR ORDINALITY clause. Note that the numbering starts with 1 for each document that is input to the XMLTABLE function.

```
SELECT X.*
  FROM emp,
       XMLTABLE ('$d/dept/employee' PASSING doc AS "d"
                 COLUMNS
                      seqno    FOR ORDINALITY,
                      empID    INTEGER    PATH '@id',
                      firstname VARCHAR(20) PATH 'name/first',
                      lastname  VARCHAR(25) PATH 'name/last') AS X
```

The result of the query is:

```
SEQNO    EMPID       FIRSTNAME            LASTNAME
-------- ----------- -------------------- ------------------------
      1         901 John                 Doe
      2         902 Peter                Pan
      1         903 Mary                 Jones
```

# Updating XML data

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. The application data type can be an XML, character, or binary type.

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.

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 assumed to be stored in a file c7.xml, whose contents look like this:

```
<customerinfo xmlns="http://posample.org" Cid="1004">
  <name>Christine Haas</name>
  <addr country="Canada">
    <street>12 Topgrove</street>
```

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

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

```
PreparedStatement updateStmt = null;
String sqls = null;
int cid = 1004;
sqls = "UPDATE 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 Info=: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. The WHERE clause in the UPDATE statements uses the relational column Cid to specify the rows to update.

## Deletion of XML data from tables

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

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

The following example demonstrates how XML data can be deleted from XML columns. The example uses 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
```

# XML schema repository

The XML schema repository (XSR) is a set of tables containing information about XML schemas.

XML instance documents might contain a reference to a Uniform Resource Identifier (URI) that points to an associated XML schema. This URI is required to process the instance documents. 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, an external resource may not be accessible when needed by the database. The XSR also removes the additional overhead required to locate external documents, along with the possible performance impact.

An XML schema consists of a set of XML schema documents. To add an XML schema to the DB2 XSR, you register XML schema documents to DB2, by calling the following DB2-supplied stored procedures:

**SYSPROC.XSR_REGISTER**
> Begins registration of an XML schema. You call this stored procedure when you add the first XML schema document to an XML schema.
>
> ```
> CALL SYSPROC.XSR_REGISTER ('user1', 'POschema',
>                     'http://myPOschema/PO',
>                     :content_host_var, NULL)
> ```

**SYSPROC.XSR_ADDSCHEMADOC**
> Adds additional XML schema documents to an XML schema that you are in the process of registering. You can call SYSPROC.XSR_ADDSCHEMADOC only for an existing XML schema that is not yet complete.
>
> ```
> CALL SYSPROC.XSR_ADDSCHEMADOC ('user1', 'POschema',
>                     'http://myPOschema/address',
>                     :content_host_var, NULL)
> ```

**SYSPROC.XSR_COMPLETE**
> Completes the registration of an XML schema.
>
> ```
> CALL SYSPROC.XSR_COMPLETE ('user1', 'POschema', :schemaproperty_host_var, 0)
> ```
>
> During XML schema completion, DB2 resolves references inside XML schema documents to other XML schema documents. 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.

To remove an XML schema from the DB2 XML schema repository, you can:
- call the SYSPROC.XSR_REMOVE stored procedure or
- use the DROP XSROBJECT SQL statement.

## Independent ASP considerations for the XML Schema Repository (XSR)

Because an independent auxiliary storage pool (ASP) can be switched between multiple systems, there are some additional considerations for administering XML schemas on an ASP.

Use of an XML schema must be contained on the independent ASP where it was registered. You cannot reference an XML schema that is defined in a different independent ASP group or in the system ASP when the job is connected to the independent ASP.

# Application programming language support

You can write applications to store XML data in DB2 databases tables, retrieve data from tables, or call stored procedures or user-defined functions with XML parameters.

You can use any of the following languages to write your applications:
- ILE RPG
- ILE COBOL
- C and C++ (embedded SQL or DB2 CLI)
- Java (JDBC or SQLJ)

An application program can retrieve an entire document that is stored in an XML column.

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 an XML value with the specified CCSID.

When an application retrieves data from XML columns, the DB2 database server converts the data from the XML value, with the specified CCSID, to the XML serialized string format. In addition, the database server might need to convert the output data from the XML CCSID to the string CCSID.

When you retrieve XML data, you need to be aware of the effect of CCSID conversion on data loss. Data loss can occur when characters in the source XML CCSID cannot be represented in the target string's CCSID.

When you fetch an XML document, you retrieve the serialized form of a document into an application variable.

## 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, you use `SQLBindParameter()` to bind parameter markers to input data buffers.

The SQL XML data type can be bound to the following application C character and graphic data types:
- SQL_C_CHAR
- SQL_VARCHAR
- SQL_C_WCHAR
- SQL_VARGRAPHIC

The following character LOB data types:
- SQL_C_CLOB
- SQL_C_CLOB_LOCATOR

and the following binary data types:
- SQL_C_BINARY
- SQL_C_BLOB
- SQL_C_BLOB_LOCATOR
- SQL_C_BINARY

When you bind a data buffer that contains XML data as a binary data type, 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.

**Note:** The XML data should be bound to a binary data type when the XML is received from many sources with different encoding schemes.

When you bind a data buffer that contains XML data as SQL_C_CHAR 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 UTF-16.
- If the C type is SQL_C_CHAR, DB2 CLI assumes that the data is encoded in the application's single-byte default CCSID.

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.

The following example shows how to update XML data in an XML column using the 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 = &apos;001&apos;", SQL_NTS);
SQLBindParameter (hStmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_XML, 0, 0,
                  xmlBuffer, 10240, &length);
SQLExecute (hStmt);
```

## 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 application C character and graphic data types, the character and graphic LOB data
types, and the binary data types. When retrieving a result set from an XML column, you should consider
binding your application variable to the binary types. Binding to character types can result in possible
data loss resulting from the CCSID conversion. Data loss can occur when characters in the source XML
CCSID cannot be represented in the target string CCSID. Binding your variable to the binary types 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 XML value encoding scheme.

- If the C type is SQL_C_CHAR, DB2 CLI returns the data in the application character encoding scheme.

- If the C type is SQL_C_WCHAR, DB2 CLI returns the data in the UTF-16 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=&apos;001&apos;", 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
```

## 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.

XML data is stored in an XML data type column in a table. Columns with the 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 or the SQL/XML extensions. 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.

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. graphic, and binary types, which include CHAR, VARCHAR, CLOB, DBCLOB, 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 applied.

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. The examples shown here are for C, but similar syntax exists for the other supported languages.

- SQL TYPE IS XML AS CLOB(n) <hostvar_name> to define a CLOB host variable that contains XML data encoded in the CCSID specified by the SQL_XML_DATA_CCSID QAQQINI file option.
- SQL TYPE IS XML AS DBCLOB(n) <hostvar_name> to define a DBCLOB host variable that contains XML data. It is encoded in the CCSID specified by the SQL_XML_DATA_CCSID QAQQINI file option if the option is UCS-2 or UTF-16, otherwise the default CCSID is UTF-16.
- SQL TYPE IS XML AS BLOB(n) <hostvar_name> to define a BLOB host variable that contains XML data internally encoded.
- SQL TYPE IS XML AS LOCATOR <hostvar_name> to define a locator that contains XML data.
- SQL TYPE IS XML AS CLOB_FILE <hostvar_name> to define a CLOB file that contains XML data encoded in the file CCSID.
- SQL TYPE IS XML AS DBCLOB_FILE <hostvar_name> to define a DBCLOB file that contains XML data encoded in the application double-byte default CCSID.
- SQL TYPE IS XML AS BLOB_FILE <hostvar_name> to define a BLOB file that contains XML data internally encoded.

**Example: Referencing XML host variables in embedded SQL applications:**

The following sample applications demonstrate how to reference XML host variables in C and COBOL.

**Embedded SQL C application:**
```
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;
// using XML AS CLOB host variable
// The XML value written to xmlBuf will be prefixed by an XML declaration
// similar to: <?xml version = "1.0" encoding = "UTF-8"?>
// Note: The encoding name will depend upon the SQL_XML_DATA_CCSID QAQQINI setting
EXEC SQL SELECT xmlCol INTO :xmlBuf
  FROM myTable
  WHERE id = '001';
EXEC SQL UPDATE myTable
  SET xmlCol = :xmlBuf
  WHERE id = '001';

// using XML AS BLOB host variable
// 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';

// using CLOB host variable
// The output will be encoded in the application single byte default CCSID,
// 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';
```

**Embedded SQL COBOL application:**

```
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 XML AS BLOB(5K).
EXEC SQL END DECLARE SECTION END-EXEC.

* using XML AS CLOB host variable
EXEC SQL SELECT xmlCol INTO :xmlBuf
  FROM myTable
  WHERE id = '001'.
EXEC SQL UPDATE myTable
  SET xmlCol = :xmlBuf
  WHERE id = '001'.

* using XML AS BLOB host variable
EXEC SQL SELECT xmlCol INTO :xmlblob
  FROM myTable
  WHERE id = '001'.
EXEC SQL UPDATE myTable
  SET xmlCol = :xmlblob
  WHERE id = '001'.

* using CLOB host variable
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'.
```

**Recommendations for developing embedded SQL applications with XML:**

The following recommendations and restrictions apply to using XML 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 host variable CCSID 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.

- When using static SQL, where an XML data type is expected for input, the use of CHAR, VARCHAR, CLOB, DBCLOB, 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.

**Identifying XML values in an SQLDA:**

To indicate that a base type holds XML data, the sqlname field of the SQLVAR must be updated.
- `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 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.

## Java
Java and XML

**XML data in JDBC applications:**

In JDBC applications, you can store data in XML columns and retrieve data from XML columns.

In database tables, the XML built-in data type is used to store XML data in a column.

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.

JDBC 4.0 `java.sql.SQLXML` objects can be used to retrieve and update data in XML columns. Invocations of metadata methods, such as ResultSetMetaData.getColumnTypeName return the integer value `java.sql.Types.SQLXML` for an XML column type.

*XML column updates in JDBC applications:*

When you update or insert data into XML columns of a database table, the input data in your JDBC applications must be in the serialized string format.

The following table lists the methods and corresponding input data types that you can use to put data in XML columns.

*Table 2. Methods and data types for updating XML columns*

| Method | Input data type |
|---|---|
| PreparedStatement.setAsciiStream | InputStream |
| PreparedStatement.setBinaryStream | InputStream |

| *Table 2. Methods and data types for updating XML columns  (continued)*

| Method | Input data type |
|---|---|
| PreparedStatement.setBlob | Blob |
| PreparedStatement.setBytes | byte[] |
| PreparedStatement.setCharacterStream | Reader |
| PreparedStatement.setClob | Clob |
| PreparedStatement.setObject | byte[], Blob, Clob, SQLXML, InputStream, Reader, String |
| PreparedStatement.setString | String |

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 data source 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 data source as character data, but the data contains encoding information. The data source handles incompatibilities between internal and external encoding by generating an error if the external and internal encoding are incompatible.

Data in XML columns is stored in the XML column CCSID. The database source handles conversion of the data from its internal or external encoding to the XML column CCSID.

The following example demonstrates inserting data from an SQLXML object into an XML column. The data is String data, so the database source treats the data as externally encoded.

```
public void insertSQLXML()
{
  Connection con = DriverManager.getConnection(url);
   SQLXML info = con.createSQLXML;
                 // Create an SQLXML object
  PreparedStatement insertStmt = null;
  String infoData =
    "<customerinfo xmlns=""http://posample.org"" " +
    "Cid=""1000"" xmlns=""http://posample.org"">...</customerinfo>";
  cid.setString(cidData);
                // Populate the SQLXML object
  int cid = 1000;
  try {
     sqls = "INSERT INTO CUSTOMER (CID, INFO) VALUES (?, ?)";
     insertStmt = con.prepareStatement(sqls);
     insertStmt.setInt(1, cid);
     insertStmt.setSQLXML(2, info);
              // Assign the SQLXML object value
          // to an input parameter
  if (insertStmt.executeUpdate() != 1) {
     System.out.println("insertSQLXML: No record inserted.");
  }
 }
  catch (IOException ioe) {
   ioe.printStackTrace();
 }
  catch (SQLException sqle) {
     System.out.println("insertSQLXML: SQL Exception: " +
       sqle.getMessage());
     System.out.println("insertSQLXML: SQL State: " +
       sqle.getSQLState());
```

```
         System.out.println("insertSQLXML: SQL Error Code: " +
            sqle.getErrorCode());
      }
   }
}
```

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());
   }
}
```

*XML data retrieval in JDBC applications:*

In JDBC applications, you use ResultSet.getXXX or ResultSet.getObject methods to retrieve data from XML columns.

When you retrieve data from XML columns of a DB2 table, the output data is in the serialized string format.

You can use one of the following techniques to retrieve XML data:
- Use the ResultSet.getSQLXML method to retrieve the data. Then use a SQLXML.getXXX method to retrieve the data into a compatible output data type. The SQLXML.getBinaryStream method adds an XML declaration with encoding specification to the output data. The SQLXML.getString and SQLXML.getCharacterStream methods do not add the XML declaration.
- Use a ResultSet.getXXX method other than ResultSet.getObject to retrieve the data into a compatible data type.

The following table lists the ResultSet methods and corresponding output data types for retrieving XML data.

*Table 3. ResultSet methods and data types for retrieving XML data*

| Method | Output data type |
|---|---|
| ResultSet.getAsciiStream | InputStream |

*Table 3. ResultSet methods and data types for retrieving XML data  (continued)*

| Method | Output data type |
|---|---|
| ResultSet.getBinaryStream | InputStream |
| ResultSet.getBytes | byte[] |
| ResultSet.getCharacterStream | Reader |
| ResultSet.getSQLXML | SQLXML |
| ResultSet.getString | String |

The following table lists the methods that you can call to retrieve data from a java.sql.SQLXML object, and the corresponding output data types and type of encoding in the XML declarations.

*Table 4. SQLXML and DB2Xml methods, data types, and added encoding specifications*

| Method | Output data type | Type of XML internal encoding declaration added |
|---|---|---|
| SQLXML.getBinaryStream | InputStream | XML column CCSID encoding |
| SQLXML.getCharacterStream | Reader | None |
| SQLXML.getSource | Source | None |
| SQLXML.getString | String | None |

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.

The following example demonstrates retrieving data from an XML column into an SQLXML object, and then using the SQLXML.getString method to retrieve the data into a string.

```
public void fetchToSQLXML()
   {
     System.out.println(">> fetchToSQLXML: Get XML data as an SQLXML object " +
     "using getSQLXML");
     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 the integer java.sql.Types.OTHER
     ResultSetMetaData meta = rs.getMetaData();
     String colType = meta.getColumnType(1);
     System.out.println("fetchToSQLXML: Column type = " + colType);
     while (rs.next()) {
       // Retrieve the XML data with getSQLXML.
       // Then write it to a string with
       // explicit internal ISO-10646-UCS-2 encoding.
       java.sql.SQLXML xml = rs.getSQLXML(1);
       System.out.println (xml.getString());
     }
     rs.close();
     }
     catch (SQLException sqle) {
       System.out.println("fetchToSQLXML: SQL Exception: " +
         sqle.getMessage());
```

```
|        System.out.println("fetchToSQLXML: SQL State: " +
|          sqle.getSQLState());
|        System.out.println("fetchToSQLXML: SQL Error Code: " +
|          sqle.getErrorCode());
|     }
| }
```

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 the integer java.sql.Types.OTHER
        ResultSetMetaData meta = rs.getMetaData();
        String colType = meta.getColumnType(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());
     }
}
```

*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 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 java.sql.SQLXML type. To register XML output parameters, register the parameters as the java.sql.Types.SQLXML type.

**Example**: JDBC program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter. This example requires JDBC 4.0.

```
java.sql.SQLXML in_xml = xmlvar;
java.sql.SQLXML out_xml = null;
java.sql.SQLXML 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.setObject (3, inout_xml);  // Set inout parameter
cstmt.registerOutParameter (2, java.sql.Types.SQLXML);
// Register out and input parameters
cstmt.registerOutParameter (3, java.sql.Types.SQLXML);
cstmt.executeUpdate();            // Call the stored procedure
out_xml = cstmt.getSQLXML(2);    // Get the OUT parameter value
inout_xml = cstmt.getSQLXML(3); // Get the INOUT parameter value
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value ");
MyUtilities.printString(out_xml.getString());
                                    // Use the SQLXML.getString
                                    // method to convert the out_xml
                                    // value to a string for printing.
                                    // Call a user-defined method called
                                    // printString (not shown) to print
                                    // the value.
System.out.println("INOUT parameter value "); MyUtilities.printString(inout_xml.getString());
                                    // Use the SQLXML.getString
                                    // method to convert the inout_xml
                                    // value to a string for printing.
                                    // Call a user-defined method called
                                    // printString (not shown) to print
                                    // the value.
```

**Example**: SQLJ program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter. This example requires JDBC 4.0.

```
java.sql.SQLXML in_xml = xmlvar;
java.sql.SQLXML out_xml = null;
java.sql.SQLXML 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 ");
MyUtilities.printString(out_xml.getString());
                                    // Use the SQLXML.getString
                                    // method toconvert the out_xml value
                                    // to a string for printing.
                                    // Call a user-defined method called
                                    // printString (not shown) to print
                                    // the value.
System.out.println("INOUT parameter value "); MyUtilities.printString(inout_xml.getString());
                                    // Use the SQLXML.getString
                                    // method to convert the inout_xml
                                    // value to a string for printing.
                                    // Call a user-defined method called
                                    // printString (not shown) to print
                                    // the value.
```

**XML data in SQLJ applications:**

In SQLJ applications, you can store data in XML columns and retrieve data from XML columns.

In DB2 tables, the XML built-in data type is used to store XML data in a column.

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.

JDBC 4.0 java.sql.SQLXML objects can be used to retrieve and update data in XML columns. Invocations of metadata methods, such as ResultSetMetaData.getColumnTypeName return the integer value `java.sql.Types.SQLXML` for an XML column type.

*XML column updates in SQLJ applications:*

When you update or insert data into XML columns of a table in an SQLJ application, the input data must be in the serialized string format.

The host expression data types that you can use to update XML columns are:
- java.sql.SQLXML (requires SQLJ Version 4.0 or later)
- String
- byte
- Blob
- Clob
- sqlj.runtime.AsciiStream
- sqlj.runtime.BinaryStream
- sqlj.runtime.CharacterStream

For stream types, you need to use an sqlj.runtime.*type*Stream 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 data source 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 data source as character data, but the data contains encoding information. The data source handles incompatibilities between internal and external encoding by generating an error if the external and internal encoding are incompatible.

Data in XML columns is stored in the XML column CCSID.

Suppose that you use the following statement to insert data from String host expression xmlString into an XML column in a table. xmlString is a character type, so its external encoding is used.
```
#sql [ctx] {INSERT INTO CUSTACC VALUES (1, :xmlString)};
```

**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.

```
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.

Suppose that you retrieve a document from an XML column into a java.sql.SQLXML host expression, and insert the data into an XML column in a table.

```
java.sql.ResultSet rs = s.executeQuery ("SELECT * FROM CUSTACC");
rs.next();
java.sql.SQLXML xmlObject = (java.sql.SQLXML)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 database table in an SQLJ application, the output data must be explicitly or implicitly serialized.

The host expression or iterator data types that you can use to retrieve data from XML columns are:
- java.sql.SQLXML (SQLJ Version 4.0)
- 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 java.sql.SQLXML or 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.

The following table lists the methods that you can call to retrieve data from a java.sql.SQLXML or a com.ibm.db2.jcc.DB2Xml object, and the corresponding output data types and type of encoding in the XML declarations.

*Table 5. SQLXML and DB2Xml methods, data types, and added encoding specifications*

| Method | Output data type | Type of XML internal encoding declaration added |
|--------|-----------------|-------------------------------------------------|
| SQLXML.getBinaryStream | InputStream | XML column CCSID encoding |
| SQLXML.getCharacterStream | Reader | None |
| SQLXML.getSource | Source | None |
| SQLXML.getString | String | None |

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.

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 and returned without any XML declaration.

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 no data conversion from UTF-8 encoding occurs, and the data is returned without any XML declaration.

Suppose that you retrieve a document from an XML column into a java.sql.SQLXML host expression, but you need the data in a binary stream.

```
#sql iterator SqlXmlIter (int, java.sql.SQLXML);
SqlXmlIter SQLXMLiter = null;
java.sql.SQLXML outSqlXml = null;
#sql [ctx] SqlXmlIter = {SELECT c1, CADOC from CUSTACC};
#sql {FETCH :SqlXmlIter INTO :row, :outSqlXml};
java.io.InputStream XmlStream = outSqlXml.getBinaryStream();
```

The FETCH statement retrieves the data into the SQLXML object in UTF-8 encoding. The SQLXML.getBinaryStream stores the data in a binary stream.

## Routines

Routines and XML

**XML 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.

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;


     /* insert the value of parm1 into table T1 */
     INSERT INTO T1 VALUES(parm1);

   /* 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, parm1 and parm2. parm1 is of the XML data type. Within the SQL procedure an XML variable
is declared named var1.

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.

*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 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.

**XML data type support in external routines:**

External procedures and functions written in programming languages that support parameters and
variables of data type XML:

These programming languages include:
- ILE RPG
- ILE COBOL
- C
- C++
- Java

XML data type values are represented in external routine code in the same way as character, graphic, and
binary 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 character, graphic, or binary data type. The size of the character, graphic, or
binary 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 'mylib1.myfunc'
```

Within external routine code, XML parameter and variable values are accessed, set, and modified in the same way as in database applications.

*Example: XML support in Java (JDBC) procedure:*

Once the basics of Java procedures, programming in Java using the JDBC application programming interface (API) are understood, you can start creating and using Java procedures that query XML data.

This example of a Java procedure illustrates:
- CREATE PROCEDURE statement for a parameter style JAVA procedure
- Source code for a parameter style JAVA procedure
- Input and output parameters of data type XML

**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:

**Java external code file format**
```
using System;
import java.lang.*;
import java.io.*;
import java.sql.*;
import java.util.*;

  public class stpclass
  {       ...
   // Java procedure implementations
        ...
  }
```

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: 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, queriesXML data using both an SQL statement and an XQuery expression, and sets two output parameters, outXML1, and outXML2.

**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 'stpclass.xmlProc1';

public void xmlProc1(int inNum,
                     CLOB  inXML ,
                     CLOB [] out1XML,
                     )
throws Exception
{
    Connection con = DriverManager.getConnection("jdbc:default:connection");

    // Insert data including the XML parameter value into a table
    String query = "INSERT INTO xmlDataTable (num, inXML ) VALUES ( ?, ? )" ;
    String xmlString = inXML.getCharacterStream() ;

    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 xmlDataTable WHERE num = ? " ;

    stmt = con.prepareStatement(query);
    stmt.setInt(1, inNum);
    ResultSet rs = stmt.executeQuery();

    if ( rs.next() )
    { out1Xml[0] = (CLOB) rs.getObject(1); }

    rs.close() ;
    stmt.close();

    return ;
}
```

*Example: XML support in C procedure:*

Once the basics of procedures, the essentials of C routines 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.

**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 `xmlDataTable`. Then an XML value is retrieved using an SQL statement. The retrieved XML value is assigned to the out1XML parameter. No result sets are returned.

```
CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER,
                            IN inXML XML as CLOB (1K),
                            OUT out1XML XML as CLOB (1K)
                          )
LANGUAGE C
PARAMETER STYLE SQL
DYNAMIC RESULT SETS 0
DETERMINISTIC
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME '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 SQL
//*************************************************************************
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlca.h>
#include <sqludf.h>
#include <sql.h>
#include <memory.h>

#ifdef __cplusplus
extern "C"
#endif
SQL_API_RC SQL_API_FN testSecA1(sqlint32* inNum,
                                SQLUDF_CLOB* inXML,
                                SQLUDF_CLOB* out1XML,
                                SQLUDF_NULLIND *inNum_ind,
                                SQLUDF_NULLIND *inXML_ind,
                                SQLUDF_NULLIND *out1XML_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;
 EXEC SQL END DECLARE SECTION;

 /* Check null indicators for input parameters */
 if ((*inNum_ind &lt; 0) || (*inXML_ind &lt; 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 xmlDataTable (num, xdata) VALUES (:hvNum1, :hvXML1);

/* Execute SQL statement */
EXEC SQL
  SELECT xdata INTO :hvXML2
    FROM xmlDataTable
      WHERE num = :hvNum1;

exit:

/* Set output return code */
*outReturnCode = sqlca.sqlcode;
*outReturnCode_ind = 0;

return 0;
}
```

# 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 host variable declared CCSID.
- 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. Externally encoded data sent to a DB2 database is checked by the database manager for internal encoding.

The effective CCSID that is associated with the internal encoding must match the external encoding; otherwise, an error occurs.

## Encoding considerations when storing or passing XML data

XML data must be encoded properly to be stored in a DB2 table. Encoding must be considered when the data is retrieved from the table and used with DB2 stored procedures or user-defined functions, or when used with external Java applications.

**Encoding considerations for input of XML data to a database:**

The internal and external encoding must be considered when storing XML data in a DB2 table.

The following rules need to be observed:

- 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.
- 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.

**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.

**Encoding considerations for passing XML data in routine parameters:**

In a DB2 database system, several XML data types are available for parameters in a stored procedure or user-defined function definition.

The following XML data types are available:

**XML**    For SQL procedures and functions.

**XML AS**
>        For external procedures and external user-defined functions.

Data in XML AS parameters is subject to character conversion. 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 CCSID conversions may occur, which can make the encoding information inaccurate. If the data is further parsed in the application, data corruption can result.

**Encoding considerations for XML data in JDBC and SQLJ applications:**

Typically, there are fewer XML encoding considerations for Java applications than for 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 CCSID 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 CCSID) to the XML column CCSID 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.

  For an explicit XMLSERIALIZE function with INCLUDING XMLDECLARATION, the database server adds encoding, and the JDBC driver does not modify it.
- 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.

## Effects of XML encoding and serialization on data conversion
The method of specifying the encoding of XML data, either internally or externally, and the method of XML serialization affect the conversion of XML data when passing the data between a database and an application.

**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 |
| Host variable data type | Binary |
| Host variable declared CCSID | 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 |
| Host variable data type | Binary |
| Host variable declared CCSID | 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 server converts the data from UTF-16 to UTF-8 when it performs the XML parse for storage in a UTF-8 XML column.

*Data loss:* None.

*Truncation:* None.

**Scenario 3**

| Encoding source | Value |
| --- | --- |
| Data encoding | ISO-8859-1 input data containing an XML encoding declaration |

| Encoding source | Value |
| --- | --- |
| Host variable data type | Binary |
| Host variable declared CCSID | 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 a UTF-8 XML column.

*Data loss:* None.

*Truncation:* None.

**Scenario 4**

| Encoding source | Value |
| --- | --- |
| Data encoding | Shift_JIS input data containing an XML encoding declaration |
| Host variable data type | Binary |
| Host variable declared CCSID | 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 a UTF-8 XML column.

*Data loss:* None.

*Truncation:* None.

**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 CCSID conversion during input to a database.

Only scenario 1 and scenario 2 apply to Java because the application code page for Java is always Unicode because character strings in Java are always Unicode.

**Scenario 1**

| Encoding source | Value |
| --- | --- |
| Data encoding | UTF-8 Unicode input data, with or without an appropriate encoding declaration or BOM |
| Host variable data type | Character |
| Host variable declared CCSID | 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 |
| Host variable data type | Graphic |
| Host variable declared CCSID | 1200 or 13488 |

*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 a UTF-8 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 |
| Host variable data type | Character |
| Host variable declared CCSID | 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 a UTF-8 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 |
| Host variable data type | Graphic |
| Host variable declared CCSID | 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 a UTF-8 XML column.

*Data loss:* None.

*Truncation:* None.

**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 applications, because the application code page for Java applications is always Unicode because character strings in Java are always Unicode.

**Scenario 1**

| Encoding source | Value |
| --- | --- |
| Target data encoding | UTF-8 Unicode |
| Target host variable data type | Binary |
| Host variable declared CCSID | 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 applications, the data is prefixed by the following XML declaration:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

For Java applications, no encoding declaration is added, unless you cast the data as the `com.ibm.db2.jcc.DB2Xml` type, and use a `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 host variable data type | Graphic |
| Host variable declared CCSID | 1200 or 13488 |

*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 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 host variable data type | Character |
| Host variable declared CCSID | 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 host variable data type | Graphic |
| Host variable declared CCSID | 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" ?>
```

**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 host variable data type | Binary |
| Host variable declared CCSID | 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 host variable data type | Graphic |
| Host variable declared CCSID | 1200 or 13488 |

*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 host variable data type | Character |
| Host variable declared CCSID | 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 host variable data type | Graphic |
| Host variable declared CCSID | 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.

## 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.

The QlgCvtTextDescToDesc API is used for mapping the IANA encoding name to the CCSID.

## Mappings of CCSIDs to encoding names for serialized XML output data

As part of an implicit or explicit XMLSERIALIZE operation, the DB2 database manager adds 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.

Where possible, the DB2 database manager chooses the IANA registry name for the CCSID, as prescribed by the XML standard. The QlgCvtTextDescToDesc API is used for mapping the CCSID to the IANA encoding name.

# Annotated XML schema decomposition

Annotated XML schema decomposition, also referred to as "decomposition" or "shredding," is the process of storing content from an XML document in columns of relational tables. Annotated XML schema decomposition operates based on annotations specified in an XML schema. After an XML document is decomposed, the inserted data has the SQL data type of the column that it is 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
* 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.

After the successful registration of the annotated schema, decomposition can be performed by calling the decomposition stored procedure.

The data from the XML document is always validated during decomposition. If information in an XML document does not comply with its specification in an XML schema then the data is not inserted into the table.

# 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.

To decompose XML documents using annotated XML schemas:

1. Annotate the schema documents with XML decomposition annotations.
2. Register the schema documents and enable the schema for decomposition.
3. If any of the registered schema documents that belong to the XML schema have changed, then all XML schema documents for this XML schema must be registered again and the XML schema must be enabled for decomposition again.
4. Decompose the XML document by calling the SYSPROC.XDBDECOMPXML stored procedure.

# Registering and enabling XML schemas for decomposition

Once an annotated schema has been successfully registered and enabled for decomposition, you can use it to decompose XML documents.

* Ensure that at least one element or attribute declaration in the XML schema is annotated with an XML decomposition annotation. This annotated element or attributes must be a descendant of, or itself be, a global element of complex type.

To register and enable XML schemas for decomposition:

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 `issuedfordecomposition` parameter set to 1.

# Sources for annotated XML schema decomposition

Annotated XML schema decomposition can be performed on elements or attributes in an XML document, and on the results of the db2-xdb:contentHandling or db2-xdb:expression annotations.

Annotated XML schema decomposition supports decomposition of the following types of content:
- The value of an attribute or element in the XML document.
- The value of an element in the XML document, where the exact content depends on the value of the <db2-xdb:contentHandling> annotation. <db2-xdb:contentHandling> values are:

  **text**    Character data from the element, but not from its descendants

  **stringValue**
          Character data from the element and its descendants

  **serializedSubtree**
          Markup of all content between the element's start tag and end tag
- Values that are generated through the db2-xdb:expression annotation:
  - A value that is 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.
  - A constant.

  An expression that is specified through the db2-xdb:expression is invoked once for every element or attribute with which it is associated.

# XML decomposition annotations

Annotated XML schema decomposition 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. Decomposition processing 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 recognized by the decomposition process only 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 referred to only 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.

## Specification and scope of XML decomposition annotations

You can specify annotations for decomposition as element or attribute declarations in an XML schema document.

You can do that in either of the following ways:
- As simple attributes in element or attribute declarations
- As structured (complex) child elements of simple element or attribute declarations

## 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. 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".

# 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 6. 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 7. 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 |
| Specify ordering dependencies between target tables | db2-xdb:rowSetOperationOrder, db2-xdb:rowSet, db2-xdb:order |

*Table 8. 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 |
| 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 |

# 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.

## 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>
```

## | db2-xdb:rowSet decomposition annotation
| The db2-xdb:rowSet annotation specifies an XML element or attribute mapping to a target base table.

| **Annotation type**

| Attribute of <xs:element> or <xs:attribute>, or child element of <db2-xdb:rowSetMapping> or
| <db2-xdb:order>.

## 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>
  ```
- ```
  <db2-xdb:order>
     <db2-xdb:rowSet>
  value</db2-xdb:rowSet>
     ...
  </db2-xdb:order>
  ```

## 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**

There are two ways of using db2-xdb:rowSet listed above.

**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 9. 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 10. ALLBOOKS

| ISBN | TITLE | AUTHORID |
|------|-------|----------|
| 1-11-111111-1 | My First XML Book | 22 |
| 0-11-011111-0 | Programming with XML | 435 |

## 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>.

## Annotation type

Global child element of <xs:appinfo> (which is a child element of <xs:annotation>)

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 paths are mapped to the same column of a table.
- 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 temporary or materialized query tables, are not supported for this mapping. Views and table aliases are not 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 11. AUTHORSCONTACT*

| AUTHID | EMAILADDR |
|--------|-----------|
| 22 | author22@anyemail.com |
| 0800 | author800@email.com |

## 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.

### 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.

### Details
- Undelimited column names are case-insensitive.

- When special characters, such as double quotation marks ("), ampersands (&), or less-than signs (<) are part of SQL identifiers, those special characters need to be replaced by their equivalent XML notations. For example, replace " with &quot;, & with &amp;, and < with &lt;.
- db2-xdb:column is an optional child element of db2-xdb:rowSetMapping if the db2-xdb:locationPath annotation is present.

**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 12. BOOKCONTENTS*

| ISBN | CHPTNUM | CHPTTITLE | CHPTCONTENT |
|---|---|---|---|
| 1-11-111111-1 | 1 | Introduction to XML | XML is fun... |

|   *Table 12. BOOKCONTENTS  (continued)*

| ISBN | CHPTNUM | CHPTTITLE | CHPTCONTENT |
|------|---------|-----------|-------------|
| 1-11-111111-1 | 2 | XML and Databases | XML can be used with... |
| ... | ... | ... | ... |
| 1-11-111111-1 | 10 | Further Reading | Recommended tutorials... |

## db2-xdb:locationPath decomposition annotation

The db2-xdb:locationPath annotation maps an XML element or attribute to different table and column pairs, depending on the path of the element or attribute.

The db2-xdb:locationPath annotation is used to describe the mappings for elements or attributes that are either declared globally or as part of :

- A named model group
- A named attribute group
- A global complex type declaration
- A global element or attribute of simple or complex type

### 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:
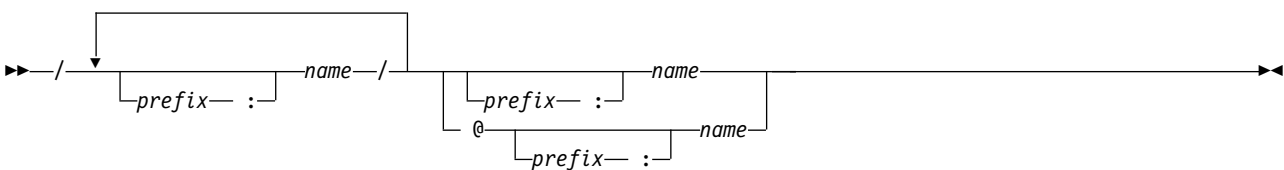


**name**   An element or attribute name.

**prefix**   A namespace prefix.

The namespace prefix must be associated with a namespace in the schema document that contains the annotation with db2-xdb:locationPath. A namespace prefix binding can be created by adding a namespace declaration to the <xs:schema> element of the schema document.

## Details

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 other paths (for example: <xs:element ref="abc">). Because annotations cannot be specified directly on references, the annotations must be specified on the corresponding global element or attribute declaration. A global 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 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 rowSt and column pair for each path. The same db2-xdb:locationPath value can be used for different rowSet and column pairs.

When a default namespace and the *attributeFormDefault ="unqualified"* are specified, decomposition annotations for unqualified attributes are ignored because the annotation processing treats the attribute as it if belongs to the default namespace. However, the *attributeFormDetail = "unqualified"* setting 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 13. BOOKS*

| ISBN | TITLE | CONTENT |
|------|-------|---------|
| NULL | My First XML Book | NULL |

*Table 14. BOOKCONTENTS*

| ISBN | CHPTNUM | CHPTTITLE | CHPTCONTENT |
|------|---------|-----------|-------------|
| NULL | NULL | Introduction to XML | NULL |
| NULL | NULL | XML and Databases | NULL |
| ... | ... | ... | ... |
| NULL | NULL | Further Reading | NULL |

## 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.

## 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:**
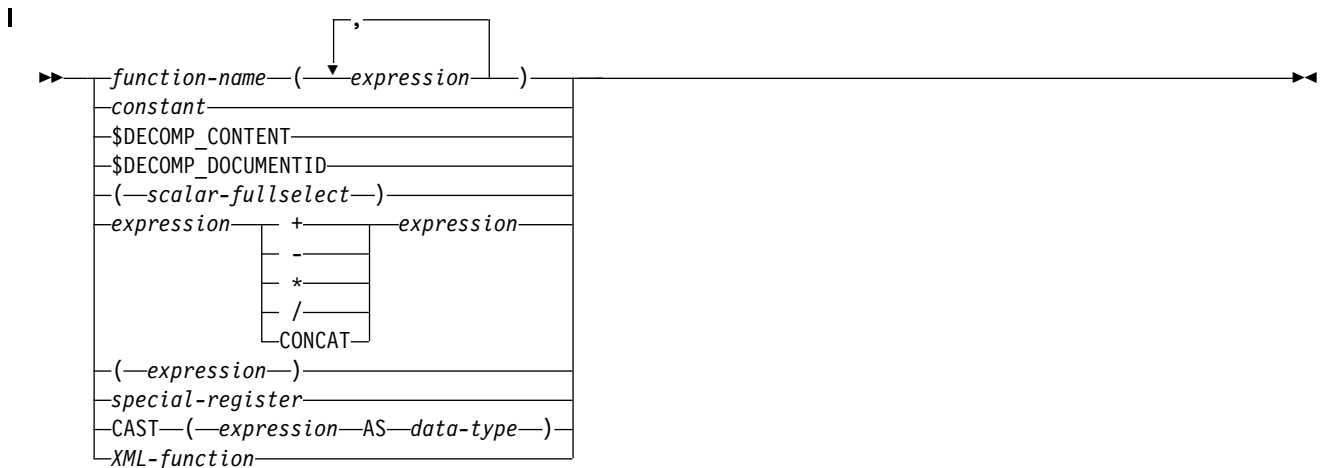


## Details

The db2-xdb:expression annotation enables you to specify a customized expression, which is applied to the content of the XML element or attribute that is annotated when $DECOMP_CONTENT is used. The result of evaluating this expression is inserted into the column that is specified during decomposition.

db2-xdb:expression is 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.

Schema names, table names, and column names in the db2-xdb:expression annotation must use SQL naming for qualification and are case sensitive only if the names are delimited.

**function ( expression-list )**
>A built-in or user-defined SQL scalar function. A scalar function returns a single value (possibly null).

**constant**
>A value that is a string constant or a numeric constant.

**$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.

**$DECOMP_DOCUMENTID**
>The string value specified in the *documentid* input parameter of the XDBDECOMPXML stored procedure, which identifies the XML document being decomposed.

**( 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.

**( 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.

**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).

**XML-function**
>Any supported SQL/XML function.

## Example

This example demonstrates the types of operations that you can perform with db2-xdb:expression annotations. The example also demonstrates cases in which XML input must be cast to an SQL type.

The input XML document looks like this:

```
<num>
1000.99
</num>
```

The annotated XML schema produces a row for a table with the following definition:

```
CREATE TABLE TAB1
    (ASIS    DECIMAL(9,2),
     ADD     INTEGER,
     TAX     INTEGER,
     STR     VARCHAR(25),
     LITERAL DECIMAL(5,1)
     SELECT  VARCHAR(25))
```

The annotated XML schema uses a table with the following definition:

```
CREATE TABLE SCH1.TAB2
    (ID   INTEGER,
     COL1 VARCHAR(25))
```

Table SCH1.TAB2 contains one row with the values (100, 'TAB2COL1VAL').

```
<xs:element name="num" type="xs:double">
  <xs:annotation>
    <xs:appinfo>
      <xdb:rowSetMapping>
        <xdb:rowSet>TAB1</xdb:rowSet>
        <xdb:column>ASIS</xdb:column>
      </xdb:rowSetMapping>
      <xdb:rowSetMapping>
        <xdb:rowSet>TAB1</xdb:rowSet>
        <xdb:column>ADD</xdb:column>
        <xdb:expression>
          CAST(XDB.ADD(1234,CAST($DECOMP_CONTENT AS INTEGER)) AS INTEGER)
        </xdb:expression>
      </xdb:rowSetMapping>
      <xdb:rowSetMapping>
        <xdb:rowSet>TAB1</xdb:rowSet>
        <xdb:column>TAX</xdb:column>
        <xdb:expression>
          CAST(XDB.TAX(CAST($DECOMP_CONTENT AS DOUBLE)) AS INTEGER)
        </xdb:expression>
      </xdb:rowSetMapping>
      <xdb:rowSetMapping>
        <xdb:rowSet>TAB1</xdb:rowSet>
        <xdb:column>STR</xdb:column>
        <xdb:expression>
          CAST($DECOMP_CONTENT AS VARCHAR(25))
        </xdb:expression>
      </xdb:rowSetMapping>
      <xdb:rowSetMapping>
        <xdb:rowSet>TAB1</xdb:rowSet>
        <xdb:column>LITERAL</xdb:column>
        <xdb:expression>32.3</xdb:expression>
      </xdb:rowSetMapping>
      <xdb:rowSetMapping>
        <xdb:rowSet>TAB1</xdb:rowSet>
        <xdb:column>SELECT</xdb:column>
        <xdb:expression>
          (SELECT "COL1" FROM "SCH1"."TAB2" WHERE "ID" = 100)
        </xdb:expression>
      </xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
</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.

*Table 15. TAB1*

| ASIS | ADD | TAX | STR | LITERAL | SELECT |
|------|-----|-----|-----|---------|--------|
| 1000.99 | 2234 | 300 | 1000.99 | 32.3 | TAB2COL1VAL |

## 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.

The condition is applied regardless of whether the annotation to which it belongs contains a column mapping.

### Annotation type

Attribute of <xs:element> or <xs:attribute>, or optional child element of <db2-xdb:rowSetMapping>.

### 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
* DISTINCT
* EXISTS
* IN
* LIKE
* NULL

The predicates must also consist of expressions that are supported by the db2-xdb:expression annotation, column names, or both.

### Details

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:condition="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 <b>, because <b> 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 <b>, "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 <b>.

```
<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 want to decompose only 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 16. 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.

## 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.

### 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 CCSID issues that should be considered. If the target column is of character or graphic types, the XML fragment is inserted in the column's CCSID. 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.

## 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 presented only 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 presented here
| only to show the boundaries and whitespaces of the inserted strings.

**db2-xdb:contentHandling="text"**

*Table 17. 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" | "<br><br>                    Escape characters such as <, >, and & ..." |
| ... | ... | ... | ... |
| 1-11-111111-1 | 10 | "Further Reading" | "Recommended tutorials..." |

Observe how the content of the <b> 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 18. 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" | "<br><br>                    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 <b> 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 19. BOOKCONTENTS*

| ISBN | CHPTNUM | CHPTTITLE | CHPTCONTENT |
|---|---|---|---|
| 1-11-111111-1 | 1 | "Introduction to XML" | "<paragraph>XML is <b>lots of</b> fun...</paragraph>" |
| 1-11-111111-1 | 2 | "XML and Databases" | "<paragraph><!-- Start of chapter -->XML can be used with...</paragraph>" |

| *Table 19. BOOKCONTENTS  (continued)*

| ISBN | CHPTNUM | CHPTTITLE | CHPTCONTENT |
|---|---|---|---|
| 1-11-111111-1 | 2 | "XML and Databases" | "<paragraph><?processInstr example?><br><br>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 <b> 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.

## 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).

### 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**
> Before the XML value is inserted into the target column, or is substituted for occurrences of $DECOMP_CONTENT that are in the same mapping as this db2-xdb:normalization annotation, the XML value is converted to its canonical form according to its XML schema type.

**original**
> Before the XML value is inserted into the target column, or is substituted for occurrences of $DECOMP_CONTENT that are in the same mapping as this db2-xdb:normalization annotation, no modification of the XML is done except for possible processing by an XML parser. This is the default.

**whitespaceStrip**

> Before the XML value is inserted into the target column, or is substituted for occurrences of $DECOMP_CONTENT that are in the same mapping as this db2-xdb:normalization annotation:

- All leading and trailing whitespace is removed from the XML value.
- Consecutive whitespace is collapsed into a single whitespace character.

## Details

db2-xdb:normalization is applicable when an element or attribute has one of the following atomic XML schema types:

- byte, unsigned byte
- integer, positiveInteger, negativeInteger, nonPositiveInteger, nonNegativeInteger
- int, unsignedInt
- long, unsignedLong
- short, unsignedShort
- decimal
- float
- double
- boolean
- time
- date
- dateTime

The target column must have one of the following data types:

- CHAR
- VARCHAR
- CLOB
- DBCLOB
- GRAPHIC
- VARGRAPHIC

db2-xdb:normalization will be ignored if specified for any other types.

## 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 20. AUTHORS*

| AUTHID | FIRSTNAME | SURNAME | ACTIVE | NUMBOOKS |
|--------|-----------|---------|--------|----------|
| 22 | Ann | __Brown_ | true | NULL |

The db2-xdb:normalization="whitespaceStrip" annotation on the ID attribute causes the leading and
trailing whitespace to be removed before the data is inserted into the AUTHORS table. The
db2-xdb:normalization.="canonical" annotation on the <activeStatus> element causes its boolean value to
be replaced with the canonical representation of that value before insertion into the AUTHORS table. The
element has a boolean type. The canonical representation of a boolean type is true or false.

## db2-xdb:order decomposition annotation
The db2-xdb:order annotation specifies the insertion order of rows among different tables.

### Annotation type

Child element of <db2-xdb:rowSetOperationOrder>.

### How to specify

db2-xdb:order is specified in the following way (where *value* represents a valid value for the annotation):
```
<xs:schema>
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetOperationOrder>
        <db2-xdb:order>
          <db2-xdb:rowSet>value</db2-xdb:rowSet>
          ...
        </db2-xdb:order>
      </db2-xdb:rowSetOperationOrder>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>
```

### Namespace

http://www.ibm.com/xmlns/prod/db2/xdb1

### Valid structure

The following are supported children elements of <db2-xdb:order>:

**db2-xdb:rowSet**
        Specifies an XML element or attribute mapping to a target base table.

## Details

The db2-xdb:order annotation is used to define the order of insertion of the rows belonging to a given rowSet, relative to the rows belonging to another rowSet. This enables XML data to be inserted into target tables in a way consistent with any referential integrity constraints defined on the tables as part of the relational schema. The number of db2-xdb:rowSet elements that can appear in db2-xdb:order element can be any number greater than 1.

All rows of a given rowSet RS1 are inserted before any rows belonging to another rowSet RS2 if RS1 is listed before RS2 within db2-xdb:order. Multiple instances of this element can be specified in order to define multiple insert order hierarchies. For rowSets that do not appear in any element, their rows may be inserted in any order, relative to the rows of any other rowSet. Also, the content of each <db2-xdb:rowSet> element must be either an explicitly defined rowSet or the name of an existing table for which no explicit rowSet declaration was made.

Multiple rowSet insertion hierarchies can be defined, though a rowSet can appear in only one instance of the <db2-xdb:order> element, and it can appear only once within that element.

For delimited SQL identifiers specified in the children elements, the quotation marks delimiter must be included in the character content and need not be escaped. The '&' and '<' characters used in SQL identifiers, however, must be escaped.

## Example

The following example demonstrates the use of the db2-xdb:order annotation.

```
<xs:schema>
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetOperationOrder>
        <db2-xdb:order>
          <db2-xdb:rowSet>CUSTOMER</db2-xdb:rowSet>
          <db2-xdb:rowSet>PURCHASE_ORDER</db2-xdb:rowSet>
        </db2-xdb:order>
        <db2-xdb:order>
          <db2-xdb:rowSet>ITEMS_MASTER</db2-xdb:rowSet>
          <db2-xdb:rowSet>PO_ITEMS</db2-xdb:rowSet>
        </db2-xdb:order>
      </db2-xdb:rowSetOperationOrder>
    </xs:appinfo>
  </xs:annotation>
</xs:schema>
```

Two disjoint hierarchies for order of insertion are specified in the above example. The first hierarchy specifies that all content for the CUSTOMER rowSet or table is inserted prior to any content collected for PURCHASE_ORDER, and the second hierarchy specifies that all content for the ITEMS_MASTER rowSet or table will be inserted before any content is inserted into PO_ITEMS. Note that the order between the two hierarchies is undefined. For example, any content for the PURCHASE_ORDER rowSet or table may be inserted before or after any content is inserted into ITEMS_MASTER

## 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.

## 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 values:

**0 or false**
> The value cannot be truncated before insertion, If the value is too long, an error occurs and the value is not inserted. This is the default.

**1 or true**
> The value can be truncated before insertion.

## 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" setting indicates that data truncation is allowed during insertion.

db2-xdb:truncate is applicable only for the following mappings:

| XML data type | Column data type |
|---|---|
| Any compatible type | CHAR<br>VARCHAR<br>CLOB<br>GRAPHIC<br>VARGRAPHIC<br>DBCLOB |
| xs:date | DATE |
| xs:time | TIME |
| xs:dateTime | TIMESTAMP |

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.

For an annotations that decompose XML datatime values into DATE, TIME, or TIMESTAMP columns, if the XML data can have a timezone, db2-xdb:truncate must be set to "true" or "1".

## 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 21. 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.

## db2-xdb:rowSetMapping decomposition annotation
The db2-xdb:rowSetMapping annotation maps 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.

## 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.

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>**
> Maps an XML element or attribute to a base table column. This element is optional if db2-xdb:condition or db2-xdb:locationPath annotation is present.

**<db2-xdb:expression>**
> Specifies a customized expression, the result of which is inserted into the table named by the db2-xdb:rowSet attribute. This element is optional.

**<db2-xdb:condition>**
> Specifies a condition or evaluation. This element is optional.

## Details

If db2-xdb:expression and db2-xdb:truncate are specified together, db2-xdb:truncate is ignored.

For mapping to a single table and column, specification of db2-xdb:rowSetMapping is equivalent to specifying a combination of db2-xdb:rowSet and db2-xdb:column annotations.

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 22. BOOKS*

| ISBN | TITLE | CONTENT |
|---|---|---|
| 1-11-111111-1 | NULL | NULL |

*Table 23. 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>
```

## db2-xdb:rowSetOperationOrder decomposition annotation

The db2-xdb:rowSetOperationOrder annotation is a parent for one or more db2-xdb:order elements. See the section for db2-xdb:order for details on usage in defining order of insertion of rows among different tables.

## Annotation type

Child element of <xs:appinfo> that is a child of a global <xs:annotation> element.

## How to specify

db2-xdb:rowSetOperationOrder is specified in the following way (where *value* represents a valid value for the annotation):

```
<xs:schema>
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetOperationOrder>
        <db2-xdb:order>
          <db2-xdb:rowSet>value</db2-xdb:rowSet>
          ...
        </db2-xdb:order>
      </db2-xdb:rowSetOperationOrder>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>
```

| http://www.ibm.com/xmlns/prod/db2/xdb1

| **Valid structure**

| The following are supported children elements of <db2-xdb:rowSetOperationOrder>:
| **db2-xdb:order**

| **Details**

| <db2-xdb:rowSetOperationOrder> groups together <db2-xdb:order> elements. Multiple instances of the
| child <db2-xdb:order> element can be present, allowing the definition of multiple insertion hierarchies.

| By allowing you to control the order in which contents of XML documents are inserted, the
| db2-xdb:rowSetOperationOrder and db2-xdb:order annotations together provide a way to ensure that the
| XML schema decomposition process respects any referential integrity constraints on target tables, as well
| as any other application requirements that rows of a table be inserted before rows of another table.

| The db2-xdb:rowSetOperationOrder annotation can appear only once in an XML schema.

| **Example**

| See "db2-xdb:order decomposition annotation" on page 95 for examples of specifying the order of rowSet
| insertion.

| # 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. The value is constructed
| according to the setting of the db2-xdb:contentHandling annotation. The value of
| $DECOMP_CONTENT has a character type.

| $DECOMP_CONTENT can be used to process the value of the mapped element or attribute,
| using customized expressions, rather than directly inserting that value.

| If db2-xdb:expression specifies $DECOMP_CONTENT and db2-xdb:normalization is specified in
| the same mapping, the $DECOMP_CONTENT value for db2-xdb:expression is normalized before
| it is passed to the expression for evaluation.

| **$DECOMP_DOCUMENTID**
| The string value specified in the *documentid* input parameter of the XDBDECOMPXML stored
| procedure, which identifies the XML document that is being decomposed. When the document is
| decomposed, the input value that is provided to the stored procedure is used as the value
| substituted for $DECOMP_DOCUMENTID.

| $DECOMP_DOCUMENTID can be used to insert unique identifiers that are not present in the
| XML document. Applications can pass uniquely generated document IDs into
| XDBDECOMPXML. These IDs can then be directly inserted into a table in the database. The IDs
| can also be passed into expressions that generate unique identifiers for elements or attributes.

# Treatment of CDATA sections in annotated XML schema decomposition

If elements that are annotated for decomposition contain CDATA sections, the decomposition process inserts the contests of the CDATA sections into tables, without the CDATA section delimiters ("<![CDATA[" and "]]>").

Carriage return and line feed pairs (U+000D and U+000A) or carriage returns (U+000D) within the CDATA section are replaced with line feeds (U+000A).

If the XML element declaration in the XML schema is annotated with the attribute db2-xdb:contentHandling="serializeSubtree", the contents of the CDATA section are inserted into tables with the following changes:

- CDATA section delimiters ("<![CDATA[" and "]]>") are removed.
- Each ampersand (&) in the CDATA section is replaced by the string &amp:.
- Each left angle bracket (<) in the CDATA section is replaced by the string &lt:.

The content of the CDATA section after decomposition is logically equivalent to the original content of the CDTATA section.

# 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 24. NULL handling for mapped elements*

| Condition | Empty string | NULL value |
|---|---|---|
| Element missing from document | | X |
| Element satisfies all of the following conditions:<br>• is present in the document<br>• contains the xsi:nil="true" or xsi:nil="1" attribute in the start tag | | X |
| Element satisfies all of the following conditions:<br>• is present and empty in the document<br>• does not contain the xsi:nil="true" or xsi:nil="1" attribute in the start tag<br>• is derived from or declared to be of list type, union type, complex type with mixed content, or the following atomic built-in types: xsd:string, xsd:normalizedString, xsd:token, xsd:hexBinary, xsd:base64Binary, xsd:anyURI, xsd:anySimpleType; any other types will result in an error. | X | |
| **Note:**<br>1. If a mapping involves the db2-xdb:condition or db2-xdb:expression annotations, then the empty string or NULL value (as shown in this table) is passed as the argument for expression evaluation.<br>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 25. 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) | | X |
| Attribute satisfies all of the following conditions:<br>• is present and empty in the document<br>• is derived from or declared to be of list type, union type, or the following atomic built-in types: xsd:string, xsd:normalizedString, xsd:token, xsd:hexBinary, xsd:base64Binary, xsd:anyURI, xsd:anySimpleType; any other types will result in an error. | X | |
| **Note:** If a mapping involves the db2-xdb:condition or db2-xdb:expression annotations, then the empty string or NULL value (as shown in this table) is passed as the argument for expression evaluation. | | |

# Checklist for annotated XML schema decomposition

Annotated XML schema decomposition can become complex. To make the task more manageable, you should take several things into consideration.

Annotated XML schema decomposition 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.
- 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.

# 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.

Common mapping scenarios:

## Annotations of derived complex types

XML schemas can contain complex types that are derived by restriction or extension (specified by `complexType` elements that include `restriction` or `extension` elements).

When you annotate those complex types for decomposition, you need to apply additional mappings.

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.

Complex types that are derived by restriction require that the common elements and attributes from the base type are repeated in the definition of the derived type. Therefore, decomposition annotations that are present in the base type must also be included in the derived type.

In the definition of complex types that are 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, you must add decomposition annotations to the base type to clearly differentiate the mappings of the base from the derived types.

Example: In the following XML schema document, outOfPrintBookType is derived by extension. It is mapped to a different table than its base type, which is bookType. 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, because these elements are involved in only 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>                                              Note 1
        <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:attribute name="title" type="xs:string"                Note 2a
                  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"                 Note 2b
                  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"                    Note 3
                   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 are decomposed into the BOOKS table, and values from the <outOfPrintBook> element will be decomposed into the OUTOFPRINT table.

Notes for the annotated XML schema document:

**1**     A <books> document can have two types of elements: a <book> element or an <outOfPrintBook> element. Information from the two types of elements is decomposed into different tables. The <outOfPrintBook> element definition has extensions to the <book> element definition.

**2a and 2b**
        In the XML schema, the title and isbn attributes have the same definition for <book> elements or <outOfPrintBook> elements, but in-print and out-of-print book information goes into different tables. Therefore, you need db2-xdb:locationPath annotations to distinguish between titles and ISBN numbers for in-print and out-of-print books.

**3**     The lastPublished and publisher elements are unique to <outOfPrintBook> elements, so no db2-xdb:locationPath annotation is necessary for them.

Suppose that you use this XML schema document to decompose the following 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 decomposition produces the following tables:

*Table 26. 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 27. 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 |

## Rowsets in annotated XML schema decomposition

The db2-xdb:rowSet annotation identifies the target table and row into which a value in an XML document is decomposed.

The value of the db2-xdb:rowSet annotation can be a table name or a rowset name.

The db2-xdb:rowSet annotation can be an attribute of an element or attribute declaration, or a child of the <db2-xdb:rowSetMapping> annotation.

In an XML schema, there can be multiple occurrences of an element or attribute, and all of those elements or attributes can have a db2-xdb:rowSet annotation with the same target table value. Each of those db2-xdb:rowSet annotations defines a row in the target table.

**Example:** Suppose that you want to decompose this document so that each book's isbn and title value is inserted into the ALLPUBLICATIONS table.

```
<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>Melanie Snowe</author>
      <publicationDate>1999</publicationDate>
   </childrensbook>
</publications>
```

You need to define the following rowsets:

- A rowset to group the isbn value for a textbook with its title
- A rowset to group the isbn value for a children's book with its title

The annotated XML schema look like this:

```
<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>                    Note 1
          <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"              Note 2a
                       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"Note 2b
                       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"                 Note 3a
                       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"Note 3b
                       db2-xdb:rowSet="childrens_rowSet" db2-xdb:column="PUBS_TITLE"/>
             </xs:complexType>
           </xs:element>
         </xs:sequence>
       </xs:complexType>
     </xs:element>
</xs:schema>
```

Notes for the annotated XML schema document:

| **1** In the global <xs:annotation>, the `textbk_rowSet` rowset and the `childrens_rowSet` rowset are
| declared for later reference in the XML schema. The <db2-xdb:name> annotation names the table
| (ALLPUBLICATIONS) to which the rowsets refer.

| **2a and 2b**
| The `textbk_rowSet` annotation is specified on the isbn element declaration and the title attribute
| declaration in the <textbook> element. This indicates that isbn and title information from
| <textbook> elements gets decomposed into a row of the ALLPUBLICATIONS table.

| **3a and 3b**
| The `childrens_rowSet` annotation is specified on the isbn element declaration and the title
| attribute declaration in the <childrensbook> element. This indicates that isbn and title information
| from <childrensbook> elements gets decomposed into a row of the ALLPUBLICATIONS table.

| The following table results from decomposing the previously shown document with the annotated XML
| schema:

| *Table 28. ALLPUBLICATIONS*

| ISBN | PUBS TITLE |
|---|---|
| 0-11-011111-0 | Programming with XML |
| 5-55-555555-5 | Children's Fables |

| The previous example 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 let you apply different transformations to the values that are being decomposed, depending on
| the values themselves.

| **Example:** The following two instances of an element named temperature have different attribute values:

```
| <temperature unit="Celsius">49</temperature>
| <temperature unit="Farenheit">49</temperature>
```

| If you decompose these values into the same table, you need to map all elements with the attribute
| unit="Celsius" to one rowset and all elements with the attribute unit="Farenheit" to another rowset. Then
| you can apply a conversion formula to values for one rowset so that all values are in the same
| temperature units before you insert the values into the table.

| The following annotated XML schema demonstrates this technique.

```
| <db2-xdb:name>TEMPERATURE_DATA</db2-xdb:name>
|   <db2-xdb:rowSet>temp_celcius</db2-xdb:rowSet>                                    Note 1
|   <db2-xdb:rowSet>temp_farenheit</db2-xdb:rowSet>
| </db2-xdb:table>
| ...
|
| <xs:element name="temperature">
|   <xs:annotation>
|     <xs:appinfo>
|       <db2-xdb:rowSetMapping>
|         <db2-xdb:rowSet>temp_celcius</db2-xdb:rowSet>
|         <db2-xdb:column>col1</db2-xdb:column>
|       </db2-xdb:rowSetMapping>
|       <db2-xdb:rowSetMapping>
|         <db2-xdb:rowSet>temp_farenheit</db2-xdb:rowSet>
|         <db2-xdb:column>col1</db2-xdb:column>
| <db2-xdb:expression>CAST(myudf_convertToCelcius(CAST($DECOMP_CONTENT AS FLOAT)) AS FLOAT)
```

```
| </db2-xdb:expression>                                                    Note 2
|       </db2-xdb:rowSetMapping>
|     </xs:appinfo>
|   </xs:annotation>
|   <xs:complexType>
|     <xs:simpleContent>
|       <xs:extension base="xs:string">
|         <xs:attribute name="unit" type="xs:string">
|           <xs:annotation>
|             <xs:appinfo>
|               <db2-xdb:rowSetMapping>                                    Note 3
|                 <db2-xdb:rowSet>temp_celcius</db2-xdb:rowSet>
|                 <db2-xdb:condition>$DECOMP_CONTENT = 'Celsius'</db2-xdb:condition>
|               </db2-xdb:rowSetMapping>
|               <db2-xdb:rowSetMapping>
|                 <db2-xdb:rowSet>temp_farenheit</db2-xdb:rowSet>
|                 <db2-xdb:condition>$DECOMP_CONTENT = 'Farenheit'</db2-xdb:condition>
|               </db2-xdb:rowSetMapping>
|             </xs:appinfo>
|           </xs:annotation>
|         </xs:attribute>
|       </xs:extension>
|     </xs:simpleContent>
|   </xs:complexType>
| </xs:element>
```

Notes for the annotated XML schema document:

**1**     In the global <xs:annotation>, the `temp_celcius` rowset and the `temp_farenheit` rowset are declared for later reference in the XML schema.

**2**     A conversion formula converts values in the `temp_farenheit` rowset to Celsius units so that all values are in Celsius units when you insert them into a table. The expression annotation must contain CAST specifications to cast the arguments and the return type of the function to the corresponding SQL data types with which the function is defined.

**3**     All elements with the attribute `unit="Celsius"` are mapped to the `temp_celcius` rowset, and all elements with the attribute `unit="Farenheit"` are mapped to the `temp_farenheit` rowset.

## 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 29. TEXTBOOKS*

| TITLE | DETAILS |
|-------|---------|
| Programming with XML | ```<textbook title="Programming with XML">```<br>```  <isbn>0-11-011111-0</isbn>```<br>```  <author>Mary Brown</author>```<br>```  <author>Alex Page</author>```<br>```  <publicationDate>2002</publicationDate>```<br>```  <university>University of London</university>```<br>```</textbook>``` |

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.

## 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. This example shows the simpler case of a one to one relationship between values in a rowSet.

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.

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 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 30. TEXTBOOKS*

| ISBN | TITLE | DATE |
|---|---|---|
| 0-11-011111-0 | Programming with XML | 2002 |

To map a value to a single table and column pair, you need to specify the table and column on the value that you want to map. Do that by using one of the following sets of annotations:

- db2-xdb:rowSet and db2-xdb:column as attributes of <xs:element> or <xs:attribute>
- <db2-xdb:rowSet> and <db2-xdb:column> as child elements of <db2-xdb:rowSetMapping>

The following annotated XML schema specifies the db2-xdb:rowSet and db2-xdb:column annotations as attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<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>"MYSCHEMA"</db2-xdb:defaultSQLSchema>
  </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" maxOccurs="1"
                        db2-xdb:rowSet="TEXTBOOKS"
                        db2-xdb:column="ISBN"/>
            <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
            <xs:element name="publicationDate" type="xs:gYear" maxOccurs="1"
                        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>
| </xs:schema>
```

The maxOccurs attribute for the elements that go into the rowset is 1, so the items that are mapped to the
TEXTBOOKS table have a one-to-one relationship with each other. Therefore, a single row is formed for
each instance of the <textbook> element.

## 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. This example shows the more complex case of a one to many
relationship between values in a rowSet.

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.

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 31. 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.

To map a value to a single table and column pair, you need to specify the table and column on the value
that you want to map. Do that by using one of the following sets of annotations:

- db2-xdb:rowSet and db2-xdb:column as attributes of <xs:element> or <xs:attribute>
- <db2-xdb:rowSet> and <db2-xdb:column> as child elements of <db2-xdb:rowSetMapping>

The following annotated XML schema specifies the <db2-xdb:rowSet> and <db2-xdb:column> annotations
as elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<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>"MYSCHEMA"</db2-xdb:defaultSQLSchema>
 </xs:appinfo>
</xs:annotation>

<xs:element name="textbook">
 <xs:complexType>
   <xs:sequence maxOccurs="unbounded">

     <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>
</xs:schema>
```

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.

## Decomposition annotation example: A value mapped to multiple tables

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 32. TEXTBOOKS*

| ISBN | TITLE |
|---|---|
| 0-11-011111-0 | Programming with XML |

*Table 33. 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<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>"MYSCHEMA"</db2-xdb:defaultSQLSchema>
  </xs:appinfo>
 </xs:annotation>

 <xs:element name="textbook">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">

      <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: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>
```

## 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, as shown in this example.

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 34. 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.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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>"MYSCHEMA"</db2-xdb:defaultSQLSchema>
    <db2-xdb:table>
      <db2-xdb:name>ALLPUBLICATIONS</db2-xdb:name>
      <db2-xdb:rowSet>testbk_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 maxOccurs="unbounded">

     <xs:element name="textbook">
       <xs:complexType>
         <xs:sequence maxOccurs="unbounded">
           <xs:element name="isbn" type="xs:string"
             db2-xdb:rowSet="testbk_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="testbk_rowSet"
             db2-xdb:column="PUBS_TITLE"/>
       </xs:complexType>
     </xs:element>

     <xs:element name="childrensbook">
       <xs:complexType>
         <xs:sequence maxOccurs="unbounded">
           <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.

## 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, as shown in this example.

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 35. 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<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>"MYSCHEMA"</db2-xdb:defaultSQLSchema>
    <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 maxOccurs="unbounded">

        <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: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: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>.

## XML schema to SQL types compatibility for annotated schema decomposition

Annotated XML schema decomposition supports the decomposition of XML values into columns that are
of a compatible SQL type.

The following tables list XML schema data types and compatible SQL character, graphic, and date data
types for XML schema decomposition.

Table 36. Compatible XML schema and SQL data types

| XML schema type | CHAR | VARCHAR | DATE | TIME | TIMESTAMP |
|---|---|---|---|---|---|
| string, normalizedString, token | 1a* | 1a | 7 | 8 | 9 |
| base64Binary, | 2* | 2 | No | No | No |
| hexBinary | 2* | 2 | No | No | No |
| byte, unsigned byte | 3* | 3 | No | No | No |
| integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger | 3* | 3 | No | No | No |
| int, unsignedInt | 3* | 3 | No | No | No |
| long, unsignedLong | 3* | 3 | No | No | No |
| short, unsignedShort | 3* | 3 | No | No | No |
| decimal | 3* | 3 | No | No | No |
| float | 3* | 3 | No | No | No |
| double | 3* | 3 | No | No | No |
| boolean | 3* | 3 | No | No | No |
| time | 4* | 4 | No | 10 | No |
| dateTime | 4* | 4 | 11 | 12 | 13 |
| duration, gMonth, gYear, gDay, gMonthDay, gYearMonth | 4* | 4 | No | No | No |
| date | 4* | 4 | 14 | No | No |
| Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY | 1a* | 1a | No | No | No |
| ENTITIES, NMTOKENS, IDREFS, list types | 1b* | 1b | No | No | No |
| anyURI | 6* | 6 | No | No | No |
| language | 1a* | 1a | No | No | No |
| anySimpleType, union types | 5* | 5 | No | No | No |

## Notes

**\*** 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.

**No** Data types are not compatible for annotated XML schema decomposition.

**1a** Compatible if the length of the XML input string, in characters, is less than or equal to the length of the target column in 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).

**2** Compatible if the length of the XML input string, in characters, is less than or equal to the length of the target column in 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 input string is normalized according to the whitespace facet of the XML schema type. The encoded (original) sting is inserted.

**3** Compatible is 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.

**4** Compatible if the number of characters is the string representation of the XML input value,

| | computed after processing according to the db2-xdb:normalization setting, is less than or equal to the length of the target column in characters. Also compatible if db2-xdb:truncate is set to "true" or "1" for this column mapping.
|---|---|
| 5 | Compatible if the length of the XML input string, in characters, is less than or equal to the length of the target column in 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. |
| 6 | Compatible if the string length of URI, in characters, is less than or equal to the length of the target column in 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. Note that the URI itself, not the resource the URI points to, is inserted. |
| 7 | Compatible if the string is of a valid date format: *yyyy-mm-dd*, *mm/dd/yyyy*, or *dd.mm.yyyy*. |
| 8 | Compatible if the string is of a valid time format: *hh.mm.ss*, *hh:mm AM or PM*, or *hh:mm:ss*. |
| 9 | Compatible if the string is of a valid timestamp format: *yyyy-mm-dd-hh.mm.ss.nnnnnn*. |
| 10 | For XML values that contain subseconds, compatible only if the decomposition annotation specifies db2-xdb:truncate as "true" or "1". For XML values with timezone indicators and db2-xdb:truncate is set to "true" or "1" , timezone indicators are inserted without the timezone. |
| 11 | Compatible if the year is composed of four digits and is not preceded by the '-' sign. |
| 12 | Compatible if the XML value does not have a timezone indicator. If the XML value has a timezone indicator, then the values are compatible if db2-xdb:truncate is set to "true" or "1". |
| 13 | Compatible if the year is composed of four digits and is not preceded by the '-' sign. For XML values with timezone indicators, compatible if db2-xdb:truncate is set to "true" or "1". If subseconds are specified with more than six digits, compatible if db2-xdb:truncate is set to "true" or "1". |
| 14 | Compatible if the year is composed of four digits and is not preceded by the '-' sign. For XML values with timezone indicators, compatible if db2-xdb:truncate is set to "true" or "1". (Date values are inserted without the timezone in this case.) |

*Table 37. Compatible XML schema and SQL data types*

| XML schema type | SQL type | | | |
|---|---|---|---|---|
| | GRAPHIC | VARGRAPHIC | CLOB | DBCLOB |
| string, normalizedString, token | 1a* | 1a | 1a | 1a* |
| base64Binary, | No | No | 3 | No |
| hexBinary | No | No | 3 | No |
| byte, unsigned byte | No | No | 4 | No |
| integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger | No | No | 4 | No |
| int, unsignedInt | No | No | 4 | No |
| long, unsignedLong | No | No | 4 | No |
| short, unsignedShort | No | No | 4 | No |
| decimal | No | No | 4 | No |
| float | No | No | 4 | No |
| double | No | No | 4 | No |
| boolean | No | No | 4 | No |
| time | No | No | 5 | No |

Table 37. Compatible XML schema and SQL data types  (continued)

| XML schema type | GRAPHIC | VARGRAPHIC | CLOB | DBCLOB |
|---|---|---|---|---|
| | | | SQL type | |
| dateTime | No | No | 5 | No |
| duration, gMonth, gYear, gDay, gMonthDay, gYearMonth | No | No | 5 | No |
| date | No | No | 5 | No |
| Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY | 1a* | 1a | 1a | 1a* |
| ENTITIES, NMTOKENS, IDREFS, list types | 1b* | 1b | 1b | 1b* |
| anyURI | No | No | 6 | No |
| language | No | No | 1a | No |
| anySimpleType, union types | 2a* | 2a | 2a | 2a* |

## Notes

**\***   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.

**No**   Data types are not compatible for annotated XML schema decomposition.

**1a**   Compatible if the length of the XML input string, in double-byte characters, is less than or equal to the length of the target column in 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 characters, is less than or equal to the length of the target column in 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 in either case is the character content of the element or attribute.

**3**   Compatible if the length of the XML input string, in characters, is less than or equal to the length of the target column in 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 input string is normalized according to the whitespace facet of the XML schema type. The encoded (original) sting is inserted.

**4**   Compatible if the length of the XML string, computed after processing according to the db2-xdb:normalization setting, is less than or equal to the length of the target column in characters. Also compatible if db2-xdb:truncate is set to "true" or "1" for this column mapping.

**5**   Compatible if the number of characters in the string representation of the XML input value, computed after processing according to the db2-xdb:normalization setting, is less than or equal to the length of the target column in characters. Also compatible if db2-xdb:truncate is set to "true" or "1" for this column mapping.

**6**   Compatible if the string length of URI, in characters, is less than or equal to the length of the target column in 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. Note that the URI itself, not the resource the URI points to, is inserted.

The following table lists XML schema data types and compatible SQL binary data types for XML schema decomposition.

*Table 38. Compatible XML schema and SQL data types*

| XML schema type | SQL type | | |
|---|---|---|---|
| | **BINARY** | **VARBINARY** | **BLOB** |
| string, normalizedString, token | 1a* | 1a | 1a |
| base64Binary, | 1a | 1a | 1a |
| hexBinary | 2* | 2 | 2 |
| byte, unsigned byte | No | No | No |
| integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger | No | No | No |
| int, unsignedInt | No | No | No |
| long, unsignedLong | No | No | No |
| short, unsignedShort | No | No | No |
| decimal | No | No | No |
| float | No | No | No |
| double | No | No | No |
| boolean | No | No | No |
| time | No | No | No |
| dateTime | No | No | No |
| duration, gMonth, gYear, gDay, gMonthDay, gYearMonth | No | No | No |
| date | No | No | No |
| Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY | 1a* | 1a | 1a |
| ENTITIES, NMTOKENS, IDREFS, list types | 1b* | 1b | 1b |
| anyURI | 1a | 1a | 1a |
| language | 1a* | 1a | 1a |
| anySimpleType, union types | 3* | 3 | 3 |

**Notes**

\*    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.

**No**   Data types are not compatible for annotated XML schema decomposition.

**1a**   Compatible if the length of the XML input string, in characters, is less than or equal to the length of the target column in 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 in either case is the character content of the element or attribute.

**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. String length is computed after normalization, where the input string is normalized according to the whitespace facet of the XML schema type. The encoded (original) string is inserted.

| 3 | 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. |

The following tables list XML schema data types and compatible SQL numeric data types for XML schema decomposition.

*Table 39. Compatible XML schema and SQL data types*

| XML schema type | SQL type | | | | | |
|---|---|---|---|---|---|---|
| | SMALLINT | INTEGER | BIGINT | REAL | DOUBLE | DECFLOAT |
| string, normalizedString, token | 1 | 1 | 1 | 1 | 1 | 1 |
| base64Binary, | No | No | No | No | No | No |
| hexBinary | No | No | No | No | No | No |
| byte, unsigned byte | 2 | 2 | 2 | 2 | 2 | 2 |
| integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger | 3 | 3 | 3 | 4 | 4 | 3 |
| int, unsignedInt | 3 | 2 | 2 | 4 | 2 | 2 |
| long, unsignedLong | 3 | 3 | 2 | 4 | 2 | 2 |
| short, unsignedShort | 2 | 2 | 2 | 2 | 2 | 2 |
| decimal | 4 | 4 | 4 | 4 | 4 | 4 |
| float | 5 | 5 | 5 | 6 | 6 | 6 |
| double | 5 | 5 | 5 | 5 | 6 | 6 |
| boolean | 7 | 7 | 7 | 7 | 7 | 7 |
| time | No | No | No | No | No | No |
| dateTime | No | No | No | No | No | No |
| duration, gMonth, gYear, gDay, gMonthDay, gYearMonth | No | No | No | No | No | No |
| date | No | No | No | No | No | No |
| Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY | No | No | No | No | No | No |
| ENTITIES, NMTOKENS, IDREFS, list types | No | No | No | No | No | No |
| anyURI | No | No | No | No | No | No |
| language | No | No | No | No | No | No |
| anySimpleType, union types | No | No | No | No | No | No |

**Notes**

| No | Data types are not compatible for annotated XML schema decomposition. |

| 1 | Compatible if the following conditions are true: |

- The string complies with the XML schema lexical representation rules for the target type.
- The string can be converted to the numeric value without truncation or loss of significant digits.

| 2 | Compatible, and where -0 is in the value space of the XML type, -0 is stored 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 in the database. |

| 4 | Compatible if value can be converted to the numeric value without truncation of loss of significant digits. Where -0 is in the value space of the XML type, -0 is stored in the database. |

| 5 | Compatible if the value can be converted to the numeric value without truncation or loss of significant digits, and the value is not "INF", "-INF", or "NaN". -0 is stored as 0 in the database. |

**6**      Compatible is the value is not "INF", "-INF", or "NaN". -0 is stored as 0 in the database.

**7**      Compatible, and the value inserted is '0' (for false) or '1' (for true).

*Table 40. Compatible XML schema and SQL data types*

| | SQL type | |
|---|---|---|
| XML schema type | DECIMAL | NUMERIC |
| string, normalizedString, token | 1 | 1 |
| base64Binary, | No | No |
| hexBinary | No | No |
| byte, unsigned byte | 2 | 2 |
| integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger | 4 | 4 |
| int, unsignedInt | 4 | 4 |
| long, unsignedLong | 4 | 4 |
| short, unsignedShort | 2 | 2 |
| decimal | 4 | 4 |
| float | 5 | 5 |
| double | 5 | 5 |
| boolean | 7 | 7 |
| time | No | No |
| dateTime | No | No |
| duration, gMonth, gYear, gDay, gMonthDay, gYearMonth | No | No |
| date | No | No |
| Name, NCName, NOTATION, ID, IDREF, QName, NMTOKEN, ENTITY | No | No |
| ENTITIES, NMTOKENS, IDREFS, list types | No | No |
| anyURI | No | No |
| language | No | No |
| anySimpleType, union types | No | No |

### Notes

**No**      Data types are not compatible for annotated XML schema decomposition.

**1**      Compatible if the following conditions are true:
  - The string complies with the XML schema lexical representation rules for the target type.
  - The string can be converted to the numeric value without truncation or loss of significant digits.

**2**      Compatible, and where -0 is in the value space of the XML type, -0 is stored 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 in the database.

**4**      Compatible if value can be converted to the numeric value without truncation of loss of significant digits. Where -0 is in the value space of the XML type, -0 is stored in the database.

**5**      Compatible if the value can be converted to the numeric value without truncation or loss of significant digits, and the value is not "INF", "-INF", or "NaN". -0 is stored as 0 in the database.

**6**      Compatible is the value is not "INF", "-INF", or "NaN". -0 is stored as 0 in the database.

**7**      Compatible, and the value inserted is '0' (for false) or '1' (for true).

## Limits and restrictions for annotated XML schema decomposition
Certain limits and restrictions apply to annotated XML schema decomposition.

# Limits

*Table 41. Limits for annotated XML schema decomposition*

| Condition | Limit value |
|---|---|
| Maximum size of document to be decomposed | 2 GB |
| Maximum number of tables referred to in a single annotated XML schema | 100 |
| Maximum length of a string value of db2-xdb:expression | 1024 bytes |
| Maximum length of a string value of db2-xdb:condition | 1024 bytes |
| Maximum number of steps in db2-xdb:locationPath | 128 |
| Maximum number of unique annotation in an XML schema | 64 KB |
| 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 |

# 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.

  However, if these elements or attributes are children of elements that are decomposed with db2-xdb:contentHandling set to `serializeSubtree` or `stringValue`, the contents of the wildcard elements or attributes are decomposed as part of the serialized subtree or string value. These wildcard elements or attributes must satisfy the namespace constraints that are specified in the corresponding <xs:any> or <xs:anyAttribute> declarations.

- Substitution groups

  An error is generated if a member of a substitution group appears in an XML document where the group head appears in the XML schema.

- Runtime substitution using xsi:type

  An element is decomposed according to the mappings in the schema type that is associated with the element name in the XML schema. Use of `xsi:type` to specify a different type for an element in the document results in a decomposition error.

- Recursive elements (elements that refer to themselves)

  Annotated XML schemas that contain recursive element declarations cannot be enabled for decomposition.

- Updates or deletes of existing rows in target tables

  Currently, decomposition can only insert new rows into target tables.

- Referential constraints between tables that are updated by decomposition

  To circumvent this restriction, use different annotated schemas for multiple decomposition operations on the same instance document, to control the order in which the parent or child tables are updated.

- Attributes of simple type derived from NOTATION: decomposition inserts only the notation name.

- Attributes of type ENTITY: decomposition inserts only the entity name.

- 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.

## 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. This topic shows the XML schema for the annotated schema as defined by XML decomposition.

```
<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: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:element name='rowSetOperationOrder'>
    <xs:complexType>
      <xs:choice minOccurs='1'  maxOccurs='1'>
        <xs:element name='order' type='orderType' minOccurs='1'
         maxOccurs='unbounded'/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:complexType name='orderType'>
```

```
    <xs:sequence>
      <xs:element name='rowSet' type='xsd:string' minOccurs='2'
      maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

# XML data model

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

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

## Sequences and items

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

A sequence can contain nodes, atomic values, or any mixture of nodes and atomic values. For example, each of the following values can each be represented as a single sequence:
- 36
-
- (2, 3, 4)
- (36, , "cat")
- ()
- An XML document

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

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

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

## Atomic values

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

These data types include strings, integers, decimals, dates, and other atomic types. These types are described as "atomic" because they cannot be subdivided. Some atomic types have literal values. For example, the following literals are atomic values:
- "this is a string"
- 45
- 1.44

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

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

# Nodes

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

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

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

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

## Node identity

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

## Document order

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

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

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

   The relative order of attribute nodes is arbitrary, but this order does not change during the processing of an XPath expression.
- Element nodes, text nodes, processing instruction nodes, and comment nodes can be children of an element node or a document node.
- The relative order of siblings is determined by their order in the node hierarchy.
- Children and descendants of a node occur before siblings that follow the node.

## Node properties

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

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

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

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

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

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

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

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

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

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

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

## Document nodes

A document node encapsulates an XML document.

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

A document node has the following node properties:

- children
- string value
- typed value

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

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

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

The document node has the following property values:

*Table 42. Properties of the document node*

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

## Element nodes

An element node encapsulates an XML element.

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

An element node has the following node properties:
* node name
* parent
* type name (The type name of an element node in DB2 is always xs:untyped.)
* children
* attributes
* string value
* typed value
* in-scope namespaces

For an element node, the string value is the concatenation of the string values of all of its text node descendents in document order. If the element is empty, the string value is the empty string "". The typed value of an element is one of the following values:
* If the element can be null, the typed value is ().
* If the element is empty, the typed value is the empty sequence ().
* Otherwise, the typed value is its string value as type xs:untypedAtomic.

For example, suppose that a document has the following textual representation:
```
<product xmlns="http://posample.org" pid="100-101-01">
  <description>
    <name>Snow Shovel, Deluxe 24"</name>
    <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic
           curved handle with D-Grip</details>
    <price>19.99</price>
    <weight>2 kg</weight>
  </description>
</product>
```

The product element node has the following property values:

*Table 43. Properties of the product node*

| Node property | Value | Value type |
|---|---|---|
| node name | product | |
| parent | document node | |

*Table 43. Properties of the product node  (continued)*

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

The name element node has the following property values:

*Table 44. Properties of the name node*

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

## Attribute nodes

An *attribute node* represents an XML attribute.

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

An attribute node has the following node properties:

- node name
- parent
- type name (The type name of an attribute node in DB2 is always xs:untypedAtomic.)
- string value
- typed value

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

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

The pid attribute has the following property values:

*Table 45. Properties of the pid attribute node*

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

## Text nodes

A text node encapsulates XML character content.

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

Text nodes have the following node properties:
* content
* parent

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

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

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

*Table 46. Properties of the name text node*

| Node property | Value |
|---|---|
| content | Snow Shovel, Deluxe 24" |
| parent | name |

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

## Processing instruction nodes

A processing instruction node encapsulates an XML processing instruction.

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

A processing instruction node has the following node properties:
* target

- content
- parent

For example, consider the following processing instruction:

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

This processing instruction has the following property values:

*Table 47. Properties of the processing instruction node*

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

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

## Comment nodes

A comment node encapsulates XML comments.

A comment node can have zero or one parent.

A comment node has the following node properties:
- content
- parent

For example, consider the following content:

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

This comment has the following property values:

*Table 48. Properties of the comment node*

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

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

## Data model generation

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

An input XML document is transformed into an instance of the XML data model through a process called *XML parsing*. Alternatively, you can generate an instance of the XML data model by using SQL XML constructors, such as XMLELEMENT and XMLATTRIBUTES. These built-in functions enable you to

generate XML data from relational data. Likewise, the result of an XPath expression can be transformed into an XML representation through a process called *XML serialization*.

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

  **Restriction:** If the XML document is validated against a schema, DB2 does not keep the type annotation for each node. The data is stored as untyped.

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

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

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

  If an input document has no schema, the document is not validated. DB2 assigns nodes and atomic values as untyped (xs:untyped or xs:untypedAtomic).

- During *serialization*, the sequence of nodes and atomic values (the instance of the XPath data model) is converted into its string representation. The result of serialization does not always represent a well-formed document. In fact, serialization can result in a single atomic value (for example, 17) or a sequence of elements that do not have a common parent.

## XML values in SQL

In terms of the XQuery/XPath 2.0 data model, DB2 for i SQL defines an XML value as a sequence that contains a single document node, with a sub-tree containing the document's content. Representing an XML value as a document node guarantees that the value can be serialized to a character representation that exactly represents the XML value. This definition is referred to as XML(CONTENT) in the 2008 Database Language SQL Part 14 - XML-Related Specifications (ISO 9075-14).

In order to ensure that the XML value is of type XML(CONTENT), a document node is constructed when an XML value is created or copied within SQL.

The following process is observed during the construction of the document node:

1. If the content sequence contains a document node, the document node is replaced by its children.
2. Any atomic values in the content sequence are converted to strings and stored in text nodes, which become children of the constructed document.
3. Adjacent text nodes in the content sequence are merged into a single text node.
4. If the content sequence contains an attribute node, an error is raised.

No validation is performed on the document node. The XML 1.0 rules that govern the structure of an XML document (for example, the document node must have exactly one child that is an element node) are not enforced during the construction of the XML value.

For example:
```
XMLCONCAT(XMLTEXT('text node one '), XMLTEXT('text node two'))
```

The result XML value will be represented as a sequence of a single document node , with a single child text node 'text node one text node two'. The representation is consistent with a serialized XML document.

Other implementations of the specification (including DB2 for z/OS® and DB2 for LUW) may define the SQL XML type as an XQuery/XPath 2.0 sequence that can contain any number of items of any type of

node or atomic value, This definition does not guarantee that the value can be serialized to a character representation that exactly represents the sequence. This type is referred to as XML(SEQUENCE) in the specification and is a superset of XML(CONTENT).

SQL implementations that create a value of type XML(SEQUENCE) will represent the previous XMLCONCAT expression's result as a sequence of two adjacent text nodes (with no parent document node): ('text node one', 'text node two') .

In most cases, this difference in representation is not significant. When a value of XML(SEQUENCE) is serialized, it must first be converted to XML(CONTENT), using the same process that occurs when an XML(CONTENT) value is created. Therefore, serializing an XML value produces the same result regardless of the XML type used by the implementation.

In addition, well formed XML documents that are obtained from a column in a table, host variable, parameter marker, or by using the XMLPARSE built in function contain a root document node. This causes the XML value to already be the more specific type of XML(CONTENT), even in environments that implement the more general type of XML(SEQUENCE).

Different results can occur in a small number of cases when an XML value is constructed in SQL and evaluated by an XPath expression.

For example:
```
select XMLSERIALIZE(OUTPUT_COL AS VARCHAR(100)) from
 XMLTABLE('$d_or_e/root/child'
          passing XMLELEMENT(NAME "root",
                             XMLELEMENT(NAME "child",
                                        XMLTEXT('hello world'))) as "d_or_e"
          ) X(OUTPUT_COL);
```

DB2 for i will assign $d_or_e to a document node that represents the XML value, The step expression is evaluated and the expected output <child>hello world</child> is returned. "root" is a child element of the document node.

However, DB2 for LUW and DB2 for z/OS will assign $d_or_e to an element "root". Since element "root" has no child called "root", this query will not return any rows.

The correct way to provide a fully platform independent solution is to write such a query so that a document node is explicitly constructed, which forces the representation to be equivalent on all platforms that comply with the specifications.
```
select XMLSERIALIZE(OUTPUT_COL AS VARCHAR(100)) from
 XMLTABLE('$d_or_e/root/child' passing
                    XMLDOCUMENT(
                            XMLELEMENT(NAME "root",
                                    XMLELEMENT(NAME "child",
                                            XMLTEXT('hello world')))
                            ) as "d_or_e"
 ) X(OUTPUT_COL);
```

Explicit construction of a document node is not necessary for XML values that are guaranteed to be XML(CONTENT), such as when:
- the value is obtained from the application via host variable or parameter marker,
- the value is from a column in a DB2 table, or
- the XML value is explicitly or implicitly created using the XMLPARSE built in function.

# Overview of XPath

XPath is an expression language that was designed by the World Wide Web Consortium (W3C) to allow processing of XML data that conforms to the XQuery 1.0 and XPath 2.0 data model. XQuery is a functional programming language that was designed by the World Wide Web Consortium (W3C) to meet specific requirements for querying and modifying XML data. DB2 XPath supports a subset of the language constructs in the XPath 2.0 recommendation.

The XPath language provides several kinds of expressions that can be constructed from keywords, symbols, and operands. In most cases, the operands of various expressions, operators, and functions must conform to the expected types. DB2 ignores type errors in certain situations.

DB2 XPath can be used as an argument to the XMLTABLE SQL built-in table function, which is used to convert an XML value into a relational result set.

## XPath expressions

The basic building block of XPath is the expression. DB2 XPath provides several kinds of expressions for working with XML data:

- Primary expressions, which include the basic primitives of the language, such as literals, variable references, and function calls
- Path expressions for locating nodes within a document tree
- Arithmetic expressions for addition, subtraction, multiplication, division, and modulus
- Comparison expressions for comparing two values
- Logical expressions for using boolean logic

XPath expressions can be composed with full generality, which means that where an expression is expected, any kind of expression can be used. In general, the operands of an expression are other expressions. In the following example, the operands of a logical expression are the comparison expressions 1 = 1 and 2 = 2:

```
1 = 1 and 2 = 2
```

## XPath processing

An XPath expression consists of an optional *prolog* that establishes the processing environment and an *expression* that generates a result. XPath processing occurs in two phases: the static analysis phase and the dynamic evaluation phase.

During the static analysis phase, the expression is parsed and augmented based on information that is defined in the prolog. The static context is used to resolve type names, function names, and variable names that are used by the expression. The *static context* includes all information that is available prior to evaluating an expression. The static phase occurs when the expression is first evaluated. If a required name is not found in the static context, an error is raised.

The dynamic evaluation phase occurs if no errors are detected during the static analysis phase. During the dynamic evaluation phase, the value of the expression is computed. A dynamic type is associated with each value as the value is computed. If an operand of an expression has a dynamic type that does not match the expected type, a type error is raised. If the evaluation generates no errors, a result is returned. The *dynamic context* includes information that is available at the time the expression is evaluated.

The result of an XPath expression is, in general, a heterogeneous sequence of XML nodes and atomic values. More specifically, the result of an XPath expression is an instance of the XPath data model.

## The XPath 2.0 and XQuery 1.0 data model

The XPath 2.0 and XQuery 1.0 data model represents an XML document as a hierarchy (tree) of nodes that represent XML elements and attributes. Each value of the data model is a sequence that can contain zero, one, or more items. The items can be atomic values or nodes. Every XPath expression takes as its input an instance of the XPath 2.0 and XQuery 1.0 data model and returns an instance of the XPath 2.0 and XQuery 1.0 data model.

## DB2 XPath data types

DB2 XPath supports the following data types:
- xs:integer
- xs:decimal
- xs:double
- xs:string
- xs:boolean
- xs:untypedAtomic
- xs:date
- xs:dateTime
- xs:time
- xs:duration
- xs:yearMonthDuration
- xs:dayTimeDuration

DB2 checks data types during the dynamic evaluation phase and the static analysis phase. When an expression encounters an inappropriate type, a type error is raised. For example, an XPath expression that uses the plus operator (+) to add two strings together results in a type error because the plus operator is only used in arithmetic expression to add numeric, yearMonthDuration, and dayTimeDuration values. Implicit type conversions and type substitutions occur, when possible, to provide the type that is expected by an expression.

## The built-in function library

DB2 XPath provides a library of built-in functions for working with XML data. The library includes the following types of functions:
- String functions
- Numeric functions
- Date and time functions
- Functions that operate on boolean values
- Functions that operate on sequences

These built-in functions are in the namespace with URI `http://www.w3.org/2005/xpath-functions`, which by default is associated with the prefix `fn`. The default function namespace is set to `fn` by default, which means that you can call functions in this namespace without specifying a prefix.

Function calls can be used anywhere in an XPath expression where an expression is expected.

# Case sensitivity in DB2 XPath

XPath is a case-sensitive language.

Keywords in XPath use lowercase characters and are not reserved. Names in XPath expressions are allowed to be the same as language keywords.

## Whitespace in DB2 XPath

Whitespace is allowed in most XPath expressions to improve readability even if whitespace is not part of the syntax for the expression. Whitespace consists of space characters (U+0020), carriage returns (U+000D), line feeds (U+000A), and tabs (U+0009).

In general, whitespace is not significant in an XPath expression, except in the following situations where whitespace is preserved:

- The whitespace is in a string literal.
- The whitespace clarifies an expression by preventing two adjacent tokens from being mistakenly recognized as one.

The following examples include expressions that require whitespace for clarity:

- `one- two` results in a syntax error. The parser recognizes `one-` as a single QName (qualified name) and raises an error when no operator is found.
- `one -two` does not result in a syntax error. The parser recognizes `one` as a QName, the minus sign (-) as an operator, and then `two` as another QName.
- `one-two` does not result in a syntax error. However, the expression parses as a single QName because a hypen (-) is a valid character in a QName.
- The following expressions all result in syntax errors:
  - `5 div2`
  - `5div2`

  In these expressions, whitespace is required for the parser to recognize each token separately. Notice that `5div 2` does not result in a syntax error.

## Comments in DB2 XPath

Comments are allowed in an XPath expression, wherever nonessential whitespace is allowed. Comments do not affect expression processing.

A comment is a string that is delimited by the symbols (: and :). The following example is a comment in XPath:

```
(: This is a comment. It makes code easier to understand. :)
```

The following general rules apply to using comments in DB2 XPath:

- Comments can be used wherever nonessential whitepace is allowed. *Nonessential whitespace* is whitespace that is not part of the syntax of an XPath expression.
- Comments can nest within each other, but each nested comment must have open and close delimiters, (: and :).

The following examples illustrate legal comments and comments that result in errors:

- `(: is this a comment? ::)` is a legal comment.
- `(: is this a comment? ::) or an error? :)` results in an error because there is an unbalanced nesting of the symbols (: and :).
- `(: commenting out a (: comment :) may be confusing, but often helpful :)` is a legal comment because a balanced nesting of comments is allowed.
- `"this is just a string :)"` is a legal expression.
- `(: "this is just a string :)" :)` results in a syntax error. Likewise, `"this is another string (:"` is a legal expression, but `(: "this is another string (:" :)` results in a syntax error. Literal content can result in an unbalanced nesting of comments.

## Character set

The DB2 XPath model uses a UTF-8 encoding. DB2 will convert all character, graphic, and XML input parameters to UTF-8. It will cast output data from UTF-8 to the CCSID specified for the result column.

## Default collation

DB2 XPath determines the default collation for the XPath expression from the collating sequence used for the SQL statement containing the XMLTABLE. DB2 XPath supports binary (*HEX) collating sequences and Unicode collating sequences using ICU (International Components for Unicode).

## XML namespaces and qualified names in DB2 XPath

DB2 XPath uses XML namespaces to prevent naming collisions. An *XML namespace* is a collection of names that is identified by a namespace URI. Namespaces provide a way of qualifying names that are used for elements, attributes, data types, and functions in XPath.

Names in XPath are called QNames (qualified names) and conform to the syntax that is defined in the W3C Recommendation *Namespaces in XML*. A *QName* consists of an optional namespace prefix and a local name. The namespace prefix, if present, is bound to a URI and provides a shortened form of the URI. During query processing, DB2 XPath expands the QName by resolving the URI that is bound to the namespace prefix. The expanded QName includes the namespace URI and a local name. Two QNames are equal if they have the same namespace URI and local name. This means that two QNames can match even if they have different prefixes provided that the prefixes are bound to the same namespace URI.

Using QNames in XPath allows expressions to refer to element types or attribute names that have the same local name, but might be associated with different DTDs or XML Schemas. In the following XML data, `pfx1` is a prefix that is bound to some URI. `pfx2` is a prefix that is bound to a different URI. `c` is the local name for all three elements:

```
<a xmlns:pfx1="uri1" xmlns:pfx2="uri2">
<b>
 <pfx1:c>C</pfx1:c>
 <pfx2:c>B</pfx2:c>
 <c>A</c>
</b>
</a>
```

The elements in this example share the same local name, `c`, but naming conflicts do not occur because the elements exist in different namespaces. During expression processing, the name `pfx1:c` is expanded into a name that includes the URI bound to `pfx1` (`uri1`) and the local name, `c`. Likewise, the name `pfx2:c` is expanded into a name that includes the URI bound to `pfx2` (`uri2`) and the local name, `c`. The element `c`, which has an empty prefix, is bound to the default element namespace because no prefix is specified. An error is raised if a name uses a prefix that is not bound to a URI.

The namespace prefix must be an NCName (non-colonized name). An XML NCName is similar to an XML Name except that NCName cannot include a colon.

Some namespaces are predeclared; others can be added through declarations in the XPath expression prolog. DB2 XPath includes the following predeclared namespace prefixes:

| Prefix | URI | Description |
| --- | --- | --- |
| xs | http://www.w3.org/2001/XMLSchema | XML Schema namespace |
| xsi | http://www.w3.org/2001/XMLSchema-instance | XML Schema instance namespace |
| fn | http://www.w3.org/2005/xpath-functions | Default function namespace |
| xdt | http://www.w3.org/2005/xpath-datatypes | XPath type namespace |
| db2-fn | http://www.ibm.com/xmlns/prod/db2/functions | DB2 function namespace |

In addition to the predeclared namespaces, namespaces can be provided in the following ways:

- The following namespace information is available in the static context:
  - *In-scope namespaces* are a set of prefix and URI pairs. The in-scope namespaces are used for resolving prefixes that are used in QNames in an XPath expression. In-scope namespaces come from the following sources:
    - Namespace declarations in an XPath expression
    - The XMLNAMESPACES DB2 built-in function in the XMLELEMENT, XMLFOREST, or XMLTABLE DB2 built-in function
  - *Default element or type namespace* is the namespace that is used for any unprefixed QName that appears where an element or type name is expected. The initial default element or type namespace is the default namespace that is provided by a `declare default element namespace` clause in the prolog of an XPath expression.
  - *Default function namespace* is the namespace that is associated with built-in functions: http://www.w3.org/2003/11/xpath-functions. There are no user-defined functions in XPath.

# XPath type system

DB2 XPath is a strongly typed language in which the operands of various expressions, operators, and functions conform to expected types.

The type system for DB2 XPath includes a subset of the built-in types of XML schema and the predefined types of XPath.

The built-in types of XML Schema are in the namespace `http://www.w3.org/2001/XMLSchema`, which has the predeclared namespace prefix `xs`. Some examples of built-in schema types include `xs:integer` and `xs:string`.

## Overview of the type system

The type system for DB2 XPath includes simple atomic types and complex types. A *simple atomic type* is a primitive or derived atomic type that does not contain elements or attributes. A *complex type* can contain mixed content or element-only content.

## Constructor functions for built-in data types

Every built-in atomic type that is defined in the XML Schema Definition language has an associated constructor function.

### Syntax

►►—*prefix:type*(*value*)—————————————————————————————►◄

*prefix*
    The prefix that is bound to the namespace for the data type. This is not the prefix that is bound to the default function namespace.

*type*
    The unqualified name of the target data type.

*value*
    The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

    If *value* is illegal for the target data type, the constructor function returns an error.

## Returned value

If *value* is not the empty sequence, the returned value is an instance of *prefix:type*.

If *value* is the empty sequence, a constructor function returns the empty sequence.

### Example:

The constructor function `xs:integer(100)` or the constructor function `xs:integer("100")` returns the `xs:integer` value 100. A constructor function whose argument is a node with the typed value 100 also returns the typed value 100.

## Generic data types

Generic data types support data that is not strongly typed.

**xs:anyType:**

The data type `xs:anyType` is the base type for all data types that are defined in the XML Schema Definition language.

**xs:anySimpleType:**

The data type `xs:anySimpleType` is the base type for all primitive types that are defined in the XML Schema Definition language.

`xs:anySimpleType` is used to define a required type (for example, in a function signature) to indicate that any simple type is acceptable. The base type of `xs:anySimpleType` is `xs:anyType`.

Casting is not supported to or from `xs:anySimpleType`.

**Lexical form**

`xs:anySimpleType` can have any lexical form.

**xs:anyAtomicType:**

The data type `xs:anyAtomicType` is the base type for all primitive atomic types that are defined in the XML Schema Definition language.

**Lexical form**

The data type `xs:anyAtomicType` can be used to define a required type (for example, in a function signature) to indicate that any of the primitive atomic types or `xs:untypedAtomic` is acceptable. The base type of `xs:anyAtomicType` is `xs:anySimpleType`.

`xs:anyAtomicType` can have any lexical form.

## Data types for untyped data

The xs:untyped and xs:untypedAtomic data types support untyped data.

**xs:untyped:**

The data type `xs:untyped` serves as a special type annotation to indicate types that have not been validated by an XML schema. The data type `xs:untyped` can be used (for example, in a function signature) to define a required type to indicate that only an untyped value is acceptable. The base type of `xs:untyped` is `xs:anyType`.

If an element node is annotated as `xs:untyped`, all of its descendant element nodes are also annotated as `xs:untyped`. DB2 for i does not retain the type annotations from schema validations. All elements in an XML document have a type of `xs:untyped`.

**xs:untypedAtomic:**

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

An attribute that has an unknown type is represented in the data model by an attribute node with the type `xs:untypedAtomic`. The data type `xs:untypedAtomic` can be used (for example, in a function signature) to define a required type to indicate that only an untyped atomic value is acceptable. The base type of `xs:untypedAtomic` is `xs:anyAtomicType`. There is no constructor for this type. DB2 for i does not preserve type annotations in an XML document, even if the document has been validated with an XML schema. All attributes will have a type of `xs:untypedAtomic` in the data model.

**Lexical form**

`xs:untypedAtomic` can have any lexical form.

## xs:string

The data type `xs:string` represents character strings in XML. Because `xs:string` is a simple type, it cannot contain any children.

### Lexical form

The lexical form of `xs:string` is a sequence of characters that can include any character that is in the range of legal characters for XML.

### Constructor

Use the following syntax to construct an instance of `xs:string`:

►►──xs:string(*value*)──────────────────────────────────────────────►◄

*value*
> The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.
>
> If *value* is illegal for the target data type, the constructor function returns an error.

## Numeric data types

The xs:decimal, xs:double, and xs:integer data types support numeric data.

**xs:decimal:**

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

**Lexical form**

The lexical form of `xs:decimal` is a finite-length sequence of decimal digits (0 to 9) that are separated by a period as a decimal indicator. An optional leading sign is allowed. If the sign is omitted, a positive sign (+) is assumed. Leading and trailing zeroes are optional. If the fractional part is zero, the period and any following zeroes can be omitted. The following numbers are all valid examples of a decimal:

• -1.23

- 12678967.543233
- +100000.00
- 210.

**Constructor**

Use the following syntax to construct an instance of `xs:decimal`:

►►—xs:decimal(*value*)————————————————————————————————◄◄

***value***

    The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

    If *value* is illegal for the target data type, the constructor function returns an error.

**xs:double:**

The data type `xs:double` is supported in DB2 XPath by the IEEE 64-bit decimal floating point.

**Lexical form**

The lexical form of `xs:double` is a mantissa followed, optionally, by the character E or e, followed by an exponent. The exponent must be an integer. The mantissa must be a decimal number. The representations for exponent and mantissa must follow the lexical rules for `xs:integer` and `xs:decimal`. If the E or e and the exponent that follows are omitted, an exponent value of 0 is assumed.

The special values positive infinity, negative infinity, and not-a-number have the lexical representations INF, -INF and NaN, respectively. Lexical representations for zero can take a positive or negative sign. The following literals are all valid examples of a double:
- -1E4
- 1267.43233E12
- 12.78e-2
- 12
- -0
- 0
- INF

**Constructor**

Use the following syntax to construct an instance of `xs:double`:

►►—xs:double(*value*)————————————————————————————————◄◄

***value***

    The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

    If *value* is illegal for the target data type, the constructor function returns an error.

**xs:integer:**

The data type `xs:integer` represents a decimal number that does not include a trailing decimal point. The base type of `xs:integer` is `xs:decimal`.

**Lexical form**

The lexical form of `xs:integer` is a finite-length sequence of decimal digits (0 to 9) with an optional leading sign. If the sign is omitted, a positive sign (+) is assumed. The following numbers are all valid examples of integers:

- -1
- 0
- 12678967543233
- +100000

**Constructor**

Use the following syntax to construct an instance of `xs:integer`:

►►──xs:integer(*value*)───────────────────────────────────────────────────────────►◄

*value*
    The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.

    If *value* is illegal for the target data type, the constructor function returns an error.

**Range limits for numeric types:**

DB2 XPath has range limits for numeric data types.

The following table lists the range limit and SQL equivalent for each XPath numeric data type.

*Table 49. Range limits for numeric types*

| XML type | DB2 XML range | SQL type mapping |
|---|---|---|
| xs:double | 34 digits of precision and an exponent range of $10^{**}-6143$ to $10^{**}+6144$ | DECFLOAT |
| xs:decimal | Up to 34 digits of precision, and a range of $1-10^{**}34$ to $10^{**}34 -1$ | DECIMAL<br>Note that truncation might happen for precision above 34 digits |
| xs:integer | -9223372036854775808 to 9223372036854775807 | BIGINT |

## xs:boolean

The data type `xs:boolean` supports the mathematical concept of binary-valued logic: true or false.

**Lexical form**

The lexical form of the data type `xs:boolean` can be one of the literal values true, false, 1, or 0.

**Constructor**

Use the following syntax to construct an instance of `xs:boolean`:

►►──xs:boolean(*value*)───────────────────────────────────────────────────────────►◄

*value*

> The value that is to be constructed. If this value is an empty sequence, the empty sequence is returned.
>
> If *value* is illegal for the target data type, the constructor function returns an error.

## Date and time data types

The xs:date, xs:time, and xs:dateTime data types support date and time data.

**xs:date:**

The date type xs:date represents an interval of exactly one day that begins on the first moment of a given day.

**Lexical form**

The lexical form of xs:date is a finite-length sequence of characters of the following form: *yyyy-mm-ddzzzzzz*. The following abbreviations describe this form:

*yyyy*
> A four-digit numeral that represents the year.
>
> The value cannot begin with a negative (-) sign or a plus (+) sign.
>
> 0001 is the lexical representation of the year 1 of the Common Era (also known as 1 AD).
>
> The value cannot be 0000.

**-** Separators between parts of the date.

*mm* A two-digit numeral that represents the month.

*dd* A two-digit numeral that represents the day.

*zzzzzz*
> Optional. If present, represents the time zone.

**Timezone indicator**

The lexical form for the time zone indicator is a string that includes one of the following forms:

- A positive (+) or negative (-) sign that is followed by `hh:mm`, where the following abbreviations are used:

  *hh* A two-digit numeral (with leading zeros as required) that represents the hours. The value must be between -14 and +14, inclusive.

  *mm* A two-digit numeral that represents the minutes. The value of the minutes property must be zero when the hours property is equal to 14.

  **+** Indicates that the specified time instant is in a time zone that is ahead of the UTC time by *hh* hours and *mm* minutes.

  **-** Indicates that the specified time instant is in a time zone that is behind UTC time by *hh* hours and *mm* minutes.

- The literal Z, which represents the time in UTC (Z represents Zulu time, which is equivalent to UTC). Specifying Z for the time zone is equivalent to specifying `+00:00` or `-00:00`.

**Example**

The following form indicates 10 October 2009, Eastern Standard Time in the United States:
`2009-10-10-05:00`

This date represents a UTC date of `2009-10-10T05:00:00Z`.

**xs:time:**

The data type xs:time represents an instant of time that recurs every day.

**Lexical form**

The lexical form of the data type xs:time is *hh:mm:ss.sssssssssssszzzzzz*.

The lexical form of the data type xs:time is *hh:mm:ss.sssssszzzzzz*.

The following abbreviations describe this form:

*hh*  A two-digit numeral (with leading zeros as required) that represents the hours.

**:**  A separator between parts of the time portion.

*mm*  A two-digit numeral that represents the minute.

*ss*  A two-digit numeral that represents the whole seconds.

*.sssssssssssssssss*
  Optional. If present, a 1-to-12 digit numeral that represents the fractional seconds.

*zzzzzz*
  Optional. If present, represents the time zone.

**Example**

The following form, which includes an optional time zone indicator, represents 1:20 p.m. Eastern Standard Time, which is five hours behind than Coordinated Universal Time (UTC):

`13:20:00-05:00`

**xs:dateTime:**

The data type xs:dateTime represents an instant in time.

The xs:dateTime data type has the following properties:
- year
- month
- day
- hour
- minute
- second
- time zone (optional)

The year, month, day, hour, and minute properties are expressed as integer values. The seconds property is expressed as a decimal value. The time zone property is expressed as a time zone indicator.

**Lexical form**

The lexical form of xs:dateTime is a finite-length sequence of characters of the following form: *yyyy-mm-ddThh:mm:ss.sssssssssssszzzzzz*. The following abbreviations describe this form:

*yyyy*
  A four-digit numeral that represents the year.

  The value cannot begin with a negative (-) sign or a plus (+) sign.

0001 is the lexical representation of the year 1 of the Common Era (also known as 1 AD).

The value cannot be 0000.

**-** Separators between parts of the date portion

*mm* A two-digit numeral that represents the month.

*dd* A two-digit numeral that represents the day.

**T** A separator to indicate that the time of day follows.

*hh* A two-digit numeral (with leading zeros as required) that represents the hours. The value must be between -14 and +14, inclusive.

**:** A separator between parts of the time portion.

*mm* A two-digit numeral that represents the minute.

*ss* A two-digit numeral that represents the whole seconds.

**.*sssssssssss***
　Optional. If present, a 1-to-12 digit numeral that represents the fractional seconds.

***zzzzzz***
　Optional. If present, represents the time zone. If a time zone is not specified the dateTime has no timezone; however, an implicit time zone of UTC (Coordinated Universal Time, also called Greenwich Mean Time) is used for comparison and arithmetic operations.

Each part of the datetime value that is expressed as a numeric value is constrained to the maximum value within the interval that is determined by the next-higher part of the datetime value. For example, the day value can never be 32 and cannot be 29 for month 02 and year 2002 (February 2002).

**Timezone indicator**

The lexical form for the time zone indicator is a string that includes one of the following forms:

- A positive (+) or negative (-) sign that is followed by `hh:mm`, where the following abbreviations are used:

  *hh* A two-digit numeral (with leading zeros as required) that represents the hours. The value must be between -14 and +14, inclusive.

  *mm* A two-digit numeral that represents the minutes. The value of the minutes property must be zero when the hours property is equal to 14.

  **+** Indicates that the specified time instant is in a time zone that is ahead of the UTC time by *hh* hours and *mm* minutes.

  **-** Indicates that the specified time instant is in a time zone that is behind UTC time by *hh* hours and *mm* minutes.

- The literal Z, which represents the time in UTC (Z represents Zulu time, which is equivalent to UTC). Specifying Z for the time zone is equivalent to specifying `+00:00` or `-00:00`.

**Example**

The following form indicates noon on 10 October 2009, Eastern Standard Time in the United States:
`2009-10-10T12:00:00-05:00`

This time is expressed in UTC as `2009-10-10T17:00:00Z`.

**xs:duration:**

The data type xs:duration represents a duration of time that is expressed by the Gregorian year, month, day, hour, minute, and second components. xs:duration is derived from data type xs:anyAtomicType.

The range that can be represented by this data type is from -P83333333333333Y3M11574074074DT1H46M39.999999999999S to P83333333333333Y3M11574074074DT1H46M39.999999999999S (or -999999999999999 months and -999999999999999.999999999999 seconds to 999999999999999 months and 999999999999999.999999999999 seconds).

The lexical form of xs:duration is the ISO 8601 extended format P*n*Y*n*M*n*DT*n*H*n*M*n*S. The following abbreviations describe the extended format:

**P**  The duration designator.

*n*Y  *n* is an unsigned integer that represents the number of years.

*n*M  *n* is an unsigned integer that represents the number of months.

*n*D  *n* is an unsigned integer that represents the number of days.

**T**  The date and time separator.

*n*H  *n* is an unsigned integer that represents the number of hours.

*n*M  *n* is an unsigned integer that represents the number of minutes.

*n*S  *n* is an unsigned decimal that represents the number of seconds. If a decimal point appears, it must be followed by one to twelve digits that represent fractional seconds.

For example, the following form indicates a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes:
```
P1Y2M3DT10H30M
```

The following form indicates a duration of negative 120 days:
```
-P120D
```

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:
- If the number of years, months, days, hours, minutes, or seconds in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator must be present.
- The seconds part can have a decimal fraction.
- The designator T must be absent if and only if all of the time items are absent.
- The designator P must always be present.

For example, the following forms are allowed:
```
P1347Y
P1347M
P1Y2MT2H
P0Y1347M
P0Y1347M0D
```

The form P1Y2MT is not allowed because no time items are present. The form P-1347M is not allowed, but the form -P1347M is allowed.

DB2 stores xs:duration values in a normalized form. In the normalized form, the seconds and minutes components are less than 60, the hours component is less than 24, and the months component is less than 12. DB2 converts each multiple of 60 seconds to one minute, each multiple of 60 minutes to one hour, each multiple of 24 hours to one day, and each multiple of 12 months to one year. For example, the following XPath expression invokes a constructor function that specifies a duration of 2 months, 63 days, 55 hours, and 91 minutes:

```
xs:duration("P2M63DT55H91M")
```

DB2 converts 55 hours to 2 days and 7 hours, and 91 minutes to 1 hour and 31 minutes. The expression returns the normalized duration value P2M65DT8H31M.

**xs:dayTimeDuration:**

The data type xs:dayTimeDuration represents a duration of time that is expressed by days, hours, minutes, and seconds components. xs:dayTimeDuration is derived from data type xs:duration.

The range that can be represented by this data type is from -P11574074073DT23H163M219.999999999999S to P11574074073DT23H163M219.999999999999S (or -999999999999999.999999999999 seconds to 999999999999999.999999999999 seconds).

The lexical form of xs:dayTimeDuration is P*n*DT*n*H*n*M*n*S, which is a reduced form of the ISO 8601 format. The following abbreviations describe this form:

**P**    The duration designator.

*n***D**   *n* is an unsigned integer that represents the number of days.

**T**    The date and time separator.

*n***H**   *n* is an unsigned integer that represents the number of hours.

*n***M**   *n* is an unsigned integer that represents the number of minutes.

*n***S**   *n* is an unsigned decimal that represents the number of seconds. If a decimal point appears, it must be followed by one to twelve digits that represent fractional seconds.

For example, the following form indicates a duration of 3 days, 10 hours, and 30 minutes:
```
P3DT10H30M
```

The following form indicates a duration of negative 120 days:
```
-P120D
```

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:
- If the number of days, hours, minutes, or seconds in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator must be present.
- The seconds part can have a decimal fraction.
- The designator T must be absent if and only if all of the time items are absent. The designator P must always be present.

For example, the following forms are allowed:

```
P13D
PT47H
P3DT2H
-PT35.89S
P4DT251M
```

The form `P-134D` is not allowed, but the form `-P1347D` is allowed.

DB2 stores xs:dayTimeDuration values in a normalized form. In the normalized form, the seconds and minutes components are less than 60, and the hours component is less than 24. DB2 converts each multiple of 60 seconds to one minute, each multiple of 60 minutes to one hour, and each multiple of 24 hours to one day. For example, the following XPath expression invokes a constructor function specifying a dayTimeDuration of 63 days, 55 hours, and 81 seconds:

```
xs:dayTimeDuration("P63DT55H81S")
```

DB2 converts 55 hours to 2 days and 7 hours, and 81 seconds to 1 minute and 21 seconds. The expression returns the normalized dayTimeDuration value P65DT7H1M21S.

**xs:yearMonthDuration:**

The data type xs:yearMonthDuration represents a duration of time that is expressed by the Gregorian year and month components. xs:yearMonthDuration is derived from data type xs:duration.

The range that can be represented by this data type is from -P83333333333333Y3M to P83333333333333Y3M (or -999999999999999 to 999999999999999 months).

The lexical form of xs:yearMonthDuration is P*n*Y*n*M, which is a reduced form of the ISO 8601 format. The following abbreviations describe this form:

**P**     The duration designator.

*n***Y**   *n* is an unsigned integer that represents the number of years.

*n***M**   *n* is an unsigned integer that represents the number of months.

An optional preceding minus sign (-) indicates a negative duration. If the sign is omitted, a positive duration is assumed.

For example, the following form indicates a duration of 1 year and 2 months:
```
P1Y2M
```

The following form indicates a duration of negative 13 months:
```
-P13M
```

Reduced precision and truncated representations of this format are allowed, but they must conform to the following requirements:
- The designator `P` must always be present.
- If the number of years or months in any expression equals zero, the number and its corresponding designator can be omitted. However, at least one number and its designator (`Y` or `M`) must be present.

For example, the following forms are allowed:
```
P1347Y
P1347M
```

The form `P-1347M` is not allowed, but the form `-P1347M` is allowed. The form `P24YM` is not allowed because `M` must have one preceding digit. `PY43M` is not allowed because `Y` must have at least one preceding digit.

DB2 stores xs:yearMonthDuration values in a normalized form. In the normalized form, the months component is less than 12. DB2 converts each multiple of 12 months to one year. For example, the following XPath expression invokes a constructor function that specifies a yearMonthDuration of 20 years and 30 months:

```
xs:yearMonthDuration("P20Y30M")
```

DB2 converts 30 months to 2 years and 6 months. The expression returns the normalized yearMonthDuration value P22Y6M.

## Casts between XML schema data types

You can use data type constructor functions to cast a value to a specific data type. Specify the value that you want to cast and the type to which you want to cast it.

The following table lists the compatible types for casting. You can cast values only of the listed input types to each target type.

*Table 50. Compatible types for casting*

| Target type | Source type | Comments |
|---|---|---|
| xs:string | Any type | • If the source type is xs:boolean, the result is 'true' or 'false'.<br><br>• If the source type is xs:integer, the result is the canonical lexical representation of the value, as defined in the XML Schema specification.<br><br>• If the source type is xs:decimal:<br><br>  – If the value has no significant digits after the decimal point, the decimal point and the zeroes that follow the decimal point are deleted, and the rules for casting from xs:integer apply.<br><br>  – Otherwise, the result is the canonical lexical representation of the value, as defined in the XML Schema specification.<br><br>• If the source type is xs:double:<br><br>  – If .000001<=*value*<=1000000, the value is converted to xs:decimal, and the rules for casting from xs:decimal apply.<br><br>  – If *value*=+0, or *value*=-0, the result is '0'.<br><br>  – Otherwise, the result is the canonical lexical representation of the value, as defined in the XML Schema specification.<br><br>• If the source type is xs:duration, xs:yearMonthDuration, or xs:dayTimeDuration, the result is the canonical lexical representation of the value.<br><br>• If the source type is xs:date, xs:dateTime, or xs:time, the result is the lexical representation of the value, with no adjustment for the time zone. If the value has no time zone, the result has no time zone. If the time zone is +00:00 or -00:00, the result has the UTC time zone "Z". |

*Table 50. Compatible types for casting (continued)*

| Target type | Source type | Comments |
|---|---|---|
| xs:boolean | xs:untypedAtomic, xs:string, xs:boolean, xs:double, xs:decimal, xs:integer | • If the source type is numeric, a value of 0 or NaN is cast to type xs:boolean with a value of `false`. All other numeric values are cast to type xs:boolean with a value of `true`.<br>• If the source type is xs:string or xs:untypedAtomic, the value "true" and the value "1" are cast to the xs:boolean value `true`. The value "false" and the value "0" are cast to the xs:boolean value `false`. All other values are invalid, and result in an error. |
| xs:decimal | Numeric types, xs:untypedAtomic, xs:string, xs:boolean | Values of numeric types are converted to a value that is within the set of possible values for type xs:decimal and is numerically closest to the source. If two values are equally close, the one that is closest to zero is chosen. The source value cannot be +INF, -INF, NaN, or outside of the range of type xs:decimal. For values of type xs:boolean, `true` is converted to 1.0, and `false` is converted to 0.0. |
| xs:double | Numeric types, xs:untypedAtomic, xs:string, xs:boolean | If the source is of type xs:decimal, or xs:integer, the cast is performed as xs:double(*SV* cast as xs:string) where *SV* is the source value. If the source is of type xs:boolean, `true` is cast to a value of 1.0E0, and `false` is cast to a value of 0.0E0. |
| xs:integer | Numeric types, xs:untypedAtomic, xs:string, xs:boolean | If the source type is a numeric type other than integer, the result is the source value with the fractional part discarded. The source cannot be outside of the range of type xs:integer. For values of type xs:boolean, `true` is converted to 1, and `false` is converted to 0. |
| xs:date | xs:dateTime, xs:untypedAtomic, xs:string | The time portion of the source value is not used in the conversion. |
| xs:time | xs:dateTime, xs:untypedAtomic, xs:string | The date portion of the source value is not used in the conversion. |
| xs:dateTime | xs:date, xs:untypedAtomic, xs:string | If the source type is xs:date, the time portion of the target value is the first moment of the day. The value is not adjustment for the time zone. |
| xs:duration | xs:dayTimeDuration, xs:yearMonthDuration, xs:untypedAtomic, xs:string | • If the source type is xs:dayTimeDuration, the target value has the same days, hours, minutes and seconds components as the source value. The year component and the month component of the target value are 0.<br>• If the source type is xs:yearMonthDuration, the target value has the same years and months components as the source value. The days, hours, minutes and seconds components are 0. |

*Table 50. Compatible types for casting  (continued)*

| Target type | Source type | Comments |
|---|---|---|
| xs:dayTimeDuration | xs:duration, xs:untypedAtomic, xs:string | A cast from xs:duration to xs:dayTimeDuration results in information loss. To avoid information loss, cast the xs:duration value to an xs:yearMonthDuration value and an xs:dayTimeDuration value and work with both values. |
| xs:yearMonthDuration | xs:duration, xs:untypedAtomic, xs:string | A cast from xs:duration to xs:yearMonthDuration results in information loss. To avoid information loss, cast the xs:duration value to an xs:yearMonthDuration value and an xs:dayTimeDuration value and work with both values. |

## Example

The following XPath expression returns purchase orders that contain more than one item. The xs:integer constructor function casts the value of the quantity element to an integer. That integer can then be compared to the integer 1.

```
declare namespace ipo="http://www.example.com/IPO";
/ipo:purchaseOrder[items/item/quantity/xs:integer(.) > 1]
```

When an xs:untypedAtomic is compared with an integer, DB2 converts both operands to type xs:double to make the numeric comparison. The cast ensures that the values are compared as values of type xs:integer.

# XPath prologs and expressions

In DB2 XPath, an XPath expression consists of an optional prolog that is followed by an expression. The *prolog* contains a series of declarations that define the processing environment for the expression. The *expression* consists of an expression that defines the result of the XPath expression.

## Syntax

The following diagrams show the general format of an XPath expression.

### XPath expression

```
►►─┬──────────┬─expression──────────────────────────────────────────►◄
   └─prolog──┘
```

### expression:

```
├──┬─primary-expression──────┬─────────────────────────────────────────┤
   ├─logical-expression──────┤
   ├─comparison-expression───┤
   ├─arithmetic-expression───┤
   └─path-expression─────────┘
```

### prolog:

```
┌──────────────────────────────────────────────┐
│ ▼ ┌─namespace-declaration─────────┐           │
├───┤                               ├───────────────────┤
    └─default-namespace-declaration─┘
```
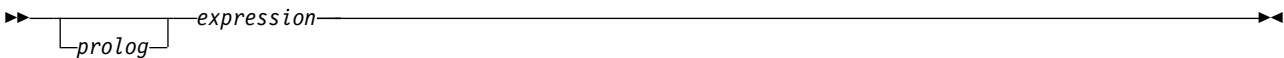
## Example

The following example illustrates the structure of a typical expression in DB2 XPath. In this example, the prolog contains a namespace declaration that binds the prefix `ipo` to a URI. The expression body contains an expression that returns one row for each `ipo:purchaseOrder` element with XML documents stored in the XMLPO column. A predicate is used to specify that the name attribute on the shipTo node is "Jane" and the name attribute on the billTo node is "Jason".

```
SELECT X.* FROM T1, XMLTABLE ('declare namespace ipo="http://www.example.com/IPO";
    /ipo:purchaseOrder[shipTo/@name = "Jane" and billTo/@name = "Jason"]'
    PASSING T1.XMLPO) X;
```

*Figure 1. Structure of a typical expression in DB2 XPath*

## Prologs

The *prolog* consists of a declaration that defines the processing environment for an XPath expression. A declaration in the prolog is followed by a semicolon (;). The prolog is an optional part of the XPath expression.

The prolog can contain zero or more namespace declarations and zero or one default namespace declarations.

**Namespace declarations:**

A *namespace declaration* is an optional declaration in the XPath expression prolog that declares a namespace prefix and associates the prefix with a namespace URI.

The declaration adds the prefix-URI pair to the set of statically known namespaces for the expression. The *statically known namespaces* include all of the namespaces that are known during the static processing of an expression. The namespace declaration is in scope throughout the XPath expression in which it is declared. Multiple declarations of the same namespace prefix in the query prolog result in an error.

**Restriction:** The prefixes `xmlns` and `xml` are reserved and cannot be specified as a prefix in a namespace declaration.

**Syntax**

**namespace-declaration**

```
►►──declare──namespace──prefix──=──stringLiteral──;──────────────────────────────►◄
```

*prefix*
> Specifies a namespace prefix that is bound to the URI. The namespace prefix is used in qualified names (QNames) to identify the namespace for an element, attribute, data type, or function.

*stringLiteral*
> Specifies a string literal that represents the URI to which the prefix is bound. The string literal value must be a valid URI and cannot be a zero-length string.
>
> You can override predeclared namespace prefixes by specifying a namespace declarations for those prefixes. However, you cannot override the URI that is associated with the prefix `xml`.

The string literal cannot be http://www.w3.org/XML/1998/namespace or http://www.w3.org/2000/xmlns/.

**Example**

The following namespace declaration declares the namespace prefix `ns1` and associates it with the namespace URI `http://posample.org`:

```
declare namespace ns1 = "http://posample.org";
/ns1:purchaseOrder[shipTo/name = "Jane" and billTo/name = "Jason"]
```

When the expression in the example executes, the namespace prefix ns1 is associated with the namespace URI http://posample.org. The instance of the purchase order document to which the expression refers is the instance with the namespace URI `http://posample.org`.

**Default namespace declarations:**

*Default namespace declarations* are optional declarations in the XPath expression prolog that specify the namespaces to use for unprefixed QNames (qualified names).

An XPath expression prolog can include a default element namespace declaration.

The default element namespace declaration specifies a namespace URI that is used for unprefixed element names. The XPath expression prolog can contain one default element namespace declaration only. This declaration is in scope throughout the expression in which it is declared. If no default element namespace is declared, unqualified element names are in no namespace.

**Syntax**

**default-namespace-declaration**

►►──declare──default──element──namespace──*stringLiteral*──;──────────────────────────►◄

**namespace**
> Specifies a string literal that represents the URI for the namespace. The string literal must be a valid URI or a zero-length string.
>
> If *namespace* is a zero-length string, unprefixed element names are in no namespace.
>
> The string literal cannot be http://www.w3.org/XML/1998/namespace or http://www.w3.org/2000/xmlns/.

**Example**

The following declaration specifies that the default namespace for element names is the namespace that is associated with the URI `http://posample.org`

```
declare default element namespace "http://posample.org";
```

When the query in the example executes, all element nodes in this expression (purchaseOrder, shipTo, billTo, and name) are associated with the namespace URI `http://posample.org`.

```
declare default element namespace "http://posample.org";
/purchaseOrder[shipTo/name = "Jane" and billTo/name = "Jason"]
```

When the expression in the example executes, the namespace URI http://posample.org is associated with all unprefixed element names in the expression.

# Expression evaluation and processing

A number of operations are often included in the processing of expressions. These operations include extracting atomic values from nodes and using type promotion and subtype substitution to obtain values of an expected type.

**Atomization:**

*Atomization* is the process of converting a sequence of items into a sequence of atomic values. Atomization is used by expressions whenever a sequence of atomic values is required.

Each item in a sequence is converted to an atomic value by applying the following rules:
- If the item is an atomic value, then the atomic value is returned.
- If the item is a node, its typed value is returned. The *typed value* of a node is a sequence of zero or more atomic values that can be extracted from the node. If the node has no typed value, then an error is returned. If an XML document is validated with a schema, DB2 for i does not keep the type annotation for each node. The data is always stored as untyped. The typed value of an untyped element or attribute is the string value as an instance of xs:untypedAtomic.

Implicit atomization of a sequence produces the same result as invoking the fn:data function explicitly on a sequence.

For example, the following sequence contains a combination of nodes and atomic values:
```
("Some text", <anElement>More text</anElement>, 1001)
```

Applying atomization to this sequence results in the following sequence of atomic values:
```
("Some text", "More text", 1001)
```

The following XPath expressions use atomization to convert items into atomic values:
- Arithmetic expressions
- Comparison expressions
- Function calls with arguments whose expected types are atomic

**Type promotion:**

*Type promotion* is a process that converts an atomic value from its original type to the type that is expected by an expression. XPath uses type promotion during the evaluation of function calls and operators that accept numeric or string operands.

XPath permits type promotion and subtype substitution. Type promotion and subtype substitution differ in the following ways:
- For type promotion, the atomic value is actually converted from its original type to the type that is expected by an expression.
- For subtype substitution, an expression that expects a specific type can be invoked with a value that is derived from that type. However, the value retains its original type.

**Numeric type promotion:**
> A value of type xs:decimal (or any type that is derived by restriction from xs:decimal) can be promoted to xs:double. The result of this promotion is created by casting the original value to the required type.
>
> In the following example, the xs:double value 13.54e-2 is added to the xs:decimal value 100. The xs:decimal is promoted to xs:double to do the arithmetic and a result with a type of xs:double is returned:
> ```
> xs:double(13.54e-2) + xs:decimal(100)
> ```

**Subtype substitution:**

*Subtype substitution* is the use of a value whose type is derived from an expected type.

Subtype substitution does not change the actual type of a value. For example, if an xs:integer value is used where an xs:decimal value is expected, the value retains its type as xs:integer.

Subtype substitution is used whenever a value that is derived from an expected type is passed to an expression.

In the following example, an xs:dayTimeDuration is substituted for an xs:duration value that is expected by the fn:hours-from-duration function.

```
fn:hours-from-duration(xs:dayTimeDuration("PT2H"))
```

## Primary expressions

A primary expression contains one of the following types of items: literal, variable reference, parenthesized expression, context item expression, or function call.

### Syntax

**primary-expression:**

```
├──┬─literal────────────────┬──────────────────────────────────────────────────┤
   ├─variable-reference──────┤
   ├─parenthesized-expression─┤
   ├─context-item-expression──┤
   └─function-call───────────┘
```

**Literals:**

DB2 XPath supports two kinds of literals: numeric literals and string literals.

A *numeric literal* is an atomic value of type `xs:integer`, `xs:decimal`, or `xs:double`. A numeric literal that contains no decimal point (.) and no e or E character is an atomic value of the type `xs:integer`. A numeric literal that contains a decimal point (.), but no e or E character is an atomic value of type `xs:decimal`. A numeric literal that contains an e or E character is an atomic value of type `xs:double`. Values of numeric literals are interpreted according to the rules of XML Schema.

A *string literal* is an atomic value of type `xs:string` that is enclosed in delimiting apostrophes or quotation marks. String literals can include predefined entity references and character references.

To include an apostrophe within a string literal that is delimited by apostrophes, specify two adjacent apostrophes. Similarly, to include a quotation mark within a string literal that is delimited by quotation marks, specify two adjacent quotation marks.

If a string literal is used in an XPath expression within the value of an XML attribute, the characters that are used to delimit the literal must be different from the characters that are used to delimit the attribute.

### Examples

*Example of XPath expressions with numeric literals:*

```
'7635'
'8735.98834'
'93948.87E+77'
```

*Example of an XPath expression that contains a string literal with an embedded double quotation mark:*

```
SELECT X.* FROM X1,
      XMLTABLE('$inp/purchaseOrder[contains(., "string literal double-quote "" in the middle")]'
                        PASSING X1.XMLPO as "inp") X;
```

*Predefined entity references:*

A *predefined entity reference* is a short sequence of characters that represents a character that has some syntactic significance in DB2 XPath.

A predefined entity reference begins with an ampersand (&) and ends with a semicolon (;). When a string literal is processed, each predefined entity reference is replaced by the character that it represents. The following table lists the predefined entity references that DB2 XPath recognizes:

*Table 51. Predefined entity references in DB2 XPath*

| Entity reference | Character represented |
| --- | --- |
| &lt; | < |
| &gt; | > |
| &amp; | & |
| &quot; | " |
| &apos; | ' |

*Character references:*

A *character reference* is an XML-style reference to a Unicode character that is identified by its decimal or hexadecimal code point.

A character reference begins with either &#x or &# and ends with a semicolon (;). If the character reference begins with &#x, the digits and letters up to the terminating semicolon (;) provide a hexadecimal representation of the character's code point in ISO/IEC 10646. If the character reference begins with &#, the digits up to the terminating semicolon (;) provide a decimal representation of the character's code point.

**Example**

The character reference &#8364; represents the Euro symbol (€).

**Variable references in DB2 XPath:**

A variable reference is a QName that is preceded by a dollar sign ($). When an XPath expression is evaluated, each variable reference resolves to the value of the expression that is bound to the variable.

Every variable reference must match a name in the in-scope variables for the XPath expression. In-scope variables are bound from the SQL context that invokes the XPath expression, such as variables defined in the row argument expression of XMLTABLE.

Two variable references are equivalent if their local names are the same and their namespace prefixes are bound to the same namespace URI in the in-scope namespaces. A variable reference with no prefix is in no namespace. DB2 for i does not allow a namespace prefix to be specified for a variable name.

**Examples**

In the following example, the XMLTABLE function binds the value of the host variable :IHV to $PARTNUMBER, and the value of column C1 to $QTY.

```
SELECT X.* FROM T1, XMLTABLE('//item[@partNum = $PARTNUMBER and quantity=$QTY]'
  PASSING T1.XMLPO, :IHV AS PARTNUMBER, T1.C1 AS QTY) X;
```

**Parenthesized expression:**

Parentheses can be used to enforce a particular order of evaluation in expressions that contain multiple
operators.

Use a parenthesized expression to explicitly specify the order of operations in a complex arithmetic
expression.

Empty parentheses are used to denote an empty sequence.

**Syntax**

**parenthesized-expression:**



**Examples**

In the following example, the parenthesized expressions 5+5 and 6+4 are evaluated first.
```
((5+5) * (6+4)) div 5
```

The result is 20.

**Context item expressions:**

A context item expression consists of a single period (.). A context item expression evaluates to the item
that is currently being processed, which is known as the *context item*. The context item can be either a
node or an atomic value.

**Example**

The following example contains a context item expression that identifies nylon pants in the products
document:
```
declare namespace ipo="http://www.example.com/IPO";
 /ipo:products/product/name[. = "Nylon pants"]
```

**Function calls:**

DB2 XPath supports calls to built-in XPath functions.

Built-in XPath functions are in the namespace http://www.w3.org/2003/11/xpath-functions. If the
function name in the function call has no namespace prefix, the function is considered to be in the default
function namespace.

DB2 XPath uses the following process to evaluate functions:
1. DB2 XPath evaluates each expression that is passed as an argument in the function call and returns a
   value for each expression.
2. The value that is returned for each argument is converted to the data type that is expected for that
   argument. When the expected type is a sequence of zero or more atomic types, DB2 XPath uses the
   following rules to convert the value to its expected type:
   a. The given value is atomized into a sequence of atomic values.

b. Each item in the atomic sequence that is of type `xs:untypedAtomic` is cast to the expected atomic type. For built-in functions where the expected type is specified as `numeric`, arguments of type `xs:untypedAtomic` are cast to `xs:double`.

c. Numeric type promotion is applied to any numeric item in the atomic sequence that can be promoted to the expected atomic type through numeric type promotion. Numeric items include items of type `xs:integer`, `xs:decimal`, or `xs:double`.

3. The function is evaluated using the converted values of its arguments. The result of the function call is either an instance of the function's declared return type or an error.

**Example**

The following example retrieves the first three characters of the pid attribute of a product document:

```
declare namespace pos="http://posample.org";
  fn:substring(/pos:product/@pid, 1, 3)
```

## Path expressions

*Path expressions* locate nodes within an XML tree. Path expressions in DB2 XPath are based on the syntax of XPath 2.0.

A path expression consists of a series of one or more steps that are separated by either a slash character (/) or two slash characters (//). The path expression can begin with a slash character (/), two slash characters(//), or a step.

A slash character (/) is used to separate individual steps. Two slash characters (//) in a path expression are expanded as /descendant-or-self::node()/, which leaves a sequence of steps separated by a slash character (/). Each step generates a sequence of items.

The steps in a path expression are evaluated from left to right. The sequence of items that a step generates are used as context nodes for the step that follows. For example, in the expression `description/name`, the first step generates a sequence of nodes that includes all `description` elements. The final step evaluates the `name` step once for each `description` item in the sequence. Each time a `name` step is evaluated, it is evaluated with a different focus, until all description items have been evaluated. The sequences that result from each evaluation of the step are merged together in document order, and duplicate nodes are eliminated based on node identity.

Although the result of an XPath step expression is determined by evaluating the steps and predicates from left to right, DB2 might perform the evaluation in a more efficient order. In some instances, this can change which errors are signaled.

A slash character (/) at the beginning of a path expression means that the path is to begin at the root node of the tree that contains the context node. That root node must be a document node.

**Recommendation:** Because the slash character can be used as both an operator and an operand, use parentheses to clarify the meaning of the slash character when it is used as the first character of an operator. For example, to specify an empty path expression as the left operand of a multiplication operation use (/)*5 instead of /*5. The later expression causes an error. Because path expressions have the higher precedence, DB2 interprets this expression as a path expression with a wildcard for a name test (/*) that is followed by the token 5.

Two slash characters (//) at the beginning of a path expression establishes an initial node sequence that contains the root of the tree in which the context node is found and all nodes descended from this root. This node sequence is used as the input to subsequent steps in the path expression. That root node must be a document node.

The value of the path expression is the combined sequence of items that results from the final step in the path. This value is a sequence of nodes or an atomic value. A path expression that returns a mixture of nodes and atomic values results in an error.

A step consists of an axis step or a filter expression.

## Syntax

**path-expression:**

```
            ┌──────────────────────┐
├──┬────┬──step──┬─▼────────────┬───────────────────────────────────────┤
   ├─/──┤        ├─/──step──────┤
   └─//─┘        └─//──step─────┘
```

**step:**

```
├──┬─filter-expression─┬─────────────────────────────────────────────────┤
   └─axis-step─────────┘
```
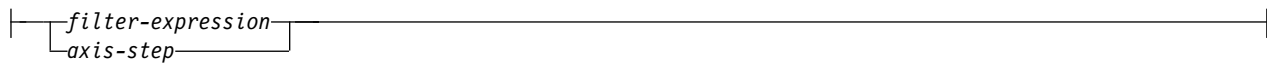
## Example

Use a path expression to determine which stocks have at least one bid for which the price is greater than the price of some offer on that stock.

```
//stock[bid/xs:double(price) > offer/price]/@stock_id
```

**Axis steps:**

An *axis step* consists of three parts: an optional axis, a node test, and zero or more predicates.

The *node test* specifies the criteria used to select nodes. The *predicates* filter the sequence that is returned by the axis step.

The result of an axis step is always a sequence of zero or more nodes, and these nodes are returned in document order. An axis step can be either a *forward step*, which starts at the context node and moves down through the XML tree, or a *reverse step*, which starts at the context node and moves up through the XML tree. If the context item is not a node, then the expression results in a type error.

The unabbreviated syntax for an axis step consists of an axis name and node test that are separated by a double colon. The syntax of an axis expression can be abbreviated by omitting the axis and using shorthand notations.

**Syntax**

**axis-step:**
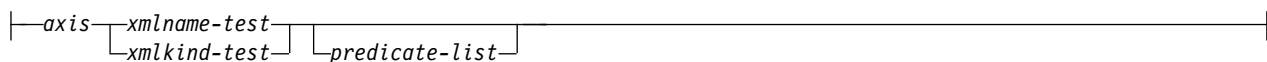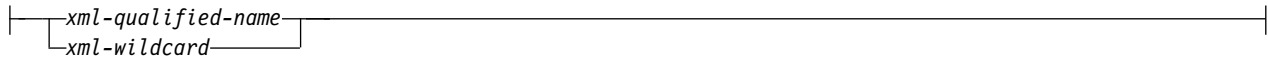
```
├──axis──┬─xmlname-test─┬──┬───────────────┬─────────────────────────────┤
         └─xmlkind-test─┘  └─predicate-list─┘
```
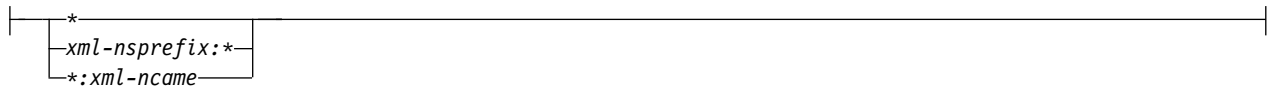
**axis:**

```
  ┌─child─::─────────────┐
├─┼──────────────────────┼──────────────────────────────────────────┤
  │ ┌─@──────────────────┤
  ├─attribute─::─────────┤
  ├─descendant─::────────┤
  ├─self─::──────────────┤
  ├─descendant-or-self─::─┤
  └─parent─::────────────┘
```

**xmlname-test:**

```
  ┌─xml-qualified-name─┐
├─┼────────────────────┼───────────────────────────────────────────┤
  └─xml-wildcard───────┘
```

**xml-wildcard:**

```
  ┌─*─────────────┐
├─┼───────────────┼─────────────────────────────────────────────────┤
  ├─xml-nsprefix:*─┤
  └─*:xml-ncame───┘
```

**xmlkind-test:**

```
├─┬─document-node─(─────────────────────────)───────────────────────┬─┤
  │               └─element─(─┬─name─┬─)─┘                           │
  │                           └─*────┘                               │
  ├─element─(─────────────)─────────────────────────────────────────┤
  │          ├─name─┤                                                │
  │          └─*────┘                                                │
  ├─attribute─(───────────)─────────────────────────────────────────┤
  │            ├─name─┤                                              │
  │            └─*────┘                                              │
  ├─processing instruction─(─┬─xml-ncname────────────────────────┬─)─┤
  │                          └─processing-instruction-literal-string─┘│
  ├─comment─(─)─────────────────────────────────────────────────────┤
  ├─text─(─)────────────────────────────────────────────────────────┤
  └─node─(─)────────────────────────────────────────────────────────┘
```

**predicate-list:**

```
  ┌────────────────┐
  │ ▼              │
├─┴─[─expression─]─┴────────────────────────────────────────────────┤
```
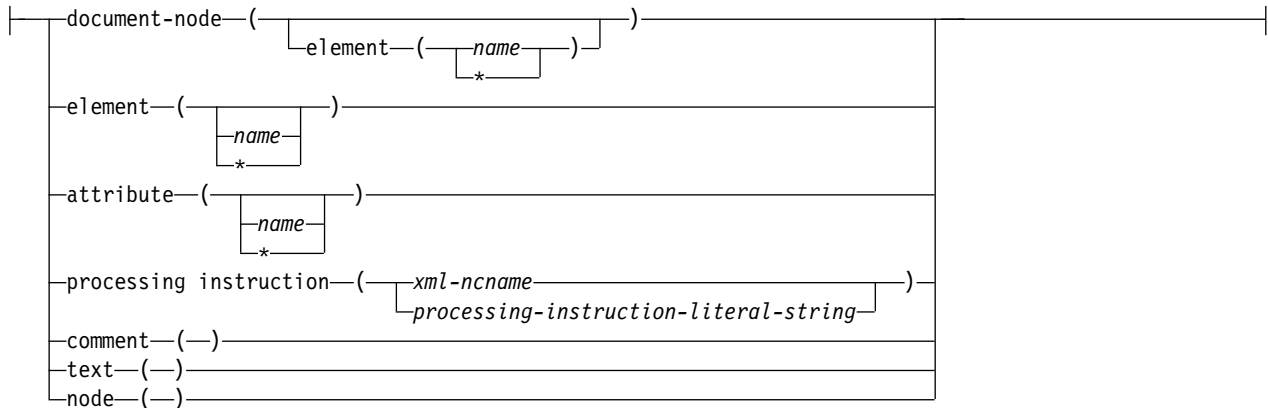
### Example

In the following example, `child` is the name of the axis and `price` is the name of the element nodes to be selected on this axis.

```
child::price
```

The axis step in this example selects all `price` elements that are children of the context node.

*Axes:*

An *axis* is an optional part of an axis step that specifies a direction of movement through an XML document.

Table 52 describes the axes that are supported in DB2 XPath.

*Table 52. Supported axes in DB2 XPath*

| Axis | Description | Notes |
| --- | --- | --- |
| child | Returns the children of the context node. This axis is the default. | Document nodes and element nodes are the only nodes that have children. If the context node is any other kind of node, or if the context node is an empty document or element node, then the child axis is an empty sequence. The children of a document node or element node may be element, processing instruction, comment, or text nodes. Attribute and document nodes can never appear as children. |
| descendant | Returns the descendants of the context node (the children, the children of the children, and so on). | |
| attribute | Returns the attributes of the context node. | This axis is empty if the context node is not an element node. |
| self | Returns the context node only. | |
| descendant-or-self | Returns the context node and the descendants of the context node. | |
| parent | Returns the parent of the context node, or an empty sequence if the context node has no parent. | An element node can be the parent of an attribute node even though an attribute node is never a child of an element node. |

DB2 XPath does not support the additional axes defined by the full axis feature of the W3 standard.

An axis can be either a forward or reverse axis. A *forward axis* contains the context node and nodes that are after the context node in document order. A *reverse axis* contains the context node and nodes that are before the context node in document order. In DB2 XPath, the forward axes include: child, descendant, attribute, self, and descendant-or-self. The only supported reverse axis is the parent axis.

When an axis step selects a sequence of nodes, each node is assigned a context position that corresponds to its position in the sequence. If the axis is a forward axis, context positions are assigned to the nodes in document order, starting with 1. If the axis is a reverse axis, context positions are assigned to the nodes in reverse document order, starting with 1.

*Node tests:*

A *node test* is a condition that must be true for each node that is selected by an axis step. The node test can be expressed as a name test or a kind test. A *name test* selects nodes based on the name of the node. A *kind test* selects nodes based on the kind of node.

**Name tests**

A name test consists of a QName or a wildcard. When a name test is specified in an axis step, the step selects the nodes on the specified axis that match the QName or wildcard. If the name test is specified on the attribute axis, the step selects any attributes that match the name test. Otherwise, on all other axes, the step selects any elements that match the name test. For the QNames to match, the expanded QName

of the node must be equal (on a codepoint basis) to the expanded QName that is specified in the name test. Two expanded QNames are equal if their namespace URIs are equal and their local names are equal (even if their namespace prefixes are not equal).

**Important:** Any prefix that is specified in a name test must correspond to one of the statically known namespaces for the expression. For name tests that are performed on the attribute axis, unprefixed QNames have no namespace URI. For name tests that are performed on all other axes, unprefixed QNames have the namespace URI of the default element namespace.

Table 53 describes the name tests that are supported in DB2 XPath.

*Table 53. Supported name tests in DB2 XPath*

| Test | Description | Examples |
|------|-------------|----------|
| *QName* | Matches any nodes (on the specified axis) whose QName is equal to the specified QName. If the axis is an attribute axis, this test matches attribute nodes that are equal to the specified QName. On all other axes, this test matches element nodes that are equal to the specified QName. | In the expression `child::para`, the name test `para` selects all of the `para` elements on the child axis. |
| * | Matches all nodes on the specified axis. If the axis is an attribute axis, this test matches all attribute nodes. On all other axes, this test matches all element nodes. | In the expression, `child::*`, the name test * matches all elements on the child axis. |

**Kind tests**

When a kind test is specified in an axis step, the step selects only those nodes on the specified axis that match the kind test. Table 54 describes the kind tests that are supported in DB2 XPath.

*Table 54. Supported kind tests in DB2 XPath*

| Test | Description | Examples |
|------|-------------|----------|
| node() | Matches any node on the specified axis. | In the expression `child::node()`, the kind test `node()` selects any nodes on the child axis. |
| text() | Matches any text node on the specified axis. | In the expression `child::text()`, the kind test `text()` selects any text nodes on the child axis. |
| comment() | Matches any comment node on the specified axis. | In the expression `child::comment()`, the kind test `comment()` selects any comment nodes on the child axis. |
| processing-instruction(*NCName*) | Matches any processing-instruction node (on the specified axis) whose name (called its "PITarget" in XML) matches the NCName that is specified in this name test. | In the expression `child::processing-instruction( xml-stylesheet)`, the kind test `processing-instruction( xml-stylesheet)` selects any processing instruction nodes on the child axis whose PITarget is xml-stylesheet. |
| processing-instruction(*StringLiteral*) | Matches any processing-instruction node (on the specified axis) whose name matches the string literal that is specified in this test.  This node test provides backwards compatibility with XPath 1.0. | In the expression `child::processing-instruction("xml-stylesheet")`, the kind test `processing-instruction("xml-stylesheet")` selects any processing instruction nodes on the child axis whose PITarget is xml-stylesheet. |

*Table 54. Supported kind tests in DB2 XPath  (continued)*

| Test | Description | Examples |
|---|---|---|
| element() | Matches any element node on the specified axis. | In the expression `child::element()`, the kind test `element()` selects any element nodes on the child axis. |
| element(*QName*) | Matches any element node (on the specified axis) whose name matches the qualified name that is specified in this test. | In the expression `child::element("price")`, the kind test `element("price")` selects any element nodes on the child axis whose name is price. |
| element(*) | Matches any element node on the specified axis. | In the expression `child::element(*)`, the kind test `element(*)` selects any element nodes on the child axis. |
| attribute() | Matches any attribute node on the specified axis. | In the expression `child::attribute()`, the kind test `attribute()` selects any attribute nodes on the child axis. |
| attribute(*QName*) | Matches any attribute node (on the specified axis) whose name matches the qualified name that is specified in this test. | In the expression `child::attribute("price")`, the kind test `attribute("price")` selects any attribute nodes on the child axis whose name is price. |
| attribute(*) | Matches any attribute node on the specified axis. | In the expression `child::attribute(*)`, the kind test `attribute(*)` selects any attribute nodes on the child axis. |
| document-node() | Matches any document node on the specified axis. | In the expression `self::document-node()`, the kind test `document-node()` selects any document nodes on the self axis. |
| document-node(element(*QName*)) | Matches any document node on the specified axis that has only one element node. | In the expression `self::document-node(element("price"))`, the kind test `document-node(element("price"))` selects any document nodes on the self axis that have a single root element whose name is price. |
| document-node(element(*)) | Matches any document node on the specified axis that has element nodes. | In the expression `self::document-node(element(*))`, the kind test `document-node(element(*))` selects any document nodes on the self axis that have element nodes. |

*Predicates:*

A *predicate* consists of an expression, called a predicate expression, that is enclosed in square brackets ([]). A predicate filters a sequence by retaining some items and discarding others.

The predicate expression is evaluated once for each item in the sequence. The result of the predicate expression is an `xs:boolean` value called the *predicate truth value*. Those items for which the predicate truth value is `true` are retained, and those for which the predicate truth value is `false` are discarded.

The value of the predicate expression can be a numeric value as long as its static type is a numeric singleton. When the static type of the predicate expression is a numeric singleton, the predicate truth value is `true` if the position of the context item within the sequence of items being evaluated matches the numeric singleton. In other words: `child::employee/child::address[2]` is equivalent to

`child::employee/child::address[fn:position() = 2]` and returns the second address under `employee`. If the predicate expression is a numeric that cannot be determined to be a singleton when the expression is parsed, a not supported error will be signaled.

For all other data types, the predicate truth value is the effective boolean value of the predicate expression. The effective boolean value is `false` if the predicate expression evaluates to an empty sequence or `false`. Otherwise, the effective boolean value is `true`.

If a predicate is used to filter an atomic value or a function call, a not supported error may be signaled.

**Examples**

The following examples are axis steps that include predicates:
- `descendant::phone[attribute::type = "work"]` selects all the descendants of the context node that are elements named `phone` and whose type attribute has the value "work".
- `child::address[prov-state][pcode-zip]` selects all the `address` children of the context node that have a `prov-state` child element and a `pcode-zip` child element.

**Abbreviated syntax for path expressions:**

DB2 XPath provides an abbreviated syntax for expressing axes in path expressions.

Table 55 describes the abbreviations that are allowed in path expressions.

*Table 55. Abbreviated syntax for path expressions*

| Abbreviated syntax | Description |
| --- | --- |
| no axis specified | Shorthand abbreviation for `child::`, except when the axis step specifies `attribute()` for the node test. When the axis step specifies an attribute test, an omitted axis is shorthand for `attribute::`. |
| @ | Shorthand abbreviation for `attribute::` . |
| // | Shorthand abbreviation for `/descendant-or-self::node()/`, except when this abbreviation appears at the beginning of the path expression.<br><br>When this abbreviation appears at the beginning of the path expression, the axis step selects an initial node sequence that contains the root of the tree in which the context node is found, plus all nodes that are descended from this root. This expression raises an error if the root node is not a document node. |
| .. | Shorthand abbreviation for `parent::node()`. |

**Examples of abbreviated and unabbreviated syntax**

Table 56 provides examples of abbreviated and unabbreviated syntax.

*Table 56. Unabbreviated and abbreviated syntax compared*

| Unabbreviated syntax | Abbreviated syntax | Result |
| --- | --- | --- |
| `child::para` | `para` | Selects the `para` elements that are children of the context node. |
| `child::*` | `*` | Selects all elements that are children of the context node. |
| `child::text()` | `text()` | Selects all text nodes that are children of the context node. |

*Table 56. Unabbreviated and abbreviated syntax compared  (continued)*

| Unabbreviated syntax | Abbreviated syntax | Result |
|---|---|---|
| `child::node()` | `node()` | Selects all of the children of the context node. This expression returns no attribute nodes, because attributes are not children of a node. |
| `attribute::name` | `@name` | Selects the name attribute of the context node |
| `attribute::*` | `@*` | Selects all of the attributes of the context node. |
| `child::para[attribute::type="warning"]` | `para[@type="warning"]` | Selects all para children of the context node that have a type attribute with the value warning. |
| `child::chapter[child::title="Introduction"]` | `chapter[title="Introduction"]` | Selects the chapter children of the context node that have one or more title children whose typed value is equal to the string Introduction. |
| `child::chapter[child::title]` | `chapter[title]` | Selects the chapter children of the context node that have one or more title children. |

## Filter expressions

A filter expression consists of a primary expression that is followed by zero or more predicates. The predicates, if present, filter the result of the primary expression.

The result of a filter expression consists of all the items that are returned by the primary expression for which all the predicates are true. If no predicates are specified, the result is the result of the primary expression. This result can contain nodes, atomic values, or a combination of nodes and atomic values. The ordering of the items that are returned by a filter expression is the same as their order in the result of the primary expression. Context positions are assigned to items based on their ordinal position in the result sequence. The first context position is 1.

### Syntax

**filter-expression**

```
►►──primary-expression──┬──────────────────┬──────────────────────────────►◄
                        └──predicate-list──┘
```
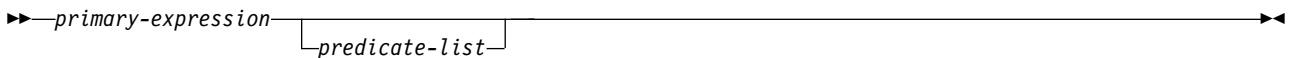
### Examples

The following example uses a filter expression that returns the value of $x if there is a customerinfo element anywhere in the document that is specified by $x:

```
declare default element namespace "http://posample.org";
  $x[.//customerinfo]
```

## Arithmetic expressions

Arithmetic expressions perform operations that involve addition, subtraction, multiplication, division, and modulus.

The following table describes the arithmetic operators and lists them in order of operator precedence from highest to lowest. Unary operators have a higher precedence than binary operators unless parentheses are used to force the evaluation of the binary operator.

*Table 57. Arithmetic operators in DB2 XPath*

| Operator | Purpose | Associativity |
|---|---|---|
| -(unary), +(unary) | negates value of operand, maintains value of operand | right-to-left |
| *, div, idiv, mod | multiplication, division, integer division, modulus | left-to-right |
| +, - | addition, subtraction | left-to-right |

**Note:** A subtraction operator must be preceded by whitespace if the operator could otherwise be interpreted as part of a previous token. For example, `a-b` is interpreted as a name, but `a - b` and `a -b` are interpreted as arithmetic operations.

The result of an arithmetic expression is one of the following items:

- A numeric value
- A date or time value
- A duration value
- An empty sequence
- An error

DB2 XPath uses the following process to evaluate an arithmetic expression.

1. Atomizes each operand into a sequence of atomic values.
2. Uses the following rules to evaluate the operands in the arithmetic expression:
    - If the atomized operand is an empty sequence, the result of the arithmetic expression is an empty sequence.
    - If the atomized operand is a sequence that contains more than one value, an error is returned.
    - If the atomized operand is an untyped atomic value (xs:untypedAtomic), DB2 XPath casts the value to xs:double. If the cast fails, DB2 XPath returns an error.
3. If the types of the operands are a valid combination for the arithmetic operator, DB2 XPath applies the operator to the atomized values. The result of this operation is an atomic value or a dynamic error (for example, an error might result from dividing an xs:integer by zero).
4. If the types of the operands are not a valid combination for the arithmetic operator, DB2 XPath raises a type error.

The following table identifies valid combinations of types for arithmetic operators. In this table, the letter A represents the first operand in the expression, and the letter B represents the second operand. The term numeric denotes the types xs:integer, xs:decimal, xs:double, or any types derived from one of these types. If the result type of an operator is listed as numeric, the result type will be the first type in the ordered list (xs:integer, xs:decimal, xs:double) into which all operands can be converted by subtype substitution and type promotion.

*Table 58. Valid types for operands of arithmetic expressions*

| Operator with operands | Type of operand A | Type of operand B | Result type |
|---|---|---|---|
| A + B | numeric | numeric | numeric |
| | xs:date | xs:yearMonthDuration | xs:date |
| | xs:yearMonthDuration | xs:date | xs:date |
| | xs:date | xs:dayTimeDuration | xs:date |
| | xs:dayTimeDuration | xs:date | xs:date |
| | xs:time | xs:dayTimeDuration | xs:time |
| | xs:dayTimeDuration | xs:time | xs:time |
| | xs:dateTime | xs:yearMonthDuration | xs:dateTime |
| | xs:yearMonthDuration | xs:dateTime | xs:dateTime |
| | xs:dateTime | xs:dayTimeDuration | xs:dateTime |
| | xs:dayTimeDuration | xs:dateTime | xs:dateTime |
| | xs:yearMonthDuration | xs:yearMonthDuration | xs:yearMonthDuration |
| | xs:dayTimeDuration | xs:dayTimeDuration | xs:dayTimeDuration |
| A - B | numeric | numeric | numeric |
| | xs:date | xs:date | xs:dayTimeDuration |
| | xs:date | xs:yearMonthDuration | xs:date |
| | xs:date | xs:dayTimeDuration | xs:date |
| | xs:time | xs:time | xs:dayTimeDuration |
| | xs:time | xs:dayTimeDuration | xs:time |
| | xs:dateTime | xs:dateTime | xs:dayTimeDuration |
| | xs:dateTime | xs:yearMonthDuration | xs:dateTime |
| | xs:dateTime | xs:dayTimeDuration | xs:dateTime |
| | xs:yearMonthDuration | xs:yearMonthDuration | xs:yearMonthDuration |
| | xs:dayTimeDuration | xs:dayTimeDuration | xs:dayTimeDuration |
| A * B | numeric | numeric | numeric |
| | xs:yearMonthDuration | numeric | xs:yearMonthDuration |
| | numeric | xs:yearMonthDuration | xs:yearMonthDuration |
| | xs:dayTimeDuration | numeric | xs:dayTimeDuration |
| | numeric | xs:dayTimeDuration | xs:dayTimeDuration |
| A idiv B | numeric | numeric | xs:integer |
| A div B | numeric | numeric | numeric; but xs:decimal if both operands are xs:integer |
| | xs:yearMonthDuration | numeric | xs:yearMonthDuration |
| | xs:dayTimeDuration | numeric | xs:dayTimeDuration |
| | xs:yearMonthDuration | xs:yearMonthDuration | xs:decimal |
| | xs:dayTimeDuration | xs:dayTimeDuration | xs:decimal |
| A mod B | numeric | numeric | numeric |

## Syntax

**arithmetic expression**

▶▶──*multiplicative-expression*────────────────────────────────────────▶◀
　　　　　　　　　　　├─+──*multiplicative-expression*─┤
　　　　　　　　　　　└─-──*multiplicative-expression*─┘

**multiplicative expression**
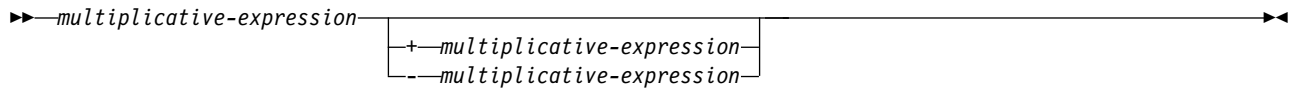
▶▶──┬──────┬──*path_expression*──┬──────────────────────────────────┬──▶◀
　　　├─+─┤　　　　　　　　　　　　　├─*─┬──────┬──*path_expression*─┤
　　　└─-─┘　　　　　　　　　　　　　├─div─┤├─+─┤
　　　　　　　　　　　　　　　　　　　├─idiv─┤└─-─┘
　　　　　　　　　　　　　　　　　　　└─mod─┘
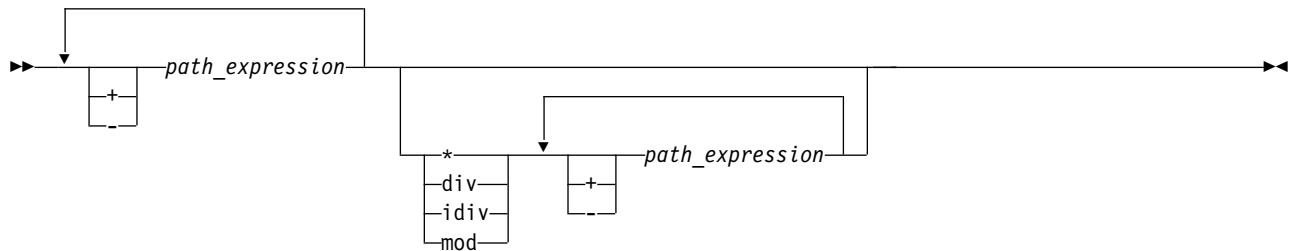
## Examples

The following query uses an arithmetic expression to calculate the amount that buyers pay in taxes on a product, at a rate of 8.25%, and selects the description elements for which the tax is greater than one unit of currency.

```
SELECT X.* FROM T1, XMLTABLE('declare namespace pos="http://posample.org";
                        /pos:product/description[price * .0825 > 1]'
                        PASSING T1.DESCRIPTION) X;
```

The following query subtracts two xs:date values, which results in the xs:yearMonthDuration value P8559D:

```
SELECT * FROM XMLTABLE('xs:date("2005-10-10") - xs:date("1982-05-05")') X;
```

## Comparison expressions

Comparison expressions compare two values. DB2 XPath provides one kind of comparison expression: general comparisons.

**General comparisons:**

General comparisons compare two sequences of any length to determine if a comparison is true for at least one item in both sequences. The general comparison operators include =, !=, <, <=, >, and >=.

The following table describes these operators, listed in order of operator precedence from highest to lowest.

*Table 59. General comparison operators in XPath*

| Operator | Purpose |
| --- | --- |
| = | Returns true if any value in the first sequence is equal to any value in the second sequence. |
| != | Returns true if any value in the first sequence is not equal to any value in the second sequence. |
| < | Returns true if any value in the first sequence is less than any value in the second sequence. |
| <= | Returns true if any value in the first sequence is less than or equal to any value in the second sequence. |

*Table 59. General comparison operators in XPath  (continued)*

| Operator | Purpose |
|---|---|
| > | Returns true if any value in the first sequence is greater than any value in the second sequence. |
| >= | Returns true if any value in the first sequence is greater than or equal to any value in the second sequence. |

The result of a general comparison expression is a boolean value or an error. DB2 XPath uses the following process to evaluate a general comparison expression.

1. Atomizes each operand into a sequence of atomic values.

2. Compares each of the values in the first sequence to each of the values in the second sequence. For string comparisons, the default collation is used. For each comparison:

   - If one of the atomic values is an instance of `xs:untypedAtomic` and the other is an instance of a numeric type (xs:integer, xs:decimal, or xs:double), the untyped value is cast to the type `xs:double`.

   - If one of the atomic values is an instance of `xs:untypedAtomic` and the other is an instance of `xs:untypedAtomic` or `xs:string`, the `xs:untypedAtomic` values are cast to the type `xs:string`.

   - If one of the atomic values is an instance of `xs:untypedAtomic` and the other is not an instance of `xs:string`, `xs:untypedAtomic`, or any numeric type, the `xs:untypedAtomic` value is cast to the dynamic type of the other value.

3. If at least one of the values in the first sequence and at least one of the values in the second sequence meet the conditions of the comparison, the general comparison is true.

**Syntax**

**comparison expression**

```
►►──arithmetic-expression─┬──────────────────────────────────┬──►◄
                          │  ┌─=──┬──arithmetic-expression─┐  │
                          └──┼─!=─┤                        ──┘
                             ├─<──┤
                             ├─>──┤
                             ├─<=─┤
                             └─>=─┘
```
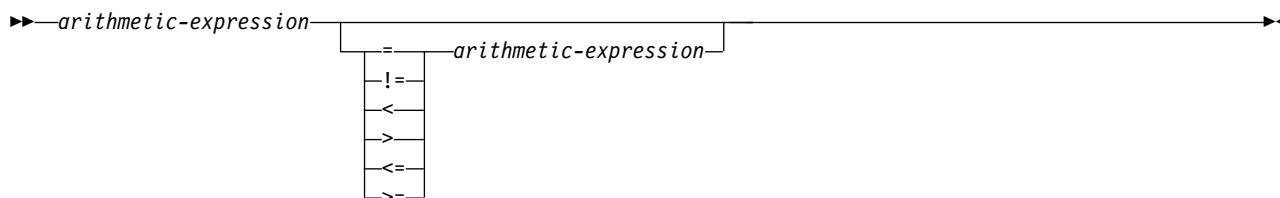
**Examples**

The following statement uses a general comparison expression to find the descriptions of products that cost less than 20 units.

```
declare namespace pos="http://posample.org";
  /pos:product/description[price < 20]
```

## Logical expressions

Logical expressions return the boolean value true if both of two expressions are true, or if one or both of two expressions are true. The operators that are used in logical expressions include `and` and `or`.

The following table describes these operators, listed in order of operator precedence from highest to lowest.

*Table 60. Logical expression operators in XPath*

| Operator | Purpose |
|---|---|
| and | Returns true if both expressions are true. |

*Table 60. Logical expression operators in XPath (continued)*

| Operator | Purpose |
|---|---|
| or | Returns true if one or both expressions are true. |

The result of a logical expression is a boolean value or an error. DB2 XPath uses the following process to evaluate a logical expression.

1. Determines the effective boolean value (EBV) of each operand.
2. Applies the operator to the effective boolean values of the operands. The result is a boolean value or an error. Table 61 shows the results that are returned by a logical expression based on the EBV of its operands and any errors that are encountered during the evaluation of an operand.

*Table 61. Results of logical expressions based on effective boolean values (EBVs) of operands*

| EBV of operand 1 | Operator | EBV of operand 2 | Result |
|---|---|---|---|
| true | and | true | true |
| true | and | false | false |
| false | and | true | false |
| false | and | false | false |
| true | and | error | error |
| error | and | true | error |
| false | and | error | false or error |
| error | and | false | false or error |
| error | and | error | error |
| true | or | true | true |
| false | or | false | false |
| true | or | false | true |
| false | or | true | true |
| true | or | error | true or error |
| error | or | true | true or error |
| false | or | error | error |
| error | or | false | error |
| error | or | error | error |

## Syntax

**logical expression**
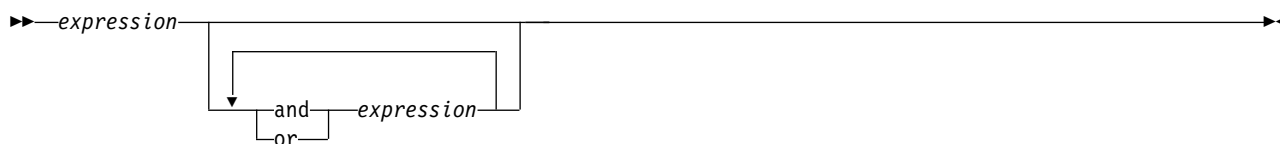


## Examples

The following example uses a logical expression to retrieve records for 22-inch snow shovels or 24-inch snow shovels.

```
declare namespace pos="http://posample.org";
  /pos:product/description[name = "Snow Shovel, Deluxe 24"
                            or name = "Snow Shovel, Basic 22"]
```

## Regular expressions

A regular expression is a sequence of characters that act as a pattern for matching and manipulating strings. Regular expressions are used in the fn:matches, fn:replace, and fn:tokenize functions.

### Syntax
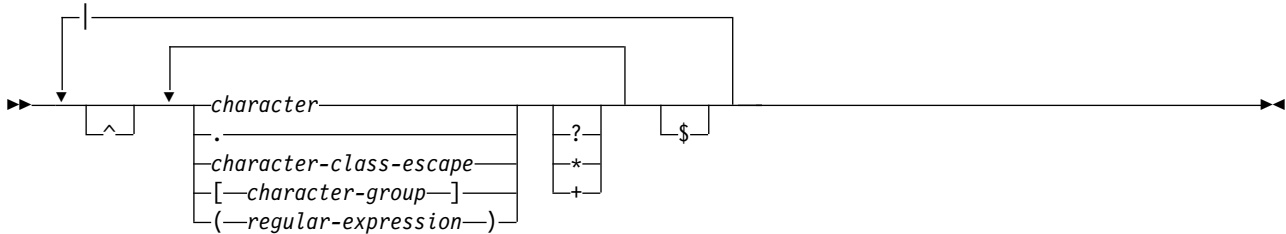


### character-group
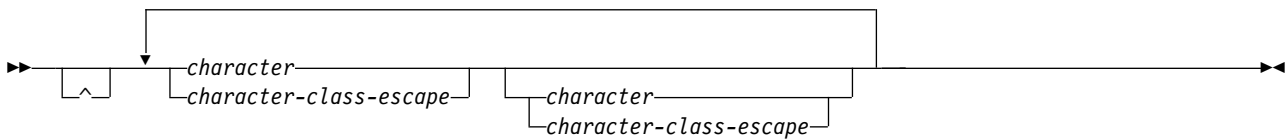


**character**
> In a regular expression, *character* is a normal XML character that is not a metacharacter.

**metacharacters**
> Metacharacters are control characters in regular expressions. The regular expression metacharacters that are currently supported are:

> **backslash (\)**
>> Begins a character class escape. A character class escape indicates that the metacharacter that follows is to be used as a character, instead of a metacharacter.

> **period (.)**
>> Matches any single character except a newline character (\n).

> **carat (^)**
>> If the carat character appears outside of a character class, the characters that follow the carat match the start of the input string or, for multi-line input strings, the start of a line. An input string is considered to be a multi-line input string if the function that uses the input string includes the m flag.

>> If the carat character appears as the first character within a character class, the carat acts as a not-sign. A match occurs if none of the characters in the character group appear in the string that is being compared to the regular expression.

> **dollar sign ($)**
>> Matches the end of the input string or, for multi-line input strings, the end of a line. An input string is considered to be a multi-line input string if the function that uses the input string includes the m flag.

> **question mark (?)**
>> Matches the preceding character or character group in the regular expression zero or one time.

**asterisk (*)**
　　Matches the preceding character or character group in the regular expression zero or more times.

**plus sign (+)**
　　Matches the preceding character or character group in the regular expression one or more times.

**pipe (|)**
　　Matches the preceding character (or character group) or the following character (or character group).

**opening bracket ([) and closing bracket (])**
　　The opening and closing brackets and the enclosed character group define a character class. For example, the character class [aeiou] matches any single vowel. Character classes also support character ranges. For example:

　　- [a-z] means any lowercase letter.
　　- [a-p] means any lowercase letter from a through p.
　　- [0-9] means any single digit.

**opening parenthesis (() and closing parenthesis ())**
　　An opening and closing parenthesis denote a grouping of some characters within a regular expression. You can then apply an operator, such as a repetition operator, to the entire group.

The left curly brace ({) and right curly brace (}) are also metacharacters, but they are currently not supported.

**character-class-escape**
　　A character class escape specifies that you want certain special characters to be treated as characters, instead of performing some function. A character class escape consists of a backslash (\), followed by a single metacharacter, newline character, return character, or tab character. The following table lists the character class escapes.

*Table 62. Single-character character class escapes*

| Character escape | Character represented | Description |
|---|---|---|
| \n | #x0A | Newline |
| \r | #x0D | Return |
| \t | #x09 | Tab |
| \\ | \ | Backslash |
| \| | | | Pipe |
| \. | . | Period |
| \? | ? | Question mark |
| \* | * | Asterisk |
| \+ | + | Plus sign |
| \( | ( | Opening parenthesis |
| \) | ) | Closing parenthesis |
| \{ | { | Opening curly brace |
| \} | } | Closing curly brace |
| \$ | $ | Dollar sign |
| \- | - | Dash |
| \[ | [ | Opening bracket |
| \] | ] | Closing bracket |
| \^ | ^ | Caret |

**character-group**

A character group is the set of characters in a character class. The character class is used for matching. It can consist of characters, character ranges, character class escapes, and an optional opening carat. If the carat is included, it indicates the complement of the set of characters that are defined by the rest of character group.

## Examples

The following examples demonstrate how each of the metacharacters affects a regular expression.

- "hello[0-9]world" matches "hello3world", but not "hello world".
- "^hello" matches this text:

  hello world

  However, "^hello" does not match this text:

  world hello
- "hello$" matches this text:

  world hello

  However, "hello$" does not match this text:

  hello world
- "(ca)|(bd)" matches "arcade" or "abdicate".
- "^((ca)|(bd))" does not match "arcade" or "abdicate".
- "w?s" matches "ws" or "s".
- "w.*s" matches "was" or "waters".
- "be+t" matches "beet" or "bet".
- "\[n\]" matches "[n]".

# Descriptions of XPath functions

The DB2 XPath functions are a subset of the XPath 2.0 and XQuery 1.0 functions and operators.

These topics provide detailed reference information for the XPath functions that are supported by DB2 for i. Functions can be used anywhere in an XPath expression where an expression is expected. The supported functions are listed in the following tables:

*Table 63. String functions*

| Function | Description | Reference |
|---|---|---|
| fn:compare | Returns an indication of whether two strings compare equal, less than, or greater than. | "fn:compare function" on page 185 |
| fn:concat | Returns two or more strings concatenated into a single string. | "fn:concat function" on page 186 |
| fn:contains | Returns an indication of whether a string contains a given substring. | "fn:contains function" on page 187 |
| fn:lower-case | Returns a string converted to lower case. | "fn:lower-case function" on page 198 |
| fn:matches | Returns an indication of whether a string matches a given pattern. | "fn:matches function" on page 199 |
| fn:max | Returns the maximum of the values in a sequence. | "fn:max function" on page 200 |
| fn:min | Returns the minimum of the values in a sequence. | "fn:min function" on page 201 |
| fn:normalize-space | Returns a string with leading and trailing whitespace removed and each internal sequence of whitespace characters replaced by a single blank character. | "fn:normalize-space function" on page 207 |

*Table 63. String functions (continued)*

| Function | Description | Reference |
|---|---|---|
| fn:replace | Returns a string that has characters that match a pattern replaced with another set of characters. | "fn:replace function" on page 209 |
| fn:starts-with | Returns whether a string begins with a given substring. | "fn:starts-with function" on page 213 |
| fn:string | Returns the string representation of a value. | "fn:string function" on page 213 |
| fn:string-length | Returns the length of a string. | "fn:string-length function" on page 214 |
| fn:substring | Returns a substring of a string. | "fn:substring function" on page 214 |
| fn:tokenize | Returns a list of substrings within a string. | "fn:tokenize function" on page 217 |
| fn:translate | Returns a string with selected characters replaced with other characters. | "fn:translate function" on page 219 |
| fn:upper-case | Returns a string converted to uppercase | "fn:upper-case function" on page 219 |

*Table 64. Number functions*

| Function | Description | Reference |
|---|---|---|
| fn:abs | Returns the absolute value of a numeric value. | "fn:abs function" on page 179 |
| fn:max | Returns the maximum of the values in a sequence. | "fn:max function" on page 200 |
| fn:min | Returns the minimum of the values in a sequence. | "fn:min function" on page 201 |
| fn:round | Returns the integer that is closest to the specified value. | "fn:round function" on page 210 |
| fn:sum | Returns the sum of the values in a sequence. | "fn:sum function" on page 215 |

*Table 65. Boolean functions*

| Function | Description | Reference |
|---|---|---|
| fn:boolean | Returns the effective boolean value of a sequence. | "fn:boolean function" on page 185 |
| fn:exists | Returns false if the argument produces an empty result; otherwise returns true. | "fn:exists function" on page 193 |
| fn:not | Returns false if the effective boolean value of a sequence expression is true and returns true is the effective boolean value of a sequence expression is false. | "fn:not function" on page 207 |

*Table 66. Date, time, and duration functions*

| Function | Description | Reference |
|---|---|---|
| fn:adjust-date-to-timezone | Returns an xs:date value with its timezone adjusted or removed. | "fn:adjust-date-to-timezone function" on page 180 |
| fn:adjust-dateTime-to-timezone | Returns an xs:dateTime value with its timezone adjusted or removed. | "fn:adjust-dateTime-to-timezone function" on page 182 |
| fn:adjust-time-to-timezone | Returns an xs:time value with its timezone adjusted or removed. | "fn:adjust-time-to-timezone function" on page 183 |

*Table 66. Date, time, and duration functions (continued)*

| Function | Description | Reference |
|---|---|---|
| fn:current-date | Returns the current date in the implicit timezone of UTC. | "fn:current-date function" on page 188 |
| fn:current-dateTime | Returns the current date and time in the implicit timezone of UTC. | "fn:current-dateTime function" on page 188 |
| db2-fn:current-local-date | Returns the current date in the local timezone. | "db2-fn:current-local-date function" on page 188 |
| db2-fn:current-local-dateTime | Returns the current date and time in the local timezone. | "db2-fn:current-local-dateTime function" on page 189 |
| db2-fn:current-local-time | Returns the current time in the local timezone. | "db2-fn:current-local-time function" on page 189 |
| fn:current-time | Returns the current time in the implicit timezone of UTC. | "fn:current-time function" on page 190 |
| fn:dateTime | Returns an xs:dateTime value from an xs:date value and an xs:time value. | "fn:dateTime function" on page 190 |
| fn:day-from-date | Returns the day component of an xs:date value. | "fn:day-from-date function" on page 191 |
| fn:day-from-dateTime | Returns the day component of an xs:dateTime value. | "fn:day-from-dateTime function" on page 191 |
| fn:days-from-duration | Returns the days component of a duration. | "fn:days-from-duration function" on page 192 |
| fn:hours-from-dateTime | Returns the hours component of an xs:dateTime value. | "fn:hours-from-dateTime function" on page 194 |
| fn:hours-from-duration | Returns the hours component of a duration value. | "fn:hours-from-duration function" on page 194 |
| fn:hours-from-time | Returns the hours component of an xs:time value. | "fn:hours-from-time function" on page 195 |
| fn:implicit-timezone | Returns the implicit time zone value of PT0S which indicates that UTC is the implicit time zone. | "fn:implicit-timezone function" on page 196 |
| db2-fn:local-timezone | Returns the time zone of the local system. | "db2-fn:local-timezone function" on page 198 |
| fn:max | Returns the maximum of the values in a sequence. | "fn:max function" on page 200 |
| fn:min | Returns the minimum of the values in a sequence. | "fn:min function" on page 201 |
| fn:minutes-from-dateTime | Returns the minutes component of an xs:dateTime value. | "fn:minutes-from-dateTime function" on page 202 |
| fn:minutes-from-duration | Returns the minutes component of a duration. | "fn:minutes-from-duration function" on page 203 |
| fn:minutes-from-time | Returns the minutes component of an xs:time value. | "fn:minutes-from-time function" on page 203 |
| fn:month-from-date | Returns the months component of an xs:date value. | "fn:month-from-date function" on page 204 |
| fn:month-from-dateTime | Returns the months component of an xs:dateTime value. | "fn:month-from-dateTime function" on page 204 |
| fn:months-from-duration | Returns the months component of a duration. | "fn:months-from-duration function" on page 205 |
| fn:seconds-from-dateTime | Returns the seconds component of an xs:dateTime value. | "fn:seconds-from-dateTime function" on page 211 |

*Table 66. Date, time, and duration functions  (continued)*

| Function | Description | Reference |
|---|---|---|
| fn:seconds-from-duration | Returns the seconds component of a duration. | "fn:seconds-from-duration function" on page 211 |
| fn:seconds-from-time | Returns the seconds component of an xs:time value. | "fn:seconds-from-time function" on page 212 |
| fn:sum | Returns the sum of the values in a sequence. | "fn:sum function" on page 215 |
| fn:timezone-from-date | Returns the timezone component of an xs:date value. | "fn:timezone-from-date function" on page 215 |
| fn:timezone-from-dateTime | Returns the timezone component of an xs:dateTime value. | "fn:timezone-from-dateTime function" on page 216 |
| fn:timezone-from-time | Returns the timezone component of an xs:time value. | "fn:timezone-from-time function" on page 217 |
| fn:year-from-date | Returns the year component of an xs:date value. | "fn:year-from-date function" on page 220 |
| fn:year-from-dateTime | Returns the year component of an xs:dateTime value. | "fn:year-from-dateTime function" on page 220 |
| fn:years-from-duration | Returns the years component of a duration. | "fn:years-from-duration function" on page 221 |

*Table 67. Sequence functions*

| Function | Description | Reference |
|---|---|---|
| fn:count | Returns the number of values in a sequence. | "fn:count function" on page 187 |
| fn:data | Returns a sequence of atomic values from a sequence of items. | "fn:data function" on page 190 |
| fn:distinct-values | Returns the distinct values in a sequence. | "fn:distinct-values function" on page 193 |
| fn:last | Returns the number of values in the sequence of items that is currently being processed. | "fn:last function" on page 196 |
| fn:position | Returns the position of the context item in a sequence that is currently being processed. | "fn:position function" on page 208 |

*Table 68. Node functions*

| Function | Description | Reference |
|---|---|---|
| fn:local-name | Returns the local name property of a node. | "fn:local-name function" on page 197 |
| fn:name | Returns the prefix and local name parts of a node name. | "fn:name function" on page 206 |

## fn:abs function

The fn:abs function returns the absolute value of a numeric value.

### Syntax

```
►►──fn:abs(numeric-value)──────────────────────────────────►◄
```

*numeric-value*

An atomic value or an empty sequence.

If *numeric-value* is an atomic value, it has one of the following types:
- xs:double
- xs:decimal
- xs:integer
- A type that is derived from any of the previously listed types
- xs:untypedAtomic

If *numeric-value* has the xs:untypedAtomic data type, it is converted to an xs:double value.

## Returned value

If *numeric-value* is not the empty sequence, the returned value is the absolute value of *numeric-value*.

If *numeric-value* is the empty sequence, fn:abs returns the empty sequence.

The data type of the returned value depends on the data type of *numeric-value*:
- If *numeric-value* is xs:double, xs:decimal or xs:integer, the value that is returned has the same type as *numeric-value*.
- If *numeric-value* has a data type that is derived from xs:double, xs:decimal or xs:integer, the value that is returned has the direct parent data type of *numeric-value*.
- If *numeric-value* has the xs:untypedAtomic data type, the value that is returned has the xs:double data type.

## Example

The following function returns the absolute value of –10.5.
```
fn:abs(-10.5)
```

The returned value is 10.5.

## fn:adjust-date-to-timezone function

The fn:adjust-date-to-timezone function adjusts an xs:date value to a specific time zone, or removes the time zone component from the value.

## Syntax

```
►►──fn:adjust-date-to-timezone(date-value─┬──────────────────┬─)──────────────────────►◄
                                          └─,timezone-value─┘
```

*date-value*

The date value that is to be adjusted.

*date-value* is of type xs:date, or is an empty sequence.

*timezone-value*

A duration that represents the time zone to which *date-value* is to be adjusted.

*timezone-value* can be an empty sequence or a single value of type xs:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

## Returned value

The returned value is either a value of type xs:date or an empty sequence depending on the parameters that are specified. If *date-value* is not an empty sequence, the returned value is of type xs:date. The following table describes the possible returned values:

*Table 69. Types of input values and returned value for fn:adjust-date-to-timezone*

| *date-value* | *timezone-value* | **Returned value** |
|---|---|---|
| *date-value* that contains a time zone component | An explicit value, or no value specified (duration of PT0H) | The *date-value* adjusted for the time zone represented by *timezone-value*. |
| *date-value* that contains a time zone component | An empty sequence | The *date-value* with no time zone component. |
| *date-value* that does not contain a time zone component | An explicit value, or no value specified (duration of PT0H) | The *date-value* with a time zone component. The time zone component is the time zone represented by *timezone-value*. The date component is not adjusted for the time zone. |
| *date-value* that does not contain a time zone component | An empty sequence | The *date-value*. |
| An empty sequence | An explicit value, empty sequence, or no value specified | An empty sequence. |

When adjusting *date-value* to a different time zone, *date-value* is treated as a dateTime value with time component 00:00:00. The returned value contains the time zone component represented by *timezone-value*. The following function calculates the adjusted date value:

```
xs:date(fn:adjust-dateTime-to-timezone(xs:dateTime(date-value),timezone-value))
```

## Examples

In the following examples, the variable $tz is a duration of -10 hours, defined as xs:dayTimeDuration("-PT10H").

The following function adjusts the date value for May 7, 2009 in the UTC+1 time zone. The function specifies a *timezone-value* of -PT10H.

```
fn:adjust-date-to-timezone(xs:date("2009-05-07+01:00"), $tz)
```

The returned date value is 2009-05-06-10:00. The date is adjusted to the UTC-10 time zone.

The following function adds a time zone component to the date value for March 7, 2009 without a time zone component. The function specifies a *timezone-value* of -PT10H.

```
fn:adjust-date-to-timezone(xs:date("2009-03-07"), $tz)
```

The returned value is 2009-03-07-10:00. The time zone component is added to the date value.

The following function adjusts the date value for February 9, 2009 in the UTC-7 time zone. Without a *timezone-value* specified, the function uses the default *timezone-value* PT0H.

```
fn:adjust-date-to-timezone(xs:date("2009-02-09-07:00"))
```

The returned date is 2009-02-09Z, the date is adjusted to UTC.

The following function removes the time zone component from the date value for May 7, 2009 in the UTC-7 time zone. The *timezone-value* is an empty sequence.

```
fn:adjust-date-to-timezone(xs:date("2009-05-07-07:00"), ())
```

The returned value is 2009-05-07.

## fn:adjust-dateTime-to-timezone function

The fn:adjust-dateTime-to-timezone function adjusts an xs:dateTime value to a specific time zone, or removes the time zone component from the value.

### Syntax

```
▶▶──fn:adjust-dateTime-to-timezone(dateTime-value──┬──────────────────┬──)──────────────────────▶◀
                                                    └─,timezone-value──┘
```

*dateTime-value*
> The dateTime value that is to be adjusted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

*timezone-value*
> A duration that represents the time zone to which *dateTime-value* is to be adjusted.
>
> *timezone-value* can be an empty sequence or a single value of type xs:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

### Returned value

The returned value is either a value of type xs:dateTime or is an empty sequence depending on the types of input values. If *dateTime-value* is not an empty sequence, the returned value is of type xs:dateTime. The following table describes the possible returned values:

*Table 70. Types of input values and returned value for fn:adjust-dateTime-to-timezone*

| dateTime-value | timezone-value | Returned value |
|---|---|---|
| *dateTime-value* that contains a time zone component | An explicit value, or no value specified (duration of PT0H) | The *dateTime-value* adjusted to the time zone represented by *timezone-value*. The returned value contains the time zone component represented by *timezone-value*. |
| *dateTime-value* that contains a time zone component | An empty sequence | The *dateTime-value* with no time zone component. |
| *dateTime-value* that does not contain a time zone component | An explicit value, or no value specified (duration of PT0H) | The *dateTime-value* with a time zone component. The time zone component is the time zone represented by *timezone-value*. The date and time components are not adjusted to the time zone. |
| *dateTime-value* that does not contain a time zone component | An empty sequence | The *dateTime-value*. |
| An empty sequence | An explicit value, empty sequence, or no value specified | An empty sequence. |

### Examples

In the following examples, the variable $tz is a duration of -10 hours, defined as `xs:dayTimeDuration("-PT10H")`.

The following function adjusts the dateTime value of March 7, 2009 at 10 a.m. in the UTC-7 time zone to the time zone specified by *time zone-value* of -PT10H.

```
fn:adjust-dateTime-to-timezone(xs:dateTime("2009-03-07T10:00:00-07:00"), $tz)
```

The returned dateTime value is 2009-03-07T07:00:00-10:00.

The following function adjusts the dateTime value for March 7, 2009 at 10 am. The *dateTime-value* does not have a time zone component, and the function specifies a *timezone-value* of -PT10H.

```
fn:adjust-dateTime-to-timezone(xs:dateTime("2009-03-07T10:00:00"), $tz)
```

The returned dateTime is 2009-03-07T10:00:00-10:00.

In the following function adjusts the dateTime value for June 4, 2009 at 10 a.m. in the UTC-7 time zone. Without a *timezone-value* specified, the function uses the default time zone value of PT0H.

```
fn:adjust-dateTime-to-timezone(xs:dateTime("2009-06-04T10:00:00-07:00"))
```

The returned dateTime value is 2009-06-04T17:00:00Z, which is the dateTime value adjusted to UTC.

The following function removes the time zone component from the dateTime value for March 7, 2009 at 10 a.m. in the UTC-7 time zone. The *timezone-value* value is the empty sequence.

```
fn:adjust-dateTime-to-timezone(xs:dateTime("2009-03-07T10:00:00-07:00"), ())
```

The returned dateTime value is 2009-03-07T10:00:00.

## fn:adjust-time-to-timezone function

The fn:adjust-time-to-timezone function adjusts an xs:time value to a specific time zone, or removes the time zone component from the value.

### Syntax

```
►►──fn:adjust-time-to-timezone(time-value──────────────────)──────────────────────►◄
                               └─,timezone-value─┘
```

*time-value*
  The time value that is to be adjusted.

  *time-value* is of type xs:time, or is an empty sequence.

*timezone-value*
  A duration that represents the time zone to which *time-value* is to be adjusted.

  *timezone-value* can be an empty sequence or a single value of type xs:dayTimeDuration between -PT14H and PT14H, inclusive. The value can have an integer number of minutes and must not have a seconds component. If *timezone-value* is not specified, the default value is PT0H, which represents UTC.

### Returned value

The returned value is either a value of type xs:time or an empty sequence depending on the parameters that are specified. If *time-value* is not an empty sequence, the returned value is of type xs:time. The following table describes the possible returned values:

*Table 71. Types of input values and returned value for fn:adjust-time-to-timezone*

| date-value | timezone-value | Returned value |
|---|---|---|
| *time-value* that contains a time zone component | An explicit value, or no value specified (duration of PT0H) | The *time-value* adjusted for the time zone represented by *timezone-value*. The returned value contains the time zone component represented by *timezone-value*. If the time zone adjustment crosses over midnight, the change in date is ignored. |
| *time-value* that contains a time zone component | An empty sequence | The *time-value* with no time zone component. |
| *time-value* that does not contain a time zone component | An explicit value, or no value specified (duration of PT0H) | The *time-value* with a time zone component. The time zone component is the time zone represented by *timezone-value*. The time component is not adjusted for the time zone. |
| *time-value* that does not contain a time zone component | An empty sequence | The *time-value*. |
| An empty sequence | An explicit value, empty sequence, or no value specified | An empty sequence. |

## Examples

In the following examples, the variable $tz is a duration of -10 hours, defined as xs:dayTimeDuration("-PT10H").

The following function adjusts the time value for 10:00 a.m. in the UTC-7 time zone, and the function specifies a *timezone-value* of -PT10H.

```
fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"), $tz)
```

The returned value is 7:00:00-10:00. The time is adjusted to the time zone represented by the duration -PT10H.

The following function adjusts the time value for 1:00 p.m. The time value does not have a time zone component.

```
fn:adjust-time-to-timezone(xs:time("13:00:00"), $tz)
```

The returned value is 13:00:00-10:00. The time contains a time zone component represented by the duration -PT10H.

The following function adjusts the time value for 10:00 a.m. in the UTC-7 time zone. The function does not specify a *timezone-value* and uses the default value of PT0H.

```
fn:adjust-time-to-timezone(xs:time("10:00:00-07:00"))
```

The returned value is 17:00:00Z, the time adjusted to UTC.

The following function removes the time zone component from the time value 8:00 am in the UTC-7 time zone. The *timezone-value* is the empty sequence.

```
fn:adjust-time-to-timezone(xs:time("08:00:00-07:00"), ())
```

The returned value is 8:00:00.

The following example compares two times. The time zone adjustment crosses over the midnight and cause a date change. However, fn:adjust-time-to-timezone ignores date changes.

```
fn:adjust-time-to-timezone(xs:time("01:00:00+14:00"), $tz)
   = xs:time("01:00:00-10:00")
```

The returned value is true.

## fn:boolean function
The fn:boolean function returns the effective boolean value of a sequence.

### Syntax

►►──fn:boolean(*sequence-expression*)──────────────────────────────────────────►◄

*sequence-expression*
>        Any sequence that contains items of any type, or the empty sequence.

### Returned value

The returned effective Boolean value (EBV) depends on the value of *sequence-expression*:

*Table 72. EBVs returned for specific types of values*

| Description of value | EBV returned |
| --- | --- |
| An empty sequence | false |
| A sequence whose first item is a node | true |
| A single value of type xs:boolean (or derived from xs:boolean) | false - if the xs:boolean value is false |
| | true - if the xs:boolean value is true |
| A single value of type xs:string or xs:untypedAtomic (or derived from one of these types) | false - if the length of the value is zero |
| | true - if the length of the value is greater than zero |
| A single value of any numeric type (or derived from any numeric type) | false - if the value is NaN or is numerically equal to zero |
| | true - if the value is not numerically equal to zero |
| All other values | error |

**Note:** The effective Boolean value of a sequence that contains at least one node and at least one atomic value is nondeterministic in a query where the order is unpredictable.

### Example

**Example with an argument that is a single numeric value:** The following function returns the effective Boolean value of 0:

- `fn:boolean(0)`

The returned value is false.

## fn:compare function
The fn:compare function compares two strings.

### Syntax

►►──fn:compare(*string-1*,*string-2*)──────────────────────────────────────────►◄

*string-1* **and** *string-2*

> The xs:string values that are to be compared. DB2 compares the numeric Unicode UTF-8 code value of each character. The comparison is made according to the default collation.

## Returned value

If *string-1* and *string-2* are not the empty sequence, one of the following xs:integer values is returned:

**-1**     If *string-1* is less than *string-2*.

**0**      If *string-1* is equal to *string-2*.

**1**      If *string-1* is greater than *string-2*.

Two strings are compared by comparing the corresponding bytes of each string. If the strings do not have the same length, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks so that it has the same length as the other string.

*string-1* and *string-2* are equal if they both have length 0 or if all corresponding bytes are equal.

If *string-1* and *string-2* are not equal, their relationship (that is, which has the greater value) is determined by the comparison of the first pair of unequal bytes from the left end of the strings.

If *string-1* is longer than *string-2*, and all bytes of *string-2* are equal to the leading bytes of *string-1*, *string-1* is greater than *string-2*.

If *string-1* or *string-2* is the empty sequence, the empty sequence is returned.

## Example

The following function compares 'ABC' to 'ABD' using the default collation.

```
fn:compare('ABC', 'ABD')
```

'ABC' is less than 'ABD'. The returned value is -1.

## fn:concat function

The fn:concat function concatenates two or more strings into a single string.

### Syntax
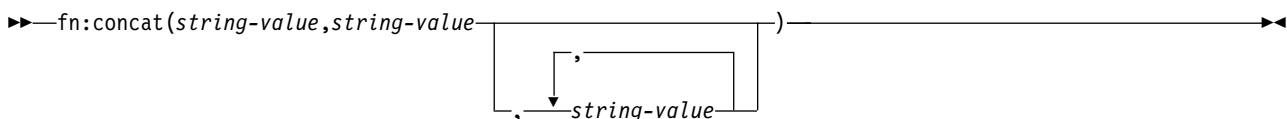


*string-value*

> An xs:string value or the empty sequence.

## Returned value

If all *string-value* arguments are the empty sequence, the returned value is the empty sequence. Otherwise, the returned value is an xs:string value that is the concatenation of all *string-value* arguments that are not the empty sequence.

## Example

The following function concatenates the strings 'ABC', 'ABD', the empty sequence, and 'ABE',

```
fn:concat('ABC', 'ABD', (), 'ABE')
```

The returned value is 'ABCABDABE'.

## fn:contains function

The fn:contains function determines whether a string contains a given substring.

## Syntax

►►──fn:contains(*string*,*substring*)───────────────────────────────────────►◄

*string*    The string to search for *substring*.

> *string* has the xs:string data type, or is the empty sequence. If *string* is the empty sequence, *string* is set to a string of length 0.

*substring*
> The substring to search for in *string*.

> *substring* has the xs:string data type, or is the empty sequence.

## Returned value

The returned value depends on the values of *string* and *substring*:

- If *string* and *substring* are not the empty sequence, the returned value is true if *substring* occurs anywhere within *string*. If *substring* does not occur within *string*, the returned value is false.
- If *string* is the empty sequence, the returned value is true if *substring* is the empty sequence or a string of length 0.
- If *substring* is the empty sequence or a string of length 0, the returned value is true.

## Example

The following function determines whether the string 'Test literal' contains the string 'lite'.

```
fn:contains('Test literal','lite')
```

The returned value is true.

## fn:count function

The fn:count function returns the number of values in a sequence.

## Syntax

►►──fn:count(*sequence-expression*)──────────────────────────────────────────►◄

*sequence-expression*
> A sequence that contains items of any atomic type, or an empty sequence.

## Returned value

If *sequence-expression* is not the empty sequence, an xs:integer value that is the number of values in *sequence-expression* is returned. If *sequence-expression* is the empty sequence, 0 is returned.

## Example

The following function returns 1:
```
fn:count(5)
```

The following function returns the number of employees with a department ID of K55:
```
fn:count(//company/emp[dept/@id="K55"])
```

## fn:current-date function
The fn:current-date function returns the current date in the implicit timezone of UTC.

### Syntax

►►──fn:current-date()──────────────────────────────────►◄

### Returned value

The returned value is an xs:date value that is the current date.

### Example

The following function returns the current date.
```
fn:current-date()
```

If this function were invoked on December 2, 2009, the returned value would be 2009-12-02Z.

## fn:current-dateTime function
The fn:current-dateTime function returns the current date and time in the implicit timezone of UTC.

### Syntax

►►──fn:current-dateTime()──────────────────────────────►◄

### Returned value

The returned value is an xs:dateTime value that is the current date and time.

### Example

The following function returns the current date and time.
```
fn:current-dateTime()
```

If this function were invoked on December 2, 2009 at 6:25 in Toronto (time zone -PT5H), the returned value might be 2009-12-02T011:25:30.864001Z.

## db2-fn:current-local-date function
The db2-fn:current-local-date function returns the current date in the local time zone.

### Syntax

►►──db2-fn:current-local-date()────────────────────────►◄

**Returned value**

The returned value is an xs:date value that is the current date. The returned value does not include a time zone component.

**Example**

The following function returns the current date.

```
db2-fn:current-local-date()
```

If this function were invoked on December 2, 2009 at 3:00 Greenwich Mean Time (GMT) and the local time zone is Eastern Standard Time (-PT5H), the returned value would be 2009-12-01.

## db2-fn:current-local-dateTime function
The db2-fn:current-local-dateTime function returns the current date and time in the local time zone.

**Syntax**

```
►►──db2-fn:current-local-dateTime()─────────────────────────────────────────►◄
```

**Returned value**

The returned value is an xs:dateTime value that is the current date and time. The returned value does not include a time zone component.

**Example**

The following function returns the current date and time.

```
db2-fn:current-local-dateTime()
```

If this function were invoked anywhere on December 2, 2009 at 6:25 local time, the returned value might be `2009-12-02T06:25:30.864001`.

## db2-fn:current-local-time function
The db2-fn:current-local-time function returns the current time in the local time zone.

**Syntax**

```
►►──db2-fn:current-local-time()─────────────────────────────────────────────►◄
```

**Returned value**

The returned value is an xs:time value that is the current time. The returned value does not include a time zone component.

**Example**

The following function returns the current time.

```
db2-fn:current-local-time()
```

If this function were invoked at 6:31 Greenwich Mean Time (GMT) and the local time zone is Eastern Standard Time (-PT5H), the returned value might be `01:31:35.519001`.

## fn:current-time function
The fn:current-time function returns the current time in the implicit timezone of UTC.

### Syntax

►►—fn:current-time()————————————————————————————————————◄◄

### Returned value

The returned value is an xs:time value that is the current time.

### Example

The following function returns the current time.
```
fn:current-time()
```

If this function were invoked at 6:31 Greenwich Mean Time, the returned value might be 06:31:35.519001Z.

## fn:data function
The fn:data function converts a sequence of items to a sequence of atomic values.

### Syntax

►►—fn:data(*sequence*)————————————————————————————————————◄◄

*sequence*
> Any sequence, including the empty sequence.

### Returned values

The returned value is a sequence of items of type xs:anyAtomicType. For each item in the sequence:
- If the item is an atomic value, the returned value is that value.
- If the item is a node, the returned value is the typed value of the node.

### Example

The following function returns the typed values of all qualifying name nodes. Qualifying name nodes are all name nodes that are children of a billTo node in the document.
```
fn:data(//billTo/name)
```

## fn:dateTime function
The fn:dateTime function constructs an xs:dateTime value from an xs:date value and an xs:time value.

### Syntax

►►—fn:dateTime(*date-value*,*time-value*)————————————————————————◄◄

*date-value*
> An xs:date value.

*time-value*
> An xs:time value.

## Returned value

The returned value is an xs:dateTime value with a date component that is equal to *date-value* and a time component that is equal to *time-value*. The time zone of the result is computed as follows:

- If neither argument has a time zone, the result has no time zone.
- If exactly one of the arguments has a time zone, or if both arguments have the same time zone, the result has this time zone.
- If the two arguments have different time zones, an error is returned.

## Example

The following function returns an xs:dateTime value from an xs:date value and an xs:time value.

```
fn:dateTime((xs:date("2009-04-16")), (xs:time("12:30:59")))
```

The returned value is the xs:dateTime value 2009-04-16T12:30:59.

## fn:day-from-date function
The fn:day-from-date function returns the day component of an xs:date value that is in its localized form.

### Syntax

►►──fn:day-from-date(*date-value*)──────────────────────────────►◄

*date-value*
> The date value from which the day component is to be extracted.
>
> *date-value* is of type xs:date, or is an empty sequence.

## Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer, and the value is between 1 and 31, inclusive. The value is the day component of *date-value*.

If *date-value* is an empty sequence, the returned value is an empty sequence.

## Example

The following function returns the day component of the date value for June 1, 2009.

```
fn:day-from-date(xs:date("2009-06-01"))
```

The returned value is 1.

## fn:day-from-dateTime function
The fn:day-from-dateTime function returns the day component of an xs:dateTime value that is in its localized form.

### Syntax

►►──fn:day-from-dateTime(*dateTime-value*)──────────────────────►◄

*dateTime-value*
> The dateTime value from which the day component is to be extracted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

## Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 1 and 31, inclusive. The value is the day component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

## Example

The following function returns the day component of the dateTime value for January 31, 2009 at 8:00 p.m. in the UTC+4 time zone.

```
fn:day-from-dateTime(xs:dateTime("2009-01-31T20:00:00+04:00"))
```

The returned value is 31.

# fn:days-from-duration function

The fn:days-from-duration function returns the days component of a duration.

## Syntax

►►──fn:days-from-duration(*duration-value*)──────────────────────────►◄

*duration-value*
> The duration value from which the days component is to be extracted.
>
> *duration-value* is an empty sequence, or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

## Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:dayTimeDuration or is of type xs:duration, the returned value is of type xs:integer, and is the days component of *duration-value* cast as xs:dayTimeDuration. The returned value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:yearMonthDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The days component of *duration-value* cast as xs:dayTimeDuration is the integer number of days computed as ($S$ idiv 86400). The value $S$ is the total number of seconds of *duration-value* cast as xs:dayTimeDuration to remove the years and months components.

## Examples

This function returns the days component of the duration -10 days and 0 hours.

```
fn:days-from-duration(xs:dayTimeDuration("-P10DT00H"))
```

The returned value is -10.

This function returns the days component of the duration 3 days and 55 hours.

```
fn:days-from-duration(xs:dayTimeDuration("P3DT55H"))
```

The returned value is 5. When calculating the total number of days in the duration, 55 hours is converted to 2 days and 7 hours. The duration is equal to P5D7H which has a days component of 5 days.

## fn:distinct-values function

The fn:distinct-values function returns the distinct values in a sequence.

### Syntax

►►──fn:distinct-values(*sequence-expression*)──────────────────────────────────────►◄

*sequence-expression*
> A sequence of atomic values or the empty sequence. The items in the sequence can have any of the following types:
> - Numeric
> - String
> - Date, time, or duration types

### Returned value

If *sequence-expression* is not the empty sequence, the returned value is a sequence that contains values that are the distinct values in *sequence-expression*. The types of the items in the result sequence match the types in the input sequence. Two items are distinct if they are not equal to each other. XPath uses the following rules to obtain a sequence of distinct values:
- If two values cannot be compared, those values are considered to be distinct.
- Values of type xs:untypedAtomic are compared using the rules for xs:string types.
- The order in which the sequence of values is returned might not be the same as the input order.
- The first value of a set of values that compare equal is returned.
- If *sequence-expression* is the empty sequence, the empty sequence is returned.
- For xs:double values, positive zero is equal to negative zero.
- If *sequence-expression* contains multiple NaN values, a single NaN value is returned.

### Example

The following example returns the distinct values of node b:

```
 <x xmlns="http://posample.org">
     <b>1</b><b>a</b><b>1.0</b><b>A</b><b>1</b>
</x>

declare default element namespace "http://posample.org";
fn:distinct-values($d/x/b)
```

The result is ("1", "a", "1.0", "A").

## fn:exists function

The fn:exists function can check for the existence of many different types of items, such as elements, attributes, text nodes, atomic values (for example, an integer), or XML documents. If the expression specified as its argument produces an empty result (the empty sequence), then fn:exists returns false. If the argument returns anything but the empty sequence, then fn:exists returns true.

### Syntax

►►──fn:exists(*sequence-expression*)──────────────────────────────────────────────►◄

*sequence-expression*
> Any sequence of any type, or the empty sequence.

## Returned value

The returned value is true if *sequence-expression* is not the empty sequence. If *sequence-expression* produces the empty sequence, the returned value is false.

## Examples

**Example 1:** Check whether there is a customer element with a child element of phone. If there is, the fn:exists function returns true:

```
fn:exists($info/customer/phone)
```

**Example 2:** Check whether there is a customer element which has an attribute of Cid. If there is, the fn:exists function returns true:

```
fn:exists($info/customer/@Cid)
```

**Example 3:** Check whether the comment element has a text node. In this example, if the comment element is an empty element it has no text node, so fn:exists returns false. Also, if there is no comment element at all, fn:exists returns false.

```
fn:exists($info/comment/text())
```

**Example 4:** Check whether there is any XML document in the XML column INFO of the CUSTOMER table. If there is, the fn:exists function returns true:

```
fn:exists(db2-fn:xmlcolumn("CUSTOMER.INFO") )
```

## fn:hours-from-dateTime function

The fn:hours-from-dateTime function returns the hours component of an xs:dateTime value that is in its localized form.

## Syntax

▶▶──fn:hours-from-dateTime(*dateTime-value*)────────────────────────────────▶◀

*dateTime-value*
> The dateTime value from which the hours component is to be extracted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

## Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 0 and 23, inclusive. The value is the hours component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

## Example

The following function returns the hours component of the dateTime value for January 31, 2009 at 2:00 p.m. in the UTC-8 time zone.

```
fn:hours-from-dateTime(xs:dateTime("2009-01-31T14:00:00-08:00"))
```

The returned value is 14.

## fn:hours-from-duration function

The fn:hours-from-duration function returns the hours component of a duration value.

## Syntax

```
►►──fn:hours-from-duration(duration-value)────────────────────────────►◄
```

*duration-value*
>   The duration value from which the hours component is to be extracted.
>
>   *duration-value* is an empty sequence or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

## Returned value

The return value depends on the type of *duration-value*:
- If *duration-value* is of type xs:dayTimeDuration or is of type xs:duration, the returned value is of type xs:integer, and is a value between -23 and 23, inclusive. The value is the hours component of *duration-value* cast as xs:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:yearMonthDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The hours component of *duration-value* cast as xs:dayTimeDuration is the integer number of hours computed as $((S \bmod 86400)$ idiv $3600)$. The value $S$ is the total number of seconds of *duration-value* cast as xs:dayTimeDuration to remove the days and months component. The value 86400 is the number of seconds in a day, and 3600 is the number of seconds in an hour.

## Example

The following function returns the hours component of the duration 126 hours.

```
fn:hours-from-duration(xs:dayTimeDuration("PT126H"))
```

The returned value is 6. When calculating the total number of hours in the duration, 126 hours is converted to 5 days and 6 hours. The duration is equal to P5DT6H which has an hours component of 6 hours.

## fn:hours-from-time function
The fn:hours-from-time function returns the hours component of an xs:time value that is in its localized form.

## Syntax

```
►►──fn:hours-from-time(time-value)────────────────────────────────────►◄
```

*time-value*
>   The time value from which the hours component is to be extracted.
>
>   *time-value* is of type xs:time, or is an empty sequence.

## Returned value

If *time-value* is not an empty sequence, the returned value is of type xs:integer, and the value is between 0 and 23, inclusive. The value is the hours component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

## Example

The following function returns the hours component of the time value for 9:30 a.m. in the UTC-8 time zone.

```
fn:hours-from-time(xs:time("09:30:00-08:00"))
```

The returned value is 9.

## fn:implicit-timezone function

The fn:implicit-timezone function returns the time zone that is used when a date, time, or dateTime value that does not have a time zone is used in a comparison or arithmetic operation.

The implicit time zone is the value of PT0S.

### Syntax

►►──fn:implicit-timezone()─────────────────────────────────────────────────────────◄◄

### Returned value

The returned implicit time zone value has type xs:dayTimeDuration.

### Example

The following function returns xs:dayTimeDuration("PT0S"):

```
fn:implicit-timezone()
```

## fn:last function

The fn:last function returns the number of values in the sequence of items that is currently being processed.

### Syntax

►►──fn:last()──────────────────────────────────────────────────────────────────────◄◄

### Returned value

If the sequence that is currently being processed is not the empty sequence, the returned value is an xs:integer value that is the number of values in the sequence. If the sequence that is currently being processed is the empty sequence, the returned value is the empty sequence.

In the following cases, an error is returned:

- fn:last is separated from its context item by "/" or "//".

  For example, the following expressions are not supported:

  ```
  /a/b/c/fn:last
  /a/b/[c/fn:last=3]
  ```

- The context node has a descendant axis or descendant-or-self axis.

  For example, the following expression is not supported:

  ```
  /a/b/descendant::c[fn:last()=1]
  ```

- The context node is a filter expression, and the filter expression has a step with a descendant axis or descendant-or-self axis, or a nested filter expression.

  For example, the following expression is not supported:

```
/a/(b/descendant::c)[fn:last()=1]
```

## Example

In the sample CUSTOMER table, the customer document for customer 1003 looks like this:

```
<customerinfo xmlns="http://posample.org" Cid="1003">
  <name>Robert Shoemaker</name>
  <addr country="Canada">
    <street>1596 Baseline</street>
    <city>Aurora</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X-7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
  <phone type="home">416-555-2937</phone>
  <phone type="cell">905-555-8743</phone>
  <phone type="cottage">613-555-3278</phone>
</customerinfo>
```

The following query retrieves one row with an xml value for the last phone number in the document. The query uses the fn:last function to determine the number of phone number items, and then uses the fn:last result to point to the last phone number.

```
SELECT X.* FROM CUSTOMER, XMLTABLE
    ('declare default element namespace "http://posample.org";
        $D/customerinfo/phone[fn:last()]'
      PASSING CUSTOMER.INFO AS "D") X WHERE CID=1003
```

The returned row contains a single XML column with a value of <phone type="cottage">613-555-3278</phone>.

## fn:local-name function

The fn:local-name function returns the local name property of a node.

## Syntax

```
►►──fn:local-name(──┬───────┬──)────────────────────────────────────►◄
                    └─node─┘
```

*node*    The node for which the local name is to be retrieved. If *node* is not specified, fn:local-name is
          evaluated for the current context node.

## Returned value

The returned value is an xs:string value. The value depends on whether *node* is specified, and the value of *node*:

- If *node* is not specified, the local name of the context node is returned.
- If *node* meets any of the following conditions, a string of length 0 is returned:
  - *node* is the empty sequence.
  - *node* is a document node, a comment, or a text node. These nodes have no name.
- In the following cases, an error is returned:
  - The context node is undefined.
  - The context item is not a node.
  - *node* is a sequence of more than one node.
- Otherwise, an xs:string value is returned that contains the local name part of the expanded name for *node*.

## Examples

The following example returns the local name for node b.

```
SELECT * FROM XMLTABLE('fn:local-name($d/x/b)'
                        PASSING XMLPARSE(DOCUMENT
                                 '<x><b><c></c></b></x>')
                                AS "d"
                        COLUMNS RESULTNAME VARCHAR(100) PATH 'http://posample.org')  X
```

The returned value is "b".

The following example demonstrates that fn:localname() with no argument returns the context node.

In a sample CUSTOMER table, the customer document for customer 1001 looks like this:

```
<customerinfo xmlns="http://posample.org" Cid="1001">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
</customerinfo>
```

The following example returns the local name for the context node.

```
SELECT XV.* FROM CUSTOMER,
      XMLTABLE(XMLNAMESPACES(DEFAULT 'http://posample.org'),
               '$X/customerinfo/*[fn:last()]/fn:local-name()'
               PASSING CUSTOMER.INFO AS "X")
               COLUMNS LOCALNAME CLOB(1K) PATH '.') XV
 WHERE CID=1001
```

The returned value is "phone".

## db2-fn:local-timezone function

The db2-fn:local-timezone function returns the time zone of the local system.

### Syntax

```
►►──db2-fn:local-timezone()──────────────────────────────────────────────────►◄
```

### Returned value

The returned value is an xs:dayTimeDuration value that represents the local time zone offset from Coordinated Universal Time (UTC).

### Example

The following function returns the local time zone.

```
db2-fn:local-timezone()
```

If you invoke this function in the local time zone of Eastern Standard Time, the returned value is -PT5HI.

## fn:lower-case function

The fn:lower-case function converts a string to lowercase.

## Syntax

```
►►──fn:lower-case(source-string)──────────────────────────────────────────────►◄
```

*source-string*
> The string that is to be converted to lowercase.
>
> *source-string* is of type xs:string, or is the empty sequence.

## Returned value

If *source-string* is not the empty sequence, the returned value is an xs:string value that is *source-string*, with each character converted to its lowercase correspondent. Every character that does not have a lowercase correspondent is included in the returned value in its original form.

If *source-string* is the empty sequence, the returned value is a string of length zero.

## Example

The following function converts the string "Wireless Router TB2561" to lowercase:

```
fn:lower-case("Wireless Router TB2561")
```

The returned value is "wireless router tb2561".

## fn:matches function

The fn:matches function determines whether a string matches a given pattern.

## Syntax

```
►►──fn:matches(source-string,pattern──────────)──────────────────────────────────►◄
                                 └─,flags─┘
```

*source-string*
> A string that is compared to a pattern.
>
> *source-string* is a literal string, or an XPath expression that resolves to an xs:string value or the empty sequence.

*pattern*
> A regular expression that is compared to *source-string*. A regular expression is a set of characters, pattern-matching characters, and operators that define a string or group of strings in a search pattern.
>
> *pattern* is string literal.

*flags*  A string literal that can contain any of the following values that control matching of *pattern* to *source-string*:

> **s**  Indicates that the dot (.) matches any character.
>
> > If the s flag is not specified, the dot (.) matches any character except the new line character (#x0A).
>
> **m**  Indicates that the caret (^) matches the start of any line (the position after a new line character), and the dollar sign ($) matches the end of any line (the position before a new line character).
>
> > If the m flag is not specified, the caret (^) matches the start of the entire string, and the dollar sign ($) matches the end of the entire string.

| **i** | Indicates that matching is case-insensitive for the letters "a" through "z" and "A" through "Z". |

If the i flag is not specified, case-sensitive matching is done.

| **x** | Indicates that whitespace characters (#x09, #x0A, #x0D, and #x20) within *pattern* are ignored, unless they are within a character class. Whitespace characters in a character class are never ignored. |

If the x flag is not specified, whitespace characters are used for matching.

## Returned value

If *source-string* is not the empty sequence, the returned value is an xs:boolean value that is true if *source-string* matches *pattern*. The returned value is false if *source-string* does not match *pattern*.

The rules for matching are:
- If *pattern* does not contain the string-starting or line-starting character caret (^), or the string-ending or line-ending character dollar sign ($), *source-string* matches *pattern* if any substring of *source-string* matches *pattern*.
- If *pattern* contains the string-starting or line-starting character caret (^), *source-string* matches *pattern* only if *source-string* matches *pattern* from the beginning of *source-string* or the beginning of a line in *source-string*.
- If *pattern* contains the string-ending or line-ending character dollar sign ($), *source-string* matches *pattern* only if *source-string* matches *pattern* at the end of *source-string* or at the end of a line of *source-string*.
- The m flag determines:
  - Whether a match occurs from the beginning of the string or the beginning of a line
  - Whether a match occurs from the end of the string or the end of a line.

If *source-string* is the empty sequence, *source-string* is considered to be a string of length 0, and *source-string* matches *pattern* if *pattern* matches a string of length 0.

## Examples

**Example of matching a pattern to any substring within a string:** The following function determines whether the strings "ac" or "bd" appear anywhere within the string "abbcacadbdcd".

```
fn:matches("abbcacadbdcd","(ac)|(bd)")
```

The returned value is true.

**Example of matching a pattern to an entire string:** The following function determines whether the strings "ac" or "bd" match the string "bd". The caret (^) character and the dollar sign ($) character indicate that the match must start at the beginning of the source string and end at the end of the source string.

```
fn:matches("bd","^(ac)|(bd)$")
```

The returned value is true.

## fn:max function
The fn:max function returns the maximum of the values in a sequence.

## Syntax

```
►►──fn:max(sequence-expression)──────────────────────────────────►◄
```

*sequence-expression*
> The empty sequence, or a sequence in which all of the items are one of the following types:
> - Numeric
> - String
> - xs:date
> - xs:dateTime
> - xs:time
> - xs:dayTimeDuration
> - xs:yearMonthDuration
>
> Input items of type xs:untypedAtomic are cast to xs:double. Numeric input items are converted to the least common type that can be compared by a combination of type promotion and subtype substitution.

## Returned value

If *sequence-expression* is not the empty sequence, the returned value is a value of type xs:anyAtomicType that is the maximum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the common data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* contains one item, that item is returned. If *sequence-expression* is the empty sequence, the empty sequence is returned. If the sequence includes the value NaN, NaN is returned.

## Example

The following query returns a single row that contains a value of type double that is the maximum of the sequence (500, 1.0E2, 40.5).

```
SELECT * FROM
 XMLTABLE (XMLNAMESPACES (DEFAULT 'http://posample.org'),
          '$d'
          PASSING XMLPARSE(DOCUMENT '<x xmlns="http://posample.org">
                             <b>500</b><b>1.0E2</b><b>40.5</b></x>') AS "d"
          COLUMNS RES DOUBLE PATH 'fn:max(x/b)') X
```

The values are promoted to the xs:double data type. The function returns the xs:double value 5.0E2, which is then converted to the SQL double data type.

## fn:min function
The fn:min function returns the minimum of the values in a sequence.

## Syntax

```
►►──fn:min(sequence-expression)──────────────────────────────────────►◄
```

*sequence-expression*
> The empty sequence, or a sequence in which all of the items are one of the following types:
> - Numeric
> - String
> - xs:date
> - xs:dateTime
> - xs:time
> - xs:dayTimeDuration

- xs:yearMonthDuration

Input items of type xs:untypedAtomic are cast to xs:double. Numeric input items are converted to the least common type that can be compared by a combination of type promotion and subtype substitution.

**Returned value**

If *sequence-expression* is not the empty sequence, the returned value is a value of type xs:anyAtomicType that is the minimum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the common data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* contains one item, that item is returned. If *sequence-expression* is the empty sequence, the empty sequence is returned. If the sequence includes the value NaN, NaN is returned.

**Example**

The following query returns the minimum of the sequence (500, 1.0E2, 40.5).

```
SELECT * FROM
 XMLTABLE (XMLNAMESPACES (DEFAULT 'http://posample.org'),
           '$d'
           PASSING XMLPARSE(DOCUMENT '<x xmlns="http://posample.org">
                             <b>500</b><b>1.0E2</b><b>40.5</b></x>') AS "d"
           COLUMNS RES DOUBLE PATH 'fn:min(x/b)') X
```

The values are promoted to the xs:double data type. The function returns the xs:double value 4.05E1, which is then converted to the SQL double data type.

## fn:minutes-from-dateTime function

The fn:minutes-from-dateTime function returns the minutes component of an xs:dateTime value that is in its localized form.

**Syntax**

▶▶──fn:minutes-from-dateTime(*dateTime-value*)──────────────────────────────────────────▶◀

*dateTime-value*
> The dateTime value from which the minutes component is to be extracted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

**Returned value**

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 0 and 59, inclusive. The value is the minutes component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

**Example**

The following function returns the minutes component from the dateTime value for January 23, 2009 at 9:42 a.m. in the UTC-8 time zone.

```
fn:minutes-from-dateTime(xs:dateTime("2009-01-23T09:42:00-08:00"))
```

The returned value is 42.

## fn:minutes-from-duration function

The fn:minutes-from-duration function returns the minutes component of a duration.

### Syntax

►►──fn:minutes-from-duration(*duration-value*)──────────────────────────────────◄◄

*duration-value*
>    The duration value from which the minutes component is to be extracted.
>
>    *duration-value* is an empty sequence, or is a value that has one of the following types:
>    xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

### Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:dayTimeDuration or is of type xs:duration, the returned value is of type
  xs:integer and is a value between -59 and 59, inclusive. The value is the minutes component of
  *duration-value* cast as xs:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:yearMonthDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The minutes component of *duration-value* cast as xs:dayTimeDuration is the integer number of minutes
computed as ((S mod 3600) idiv 60). The value $S$ is the total number of seconds of *duration-value* cast as
xs:dayTimeDuration to remove the years and months components.

### Example

The following function returns the minutes component of the duration 2 days, 16 hours, and 93 minutes.

```
fn:minutes-from-duration(xs:dayTimeDuration("P2DT16H93M"))
```

The returned value is 33. When calculating the total number of minutes in the duration, 93 minutes is
converted to 1 hour and 33 minutes. The duration is equal to P2DT17H33M which has a minutes
component of 33 minutes.

## fn:minutes-from-time function

The fn:minutes-from-time function returns the minutes component of an xs:time value that is in its
localized form.

### Syntax

►►──fn:minutes-from-time(*time-value*)──────────────────────────────────────◄◄

*time-value*
>    The time value from which the minutes component is to be extracted.
>
>    *time-value* is of type xs:time, or is an empty sequence.

### Returned value

If *time-value* is of type xs:time, the returned value is of type xs:integer, and the value is between 0 and 59,
inclusive. The value is the minutes component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

## Example

The following function returns the minutes component of the time value for 8:59 a.m. in the UTC-8 time zone.

```
fn:minutes-from-time(xs:time("08:59:00-08:00"))
```

The returned value is 59.

## fn:month-from-date function

The fn:month-from-date function returns the month component of a xs:date value that is in its localized form.

### Syntax

►►──fn:month-from-date(*date-value*)──────────────────────────────────────────────────►◄

*date-value*
      The date value from which the month component is to be extracted.

      *date-value* is of type xs:date, or is an empty sequence.

### Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer, and the value is between 1 and 12, inclusive. The value is the month component of *date-value*.

If *date-value* is an empty sequence, the returned value is an empty sequence.

### Example

The following function returns the month component of the date value for December 1, 2009.

```
fn:month-from-date(xs:date("2009-12-01"))
```

The returned value is 12.

## fn:month-from-dateTime function

The fn:month-from-dateTime function returns the month component of an xs:dateTime value that is in its localized form.

### Syntax

►►──fn:month-from-dateTime(*dateTime-value*)──────────────────────────────────────────►◄

*dateTime-value*
      The dateTime value from which the month component is to be extracted.

      *dateTime-value* is of type xs:dateTime, or is an empty sequence.

### Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer, and the value is between 1 and 12, inclusive. The value is the month component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

## Example

The following function returns the month component of the dateTime value for October 31, 2009 at 8:15 a.m. in the UTC-8 time zone.

```
fn:month-from-dateTime(xs:dateTime("2009-10-31T08:15:00-08:00"))
```

The returned value is 10.

## fn:months-from-duration function

The fn:months-from-duration function returns the months component of a duration value.

### Syntax

►►──fn:months-from-duration(*duration-value*)────────────────────────────────►◄

*duration-value*

> The duration value from which the months component is to be extracted.

> *duration-value* is an empty sequence, or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

### Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:duration or is of type xs:yearMonthDuration, the returned value is of type xs:integer, and is a value is between -11 and 11, inclusive. The value is the months component of *duration-value* cast as xs:yearMonthDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:dayTimeDuration, the returned value is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The months component of *duration-value* cast as xs:yearMonthDuration is the integer number of months remaining from the total number of months of *duration-value* divided by 12.

### Examples

The following function returns the months component of the duration 20 years and 5 months.

```
fn:months-from-duration(xs:duration("P20Y5M"))
```

The returned value is 5.

The following function returns the months component of the yearMonthDuration -9 years and -13 months.

```
fn:months-from-duration(xs:yearMonthDuration("-P9Y13M"))
```

The returned value is -1. When calculating the total number of months in the duration, -13 months is converted to -1 year and -1 month. The duration is equal to -P10Y1M which has a month component of -1 month.

The following function returns the months component of the duration 14 years, 11 months, 40 days, and 13 hours.

```
xquery fn:months-from-duration(xs:duration("P14Y11M40DT13H"))
```

The returned value is 11.

## fn:name function

The fn:name function returns the prefix and local name parts of a node name.

### Syntax

```
►►─fn:name(────────)──────────────────────────────────────────────────────►◄
              └node┘
```

*node*  The qualified name of a node for which the name is to be retrieved. If *node* is not specified, fn:name is evaluated for the current context node.

### Returned value

The returned value is an xs:string value. The value depends on the value of *node*:
- If *node* meets any of the following conditions, a string of length 0 is returned:
  - *node* is the empty sequence.
  - *node* is a document node, a comment, or a text node. These nodes have no name.
- In the following cases, an error is returned:
  - The context node is undefined.
  - The context item is not a node.
  - *node* is a sequence of more than one node.
- Otherwise, an xs:string value is returned that contains the prefix (if present) and local name for *node*.

### Example

The following example returns one row containing a CLOB column containing the qualified name for each b node.

```
SELECT * FROM
 XMLTABLE (  'declare namespace ns1="http://posample.org";
      $d/x/ns1:b/fn:name(.)'
      PASSING XMLPARSE(DOCUMENT
                    '<x xmlns:n="http://posample.org">
                      <n:b><n:c></n:c></n:b></x>')
             AS "d")
      COLUMNS COL1 CLOB(1K) PATH(.) ) X
```

The returned value is "n:b".

The following example demonstrates that fn:name() with no argument returns the context node.

In the sample CUSTOMER table, the customer document for customer 1001 looks like this:

```
<customerinfo xmlns="http://posample.org" Cid="1001">
  <name>Kathy Smith</name>
  <addr country="Canada">
    <street>25 EastCreek</street>
    <city>Markham</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9C 3T6</pcode-zip>
  </addr>
  <phone type="work">905-555-7258</phone>
</customerinfo>
```

The following example returns the qualified name for the context node.

```
SELECT * FROM CUSTOMER,
 XMLTABLE('declare default element namespace "http://posample.org";
      $X/customerinfo/phone/fn:name()'
      PASSING INFO AS "X")
      COLUMNS RESULT1 CLOB(1K) PATH '.') X
 WHERE CID=1001
```

The returned value is "phone".

## fn:normalize-space function

The fn:normalize-space function strips leading and trailing whitespace characters from a string and replaces multiple consecutive whitespace characters in the string with a single blank character.

### Syntax

►►──fn:normalize-space(─┬──────────────┬─)────────────────────────────────────────►◄
                       └─*source-string*─┘

*source-string*

> A string in which whitespace is to be normalized.

> *source-string* is an xs:string value or the empty sequence.

> If *source-string* is not specified, the argument of fn:normalize-space is the current context item, which is converted to an xs:string value by using the fn:string function.

### Returned value

The returned value is the xs:string value that results when the following operations are performed on *source-string*:

- Leading and trailing whitespace characters are removed.
- Each internal sequence of one or more adjacent whitespace characters is replaced by a single space (U+0020) character.

Whitespace characters are the space character, (U+0020), carriage return, (U+000D), line feed, (U+000A), and tab (U+0009).

If *source-string* is the empty sequence, a string of length 0 is returned.

### Example

The following function removes extra whitespace characters from the string "a b c d ".

```
fn:normalize-space(" a b c d " )
```

The returned value is "a b c d".

## fn:not function

The fn:not function returns false if the effective boolean value of a sequence expression is true. fn:not returns true if the effective boolean value of a sequence expression is false.

### Syntax

►►──fn:not(*sequence-expression*)─────────────────────────────────────────────────►◄

*sequence-expression*

> Any sequence that contains items of any type, or the empty sequence.

## Returned value

The returned value is an xs:boolean value. If the effective boolean value of *sequence-expression* is false, this function returns true. If the effective boolean value of *sequence-expression* is true, this function returns false.

## Example

The following function returns true:

```
fn:not("a"="b")
```

The following function returns false:

```
fn:not("false")
```

## fn:position function

The fn:position function returns the position of the context item in the sequence that is currently being processed.

The position function is typically used in a predicate. However it can also be used to produce the position of each occurrence of its context item.

## Syntax

```
►►──fn:position()──────────────────────────────────────────────────────►◄
```

## Returned value

The returned value is an xs:integer value that indicates the position of the context item in the sequence that is currently being processed. The first item in the sequence has position 1. If the context item is undefined, an error is returned. The position function returns a deterministic result only if the sequence that contains the context item has a deterministic order.

In the following cases, an error is returned:
- The context step is the descendant or descendant-or-self axis
- The context is a sequence of atomic values
- fn:position occurs as part of a nested filter expression within a predicate. I
  `$x/a[$y/c/fn:position()] $x/a[(b/c)[2]]`

## Examples

The following query returns one row that contains an XML column with the second element in the sequence of `<c>` elements in the document.

```
<x xmlns="http://posample.org"><b><c>x</c><c>y</c><c>z</c></b></x>

SELECT * FROM
    XMLTABLE(XMLNAMESPACES(DEFAULT 'http://posample.org'),
            '$d/x/b/c[fn:position()=2]'
            PASSING XMLPARSE(DOCUMENT '<x xmlns="http://posample.org">
                                        <b><c>x</c><c>y</c><c>z</c></b></x>') AS "d"
            COLUMNS RESULT XML PATH '.') X
```

The returned value is `<c>y</c>`.

The following query returns the position of each occurrence of `<a><b><c>` as a single XML value.

```
SELECT * FROM
    XMLTABLE('.'
            PASSING XMLPARSE(DOCUMENT
                    '<a><b><c>c1</c></b><b><c>c2</c><c>c3</c></b></a>')
            COLUMNS RESULT_POS XML PATH '/a/b/c/fn:position()') X
```

The returned value is "1 2 3".

## fn:replace function

The fn:replace function compares each set of characters within a string to a given pattern. fn:replace replaces the characters that match the pattern with another set of characters.

### Syntax

►►—fn:replace(*source-string*,*pattern*,*replacement-string*─┬──────┬─)──────────────────────────►◄
                                                            └─,*flags*─┘

*source-string*
> A string that contains characters that are to be replaced.
>
> *source-string* is a literal string, or an Xpath expression that resolves to an xs:string value or the empty sequence.

*pattern*
> A regular expression that is compared to *source-string*. A regular expression is a set of characters, pattern-matching characters, and operators that define a string or group of strings in a search pattern.
>
> *pattern* is string literal.

*replacement-string*
> A string that contains characters that replace characters that match *pattern* in *source-string*.
>
> *replacement-string* is an xs:string value. A literal $ symbol must be written as \$. A literal \ symbol must be written as \\.

*flags*   A string literal that can contain any of the following values that control matching of *pattern* to *source-string*:

    **s**      Indicates that the dot (.) matches any character.

            If the s flag is not specified, the dot (.) matches any character except the new line character (#x0A).

    **m**      Indicates that the caret (^) matches the start of any line (the position after a new line character), and the dollar sign ($) matches the end of any line (the position before a new line character).

            If the m flag is not specified, the caret (^) matches the start of the entire string, and the dollar sign ($) matches the end of the entire string.

    **i**       Indicates that matching is case-insensitive for the letters "a" through "z" and "A" through "Z".

            If the i flag is not specified, case-sensitive matching is done.

    **x**      Indicates that whitespace characters (#x09, #x0A, #x0D, and #x20) within *pattern* are ignored, unless they are within a character class. Whitespace characters in a character class are never ignored.

            If the x flag is not specified, whitespace characters are used for matching.

**Returned value**

If *source-string* is not the empty sequence, the returned value is an xs:string value that results when the following operations are performed on a copy of *source-string*:

- *source-string* is searched for characters that match *pattern*.
    - If two overlapping substrings of *source-string* match *pattern*, only the substring whose first character comes first in *source-string* is considered to match *pattern*.
    - If *pattern* contains two or more alternative sets of characters, and the alternative sets of characters match characters that start at the same position in *source-string*, the first set of characters in *pattern* that matches characters in *source-string* is considered to match *pattern*.
- Each set of characters in *source-string* that matches *pattern* is replaced with *replacement-string*.

If *pattern* is not found in *source-string*, *source-string* is returned.

If *pattern* matches a string of length zero, an error is returned.

If *source-string* is the empty sequence, a string of length 0 is returned.

### Example

The following function replaces all instances of "a" followed by any single character or "b" followed by any single character with "*@".

```
fn:replace("abbcacadbdcd","(a(.))|(b(.))","*@")
```

The returned value is "*@*@*@*@*@cd".

## fn:round function

The fn:round function returns the integer that is closest to the specified numeric value.

### Syntax

►►—fn:round(*numeric-value*)——————————————————————————————►◄

*numeric-value*
> An atomic value or an empty sequence.
>
> If *numeric-value* is an atomic value, it has one of the following types:
> - xs:double
> - xs:decimal
> - xs:integer
> - A type that is derived from any of the previously listed types

### Returned value

If *numeric-value* is not the empty sequence, the returned value is the integer that is closest to *numeric-value*. The data type of the returned value depends on the data type of *numeric-value*:

- If *numeric-value* is xs:double, xs:decimal or xs:integer, the value that is returned has the same type as *numeric-value*.
- If *numeric-value* has a data type that is derived from xs:double, xs:decimal or xs:integer, the value that is returned has the direct parent data type of *numeric-value*.

If *numeric-value* is the empty sequence, the returned value is the empty sequence.

## Examples

**Example with a positive argument:** The following function returns the rounded value of 0.5:

```
fn:round(0.5)
```

The returned value is 1.

**Example with a negative argument:** The following function returns the rounded value of (-1.5):

```
fn:round(-1.5)
```

The returned value is -1.

## fn:seconds-from-dateTime function
The fn:seconds-from-dateTime function returns the seconds component of an xs:dateTime value that is in its localized form.

### Syntax

```
►►──fn:seconds-from-dateTime(dateTime-value)──────────────────────────────────►◄
```

*dateTime-value*
> The dateTime value from which the seconds component is to be extracted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

### Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:decimal, and the value is greater than or equal to 0 and less than 60. The value is the seconds and fractional seconds component of *dateTime-value*.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

### Examples

The following function returns the seconds component of dateTime value for February 8, 2009 at 2:16:23 p.m. in the UTC-8 time zone.

```
fn:seconds-from-dateTime(xs:dateTime("2009-02-08T14:16:23-08:00"))
```

The returned value is 23.

The following function returns the seconds component of dateTime value for June 23, 2009 at 9:16:20.43 a.m. in the UTC time zone.

```
fn:seconds-from-dateTime(xs:dateTime("2009-06-23T09:16:20.43Z"))
```

The returned value is 20.43.

## fn:seconds-from-duration function
The fn:seconds-from-duration function returns the seconds component of a duration.

### Syntax

```
►►──fn:seconds-from-duration(duration-value)──────────────────────────────────►◄
```

*duration-value*
>    The duration value from which the seconds component is to be extracted.

>    *duration-value* is an empty sequence, or is a value that has one of the following types:
>    xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

## Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:dayTimeDuration, or is of type xs:duration, the returned value is of type xs:decimal, and is a value greater than -60 and less than 60. The value is the seconds and fractional seconds component of *duration-value* cast as xs:dayTimeDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:yearMonthDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The seconds and fractional seconds component of *duration-value* cast as xs:dayTimeDuration is computed as ($S$ mod 60). The value $S$ is the total number of seconds and fractional seconds of *duration-value* cast as xs:dayTimeDuration to remove the years and months components.

## Example

The following function returns the seconds component of the duration 150.5 seconds.

```
fn:seconds-from-duration(xs:dayTimeDuration("PT150.5S"))
```

The returned value is 30.5. When calculating the total number of seconds in the duration, 150.5 seconds is converted to 2 minutes and 30.5 seconds. The duration is equal to PT2M30.5S which has a seconds component of 30.5 seconds.

## fn:seconds-from-time function

The fn:seconds-from-time function returns the seconds component of an xs:time value that is in its localized form.

## Syntax

```
►►──fn:seconds-from-time(time-value)────────────────────────────────►◄
```

*time-value*
>    The time value from which the seconds component is to be extracted.

>    *time-value* is of type xs:time, or is an empty sequence.

## Returned value

If *time-value* is of type xs:time, the returned value is of type xs:decimal, and the value is greater than or equal to zero and less than 60. The value is the seconds and fractional seconds component of *time-value*.

If *time-value* is an empty sequence, the returned value is an empty sequence.

## Example

The following function returns the seconds component of the time value for 08:59:59 a.m. in the UTC-8 time zone.

```
fn:seconds-from-time(xs:time("08:59:59-08:00"))
```

The returned value is 59.

## fn:starts-with function

The fn:starts-with function determines whether a string begins with a given substring. The substring is matched using the default collation.

### Syntax

►►─fn:starts-with(*string*,*substring*)────────────────────────────◄◄

*string*   The string in which to search for *substring*.

      *string* has the xs:string data type, or is the empty sequence. If *string* is the empty sequence, *string* is set to a string of length 0.

*substring*
      The substring to search for.

      *substring* has the xs:string data type, or is the empty sequence.

### Returned value

The returned value is the xs:boolean value `true` if either of the following conditions are satisfied:
- *substring* occurs at the beginning of *string*.
- *substring* is an empty sequence or a string of length zero.

Otherwise, the returned value is `false`.

### Example

The following function determines whether the string 'Test literal' begins with the string 'lite'.
```
fn:starts-with('Test literal','lite')
```

The returned value is `false`.

## fn:string function

The fn:string function returns the string representation of a value.

### Syntax

►►─fn:string(─┬────────┬─)──────────────────────────────────◄◄
            └─*value*─┘

*value*   The value that is to be represented as a string.

      *value* is a node or an atomic value, or is the empty sequence.

      If *value* is not specified, fn:string is evaluated for the current context item. If the current context item is undefined, an error is returned.

### Returned value

If *value* is not the empty sequence, an xs:string value is returned:
- If *value* is a node, the returned value is the string value property of the *value* node.
- If *value* is an atomic value, the returned value is the result of casting *value* to the xs:string type.

If *value* is the empty sequence, the result is a string of length 0.

## Example

The following function returns the string representation of 123:

```
fn:string(xs:integer(123))
```

The returned value is '123'.

## fn:string-length function

The fn:string-length function returns the length of a string.

### Syntax

```
►►──fn:string-length(─┬─────────────┬─)────────────────────────────►◄
                      └─source-string─┘
```

*source-string*
> The string for which the length is to be returned.
>
> *source-string* has the xs:string data type, or is an empty sequence.

### Returned value

If *source-string* is not the empty sequence, the returned value is an xs:integer value that is the number of characters in *source-string*.

If *source-string* is the empty sequence, the returned value is the xs:integer value 0.

If *source-string* is not specified, the argument of fn:string-length defaults to the string value of the context item. If the context item is undefined, an error is raised.

### Example

The following function returns the length of the string "Test literal".

```
fn:string-length("Test literal")
```

The returned value is 12.

## fn:substring function

The fn:substring function returns a substring of a string.

### Syntax

```
►►──fn:substring(source-string,start─┬─────────┬─)───────────────────►◄
                                     └─,length─┘
```

*source-string*
> The string from which the substring is retrieved.
>
> *source-string* has the xs:string data type, or is an empty sequence.

*start*  The starting position in *source-string* of the substring. The first position of *source-string* is 1. If *start*<=0, *start* is set to 1.

> *start* has the xs:double data type.

*length*  The length of the substring. The default for *length* is the length of *source-string*. If *start+length-1* is greater than the length of *source-string*, *length* is set to (length of *source-string*)-*start*+1.

*length* has the xs:double data type.

**Returned value**

If *source-string* is not the empty sequence, the returned value is an xs:string value that is the substring of *source-string* that starts at position *start* and is *length* bytes. If *source-string* is the empty sequence, the result is a string of length 0.

**Example**

The following function returns seven characters starting at the sixth character of the string 'Test literal'.

```
fn:substring('Test literal',6,7)
```

The returned value is 'literal'.

## fn:sum function
The fn:sum function returns the sum of the values in a sequence.

**Syntax**

▶▶──fn:sum(*sequence-expression*)────────────────────────────────────────◀◀

*sequence-expression*
> A sequence that contains items of any of the following atomic types, or an empty sequence:
> * xs:double
> * xs:decimal
> * xs:integer
> * A type that is derived from any of the previously listed types
> * xs:dayTimeDuration
> * xs:yearMonthDuration
> * xs:untypedAtomic
>
> All values in the sequence must be of the same type or a derived type of the type, or must be promotable to a single common type. An xs:untypedAtomic value is promoted to the xs:double data type. A derived type is promoted to its direct parent data type.

**Returned value**

If *sequence-expression* is not the empty sequence, the returned value is the sum of the values in *sequence-expression*. The data type of the returned value is the same as the data type of the items in *sequence-expression*, or the data type to which the items in *sequence-expression* are promoted.

If *sequence-expression* is the empty sequence, fn:sum returns 0.0E0.

**Example**

The following function returns the sum of the sequence (500, 1.0E2, 40.5):

```
fn:sum((500, 1.0E2, 40.5))
```

The values are promoted to the xs:double data type. The returned value is 6.405E2.

## fn:timezone-from-date function
The fn:timezone-from-date function returns the time zone component of an xs:date value.

**Syntax**

```
►►──fn:timezone-from-date(date-value)──────────────────────────────────────────────►◄
```

*date-value*
>    The date value from which the time zone component is to be extracted.
>
>    *date-value* is of type xs:date, or is an empty sequence.

**Returned value**

If *date-value* is of type xs:date and has an explicit time zone component, the returned value is of type xs:dayTimeDuration, and the value is between -PT14H hours and PT14H, inclusive. The value is the deviation of the *date-value* time zone component from the UTC time zone.

If *date-value* does not have an explicit time zone component or is an empty sequence, the returned value is an empty sequence.

**Example**

The following function returns the time zone component of the date value for March 13, 2009 in the UTC-8 time zone.

```
fn:timezone-from-date(xs:date("2009-03-13-08:00"))
```

The returned value is -PT8H.

## fn:timezone-from-dateTime function
The fn:timezone-from-dateTime function returns the time zone component of an xs:dateTime value.

**Syntax**

```
►►──fn:timezone-from-dateTime(dateTime-value)──────────────────────────────────────►◄
```

*dateTime-value*
>    The dateTime value from which the time zone component is to be extracted.
>
>    *dateTime-value* is of type xs:dateTime, or is an empty sequence.

**Returned value**

If *dateTime-value* is of type xs:dateTime and has an explicit time zone component, the returned value is of type xs:dayTimeDuration, and the value is between -PT14H and PT14H, inclusive. The value is the deviation of the *dateTime-value* time zone component from the UTC time zone.

If *dateTime-value* does not have an explicit time zone component, or is an empty sequence, the returned value is an empty sequence.

**Examples**

The following function returns the time zone component of the dateTime value for October 30, 2009 at 7:30 a.m. in the UTC-8 time zone.

```
fn:timezone-from-dateTime(xs:dateTime("2009-10-30T07:30:00-08:00"))
```

The returned value is -PT8H.

The following function returns the time zone component of the dateTime value for January 1, 2009 at 2:30 p.m. in the UTC+10:30 time zone.

```
fn:timezone-from-dateTime(xs:dateTime("2009-01-01T14:30:00+10:30"))
```

The returned value is PT10H30M.

## fn:timezone-from-time function

The fn:timezone-from-time function returns the time zone component of an xs:time value.

### Syntax

►►──fn:timezone-from-time(*time-value*)──────────────────────────────────►◄

*time-value*
> The time value from which the time zone component is to be extracted.
>
> *time-value* is of type xs:time, or is an empty sequence.

### Returned value

If *time-value* is of type xs:time and has an explicit time zone component, the returned value is of type xs:dayTimeDuration, and the value is between -PT14H and PT14H, inclusive. The value is the deviation of the *time-value* time zone component from UTC time zone.

If *time-value* does not have an explicit time zone component, or is an empty sequence, the returned value is an empty sequence.

### Examples

The following function returns the time zone component of the time value for 12 noon in the UTC-5 time zone.

```
fn:timezone-from-time(xs:time("12:00:00-05:00"))
```

The returned value is -PT5H.

In the following function, the time value for 1:00 p.m. does not have a time zone component.

```
fn:timezone-from-time(xs:time("13:00:00"))
```

The returned value is the empty sequence.

## fn:tokenize function

The fn:tokenize function breaks a string into a sequence of substrings.

### Syntax

►►──fn:tokenize(──*source-string*──,──*pattern*──┬───────────┬──)──────────►◄
                                                 └─,──*flags*─┘

*source-string*
> A string that is to be broken into a sequence of substrings.
>
> *source-string* is a literal string, or an XPath expression that resolves to an xs:string value or the empty sequence.

*pattern*
> The delimiter between substrings in *source-string*.

*pattern* is a string literal that contains a regular expression. A regular expression is a set of characters, pattern-matching characters, and operators that define a string or group of strings in a search pattern.

*flags*    A string literal that can contain any of the following values that control matching of *pattern* to *source-string*:

> **s**    Indicates that the dot (.) matches any character.
>
> If the s flag is not specified, the dot (.) matches any character except the new line character (#x0A).
>
> **m**    Indicates that the caret (^) matches the start of any line (the position after a new line character), and the dollar sign ($) matches the end of any line (the position before a new line character).
>
> If the m flag is not specified, the caret (^) matches the start of the entire string, and the dollar sign ($) matches the end of the entire string.
>
> **i**    Indicates that matching is case-insensitive for the letters "a" through "z" and "A" through "Z".
>
> If the i flag is not specified, case-sensitive matching is done.
>
> **x**    Indicates that whitespace characters (#x09, #x0A, #x0D, and #x20) within *pattern* are ignored, unless they are within a character class. Whitespace characters in a character class are never ignored.
>
> If the x flag is not specified, whitespace characters are used for matching.

## Returned value

If *source-string* is not the empty sequence or a zero-length string, the returned value is a sequence of xs:string values that results when the following operations are performed on *source-string*:

- *source-string* is searched for characters that match *pattern*.
- If *pattern* contains two or more alternative sets of characters, and the alternative sets of characters match characters that start at the same position in *source-string*, the first set of characters in *pattern* that matches characters in *source-string* is considered to match *pattern*.
- Each set of characters that does not match *pattern* becomes an item in the result sequence.
- If *pattern* matches characters at the beginning of *source-string*, the first item in the returned sequence is a string of length 0.
- If two successive matches for *pattern* are found within *source-string*, a string of length 0 is added to the sequence.
- If *pattern* matches characters at the end of *source-string*, the last item in the returned sequence is a string of length 0.

If *pattern* is not found in *source-string*, *source-string* is returned.

If *pattern* matches a string of length zero, an error is returned.

If *source-string* is the empty sequence, or is a zero-length string, the result is the empty sequence.

## Example

The following function creates a sequence from the string "?A?B?C?D?" by removing the question mark (?) characters and creating a sequence from the remaining characters.

```
fn:tokenize("?A?B?C?D?","\?")
```

The returned value is the sequence ("", "A", "B", "C", "D", "").

## fn:translate function

The fn:translate function replaces selected characters in a string with replacement characters.

### Syntax

►►──fn:translate(*source-string*,*original-string*,*replacement-string*)──────────────────◄◄

*source-string*
   The string in which characters are to be converted.

   *source-string* has the xs:string data type, or is the empty sequence.

*original-string*
   A string that contains the characters that can be converted.

   *original-string* has the xs:string data type.

*replacement-string*
   A string that contains the characters that replace the characters in *original-string*.

   *replacement-string* has the xs:string data type.

   If the length of *replacement-string* is greater than the length of *original-string*, the additional characters in *replacement-string* are ignored.

### Returned value

If *source-string* is not the empty sequence, the returned value is the xs:string value that results when the following operations are performed:

- For each character in *source-string* that appears in *original-string*, replace the character in *source-string* with the character in *replacement-string* that appears at the same position as the character in *original-string*.

  If the length of *original-string* is greater than the length of *replacement-string*, delete each character in *source-string* that appears in *original-string*, but the character position in *original-string* does not have a corresponding position in *replacement-string*.

  If a character appears more than once in *original-string*, the position of the first occurrence of the character in *original-string* determines the character in *replacement-string* that is used.

- For each character in *source-string* that does not appear in *original-string*, leave the character as it is.

If *source-string* is the empty sequence, a string of length 0 is returned.

### Example

The following function replaces the character a with the character A and deletes any - characters from the string "—aaa—".
```
fn:translate("---aaa---","a-","A")
```

The returned value is "AAA".

## fn:upper-case function

The fn:upper-case function converts a string to uppercase.

### Syntax

►►──fn:upper-case(*source-string*)──────────────────────────────────────◄◄

*source-string*
> The string that is to be converted to uppercase.

> *source-string* has the xs:string data type, or is an empty sequence.

## Returned value

If *source-string* is not an empty sequence, the returned value is the xs:string value *source-string*, with each character converted to its uppercase correspondent. Every character that does not have an uppercase correspondent is included in the returned value in its original form.

If *source-string* is the empty sequence, the returned value is a string of length zero.

## Examples

The following function converts the string 'Test literal 1' to uppercase.

```
fn:upper-case("Test literal 1")
```

The returned value is "TEST LITERAL 1".

The argument of the following function resolves to "ıi".

```
fn:upper-case("&#x131;i")
```

The returned value is "II".

# fn:year-from-date function
The fn:year-from-date function returns the year component of an xs:date value that is in its localized form.

## Syntax

►►—fn:year-from-date(*date-value*)————————————————————————————————►◄

*date-value*
> The date value from which the year component is to be extracted.

> *date-value* is of type xs:date, or is an empty sequence.

## Returned value

If *date-value* is of type xs:date, the returned value is of type xs:integer. The value is the year component of the *date-value*, a non-negative value.

If *date-value* is an empty sequence, the returned value is an empty sequence.

## Example

The following function returns the year component of the date value for October 29, 2009.

```
fn:year-from-date(xs:date("2009-10-29"))
```

The returned value is 2009.

# fn:year-from-dateTime function
The fn:year-from-dateTime function returns the year component of an xs:dateTime value that is in its localized form.

## Syntax

►►─── fn:year-from-dateTime(*dateTime-value*) ──────────────────────────────────────◄◄

*dateTime-value*
> The dateTime value from which the year component is to be extracted.
>
> *dateTime-value* is of type xs:dateTime, or is an empty sequence.

### Returned value

If *dateTime-value* is of type xs:dateTime, the returned value is of type xs:integer. The value is the year component of the *dateTime-value*, a non-negative value.

If *dateTime-value* is an empty sequence, the returned value is an empty sequence.

### Example

The following function returns the year component of the dateTime value for October 29, 2009 at 8:00 a.m. in the UTC-8 time zone.

```
fn:year-from-dateTime(xs:dateTime("2009-10-29T08:00:00-08:00"))
```

The returned value is 2009.

## fn:years-from-duration function

The fn:years-from-duration function returns the years component of a duration.

### Syntax

►►─── fn:years-from-duration(*duration-value*) ─────────────────────────────────────◄◄

*duration-value*
> The duration value from which the years component is to be extracted.
>
> *duration-value* is an empty sequence, or is a value that has one of the following types: xs:dayTimeDuration, xs:duration, or xs:yearMonthDuration.

### Returned value

The return value depends on the type of *duration-value*:

- If *duration-value* is of type xs:yearMonthDuration or is of type xs:duration, the returned value is of type xs:integer. The value is the years component of *duration-value* cast as xs:yearMonthDuration. The value is negative if *duration-value* is negative.
- If *duration-value* is of type xs:dayTimeDuration, the returned value is of type xs:integer and is 0.
- If *duration-value* is an empty sequence, the returned value is an empty sequence.

The years component of *duration-value* cast as xs:yearMonthDuration is the integer number of years determined by the total number of months of *duration-value* cast as xs:yearMonthDuration divided by 12.

### Examples

The following function returns the years component of the duration -4 years, -11 months, and -320 days.

```
fn:years-from-duration(xs:duration("-P4Y11M320D"))
```

The returned value is -4.

The following function returns the years component of the duration 9 years and 13 months.

```
fn:years-from-duration(xs:yearMonthDuration("P9Y13M"))
```

The returned value is 10. When calculating the total number of years in the duration, 13 months is converted to 1 year and 1 month. The duration is equal to P10Y1M which has a years component of 10 years.

## Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant 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 or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country 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 which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

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 is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

## Programming interface information

This SQL XML Programming publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

## Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

## 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, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works 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.

# Index

# Y

**IBM** ®

Product Number: 5770-SS1

Printed in USA