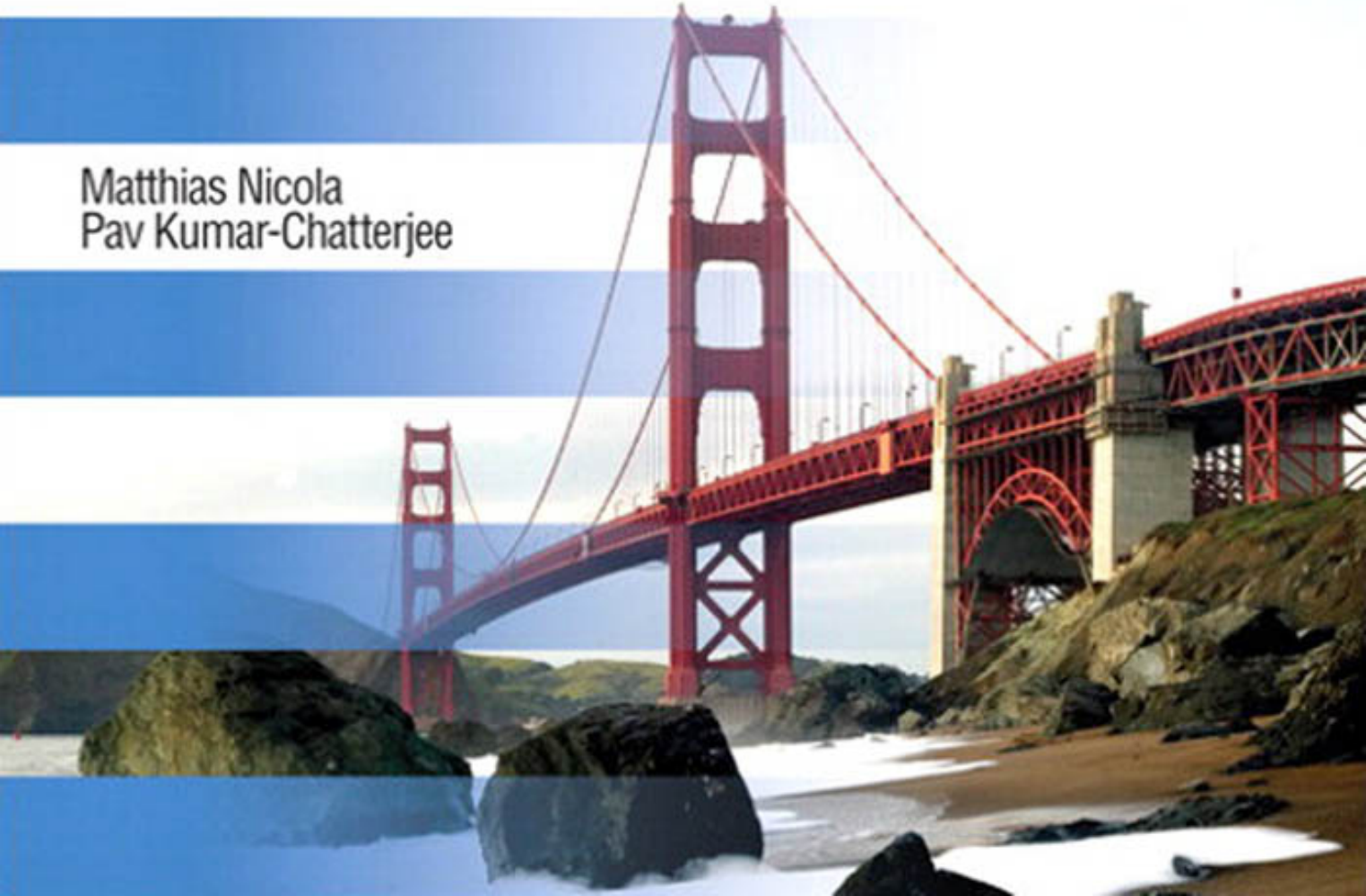
IBM PRESS

# DB2 pureXML Cookbook

## Master the Power of the IBM Hybrid Data Server

Matthias Nicola
Pav Kumar-Chatterjee



IBM Press Program Managers: Steven M. Stansel, Ellice Uffer

Cover design: IBM Corporation

Associate Publisher: Greg Wiegand
Marketing Manager: Kourtnaye Sturgeon
Publicist: Heather Fox
Acquisitions Editor: Bernard Goodwin
Managing Editor: Kristy Hart
Designer: Alan Clements
Project Editor: Andy Beaster
Copy Editor: Paula Lowell
Senior Indexer: Cheryl Lenser
Compositor: Gloria Schurick
Proofreader: Leslie Joseph
Manufacturing Buyer: Dan Uhrig

Published by Pearson plc

Publishing as IBM Press

IBM Press offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com.

For sales outside the U.S., please contact:

International Sales
international@pearson.com.

Library of Congress Cataloging-in-Publication Data

Nicola, Matthias.
DB2 PureXML cookbook : master the power of IBM's hybrid data server / Matthias Nicola and Pav Kumar-Chatterjee.
p. cm.
Includes indexes.
ISBN-13: 978-0-13-815047-1 (hardback : alk. paper)
ISBN-10: 0-13-815047-8 (hardback : alk. paper) 1. IBM Database 2. 2. XML (Document markup language) 3. Database management. I. Kumar-Chatterjee, Pav. II. Title.
QA76.9.D3N525 2009
006.7'4—dc22
2009020222



ISBN-13: 978-0-13-815047-1

ISBN-10: 0-13-815047-8

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing August 2009

# Foreword

In the years since E.F. Codd's groundbreaking work in the 1970s, relational database systems have become ubiquitous in the business world. Today, most of the world's business data is stored in the rows and columns of relational databases. The relational model is ideally suited to applications in which data has a relatively simple and uniform structure, and in which database structure evolves much more slowly than data values.

With the advent of the Web, however, big changes began to occur in the database world, driven by globalization and by dramatic reductions in the cost of storing, transmitting, and processing data. Today, businesses are globally interconnected and exchange large volumes of data with customers, suppliers, and governments. Much of this data consists of things that do not fit neatly into rows and columns, such as medical records, legal documents, incident reports, tax returns, and purchase orders. The new kinds of data tend to be more heterogeneous than traditional business data, having more variation and a more rapidly evolving structure.

In response to the changing requirements of business data, a new generation of standards have appeared. XML has emerged as an international standard for the exchange of self-describing data, unifying structured, unstructured, and semi-structured information formats. XML Schema has been adopted as the metadata syntax for describing the structure of XML documents. Industry-specific XML schemas have been developed for medical, insurance, retail, publishing, banking, and other industries. XPath and XQuery have been adopted as standard languages for retrieving and manipulating data in XML format, and new facilities have been added to the SQL standard for interfacing between relational and XML data.

In DB2, the new generation of XML-related standards is reflected in *pureXML*, a broad new set of XML functionality implemented in both DB2 for z/OS and DB2 for Linux, UNIX, and Windows. pureXML bridges the gap between the XML and relational worlds and makes DB2 a true hybrid database management system. DB2 pureXML stores and indexes XML data alongside relational data in a highly efficient new storage format, and supports XML query languages such as XPath and XQuery alongside the traditional SQL.

pureXML is perhaps the largest new package of functionality in the history of DB2, impacting nearly every aspect of the system. The implementation of pureXML required deep changes in the database kernel, optimization methods, database administrator tools, system utilities, and application programming interfaces. New facilities were added for registering XML schemas and using them to validate stored documents. New kinds of statistics on XML documents had to be gathered and exploited. Facilities for replicated, federated, and partitioned databases had to be updated to accommodate the new XML storage format.

pureXML provides DB2 users with a new level of capability, but using this capability to full advantage requires users to have a new level of sophistication. A new user of pureXML is

confronted with many complex choices. What kinds of data should be represented in XML rather than in normalized tables? How can data be converted between XML and relational formats? How can a hybrid database be designed to take advantage of both data formats? What are the most appropriate uses for SQL, XQuery, and XPath? What kinds of indexes should be maintained on XML data? What is the XML equivalent of a NULL value? These and many other questions are considered in detail in the *DB2 pureXML Cookbook.*

Matthias Nicola has been deeply involved in the design and implementation of DB2 pureXML since its inception. As a Senior Engineer at IBM's Silicon Valley Laboratory, his work has focused on measuring and optimizing the performance of new storage and indexing techniques for XML. After the release of pureXML, he worked with many IBM customers and business partners to create, deploy, and optimize XML applications for government, banking, telecommunications, retail, and other industries.

Pav Kumar-Chatterjee is a technical specialist with many years of experience in consulting with IBM customers throughout the UK and Europe on developing and deploying DB2 and XML solutions.

Through their work with customers, Matthias and Pav have learned how to explain concepts clearly and how to identify and avoid common pitfalls in the application development process. They have also developed a set of "best practices" that they have shared at numerous conferences, classes, workshops, and customer engagements. Between them, Matthias and Pav have accumulated all the knowledge and experience you need to successfully create and deploy solutions using DB2 pureXML. Their expertise is encapsulated in this book in the form of hundreds of practical examples, tested and clearly explained. The book also includes a comprehensive set of questions to test your understanding.

*DB2 pureXML Cookbook* includes both an introduction to basic XML concepts and a comprehensive description of the XML-related features of DB2 for z/OS and DB2 for Linux, UNIX, and Windows. Chapters are organized around tasks that reflect the lifecycle of XML projects, including designing databases, loading and validating data, writing queries and updates, developing applications, optimizing performance, and diagnosing problems. Each topic provides a clear progression from introductory material to more advanced concepts. The writing style is informal and easy to understand for both beginners and experts.

If you are an application developer, database administrator, or system architect, this is the book you need to gain a comprehensive understanding of DB2 pureXML.

Don Chamberlin
IBM Fellow, Emeritus
Almaden Research Center
April 10, 2009

# Preface

In recent years XML has continued to emerge as the de-facto standard for data exchange, because it is flexible, extensible, self-describing, and suitable for any combination of structured and unstructured data. With the increasing use of XML as a pervasive data format, there is a growing need to store, index, query, update, and validate XML documents in database systems. In response to this demand, IBM has developed sophisticated XML data management capabilities that are deeply integrated in the DB2 database system. This novel technology is called *DB2 pureXML* and is available in DB2 for z/OS and DB2 for Linux, UNIX, and Windows. With pureXML, DB2 has evolved into a *hybrid* database system that allows you to manage both XML and relational data in a tightly integrated manner.

The *DB2 pureXML Cookbook* provides the single most comprehensive coverage of DB2's pureXML functionality in DB2 for Linux, UNIX, and Windows as well as DB2 for z/OS. This book is a "cookbook" because it is more than just a description of functions and features ("ingredients"). This book provides "recipes" that show you how to combine the pureXML ingredients to efficiently perform typical user tasks for managing XML data. This book explains DB2 pureXML in more than 700 practical examples, including 250+ XQuery and SQL/XML queries, taking you from simple introductions all the way to advanced scenarios, tuning, and troubleshooting.

Since the first release of DB2 pureXML in 2006 we have worked with numerous companies to help them design, implement, optimize, and deploy XML applications with DB2. In this book we have distilled our experience from these pureXML projects so that you can benefit from proven implementation techniques, best practices, tips and tricks, and performance guidelines that are not described elsewhere.

## Who Should Read This Book?

This book is written for database administrators, application developers, IT architects, and everyone who wants to get a deep technical understanding of DB2's pureXML technology and how to use it most effectively. As a DBA you will learn, for example, how to design and manage XML storage objects, how to index XML data, where to find XML-related information in the DB2 catalog, and how to mange XML with DB2 utilities. Application developers learn, among other things, how to write XML queries and XML updates with XPath, SQL/XML, and XQuery, and how to code XML applications with Java, .NET, C, COBOL, PL/1, PHP, or Perl.

This book is suitable for both beginners and experts. Each topic starts with simple examples, which provide an easy introduction, and works towards advanced concepts and solutions to complex problems. Extensive XML knowledge is not required to read this book because it includes the necessary introductions to XML, XPath, XQuery, XML Schema, and namespaces. These

concepts are explained through numerous examples that are easy to follow. We assume that you have some experience with relational databases and SQL, but we show all the relevant DB2 commands that are required to work through the examples in this book. Appendix C, *Further Reading*, also contains links to additional educational material about both DB2 and XML.

## Coverage of DB2 for z/OS and DB2 for Linux, UNIX, and Windows in This Book

The book describes DB2 pureXML on all supported platforms and versions, which at the time of writing are DB2 9 for z/OS as well as DB2 9.1, 9.5, and 9.7 for Linux, UNIX, and Windows. Many pureXML features and functions are identical across DB2 for Linux, UNIX, and Windows and DB2 for z/OS.

Where platform-specific differences exist we point them out along the way. However, this book does not intend to be a reference that lists all functions and features according to platform and version of DB2. Instead, this book is a "cookbook" that focuses on concepts, examples, and best practices. The capabilities in DB2 for z/OS and DB2 for Linux, UNIX, and Windows continue to grow and converge over time. For the latest information on which feature is available in which version, please consult the respective DB2 information center. DB2 for z/OS also continues to deliver pureXML enhancements via APARs. Please look at APAR II14426, which is an informational APAR that summarizes and links all other XML-related APARs for DB2 on z/OS.

In our work with users who adopt DB2 pureXML we have made the following observation: Some of the users who begin to use DB2 pureXML on Linux, UNIX, and Windows have little or no prior experience with DB2. In contrast, most users who are interested in DB2 pureXML on z/OS are already familiar with DB2 for z/OS in general. This difference is reflected in this book; that is, we describe some DB2 concepts, such as monitoring or the use of DB2 utilities, in more detail for DB2 for Linux, UNIX, and Windows than for DB2 for z/OS.

## Do It Yourself!

The best way to learn a new technology is hands-on. We strongly recommend that you download DB2 Express-C, which is free, and try the concepts that you learn in this book in DB2's sample database. Appendixes A and B contain the necessary information to get you started.

## Don't Hesitate to Ask Questions!

If any pureXML question is not covered in this book, the fastest way to get an answer is to post a question in the DB2 pureXML forum at http://www.ibm.com/developerworks/forums/forum.jspa?forumID=1423.

Whether you seek clarification about specific features or functions, or if you need help with a tricky query, this forum is the right place to ask for help. You are also welcome to contact the

authors directly. If you want to discuss an XML project or if you have comments or feedback on the material in this book—we will be happy to hear from you. Please contact Matthias at mnicola@us.ibm.com and Pav at kumarp2@uk.ibm.com.

## How This Book Is Structured

The *DB2 pureXML Cookbook* takes you through the different tasks and topics that you typically encounter during the life cycle of an XML project. The structure of this book with its 23 chapters is the following:

### Planning

**Chapter 1, *Introduction*,** provides an overview of XML and its differences to relational data, and discusses scenarios where XML has advantages over the relational model. This chapter also includes a summary of the pureXML technology.

**Chapter 2, *Designing XML Data and Applications*,** covers fundamental XML design questions such as choosing between XML elements and attributes, selecting an appropriate XML document granularity, and deciding on a "good" mix of XML and relational data for your application.

### Designing and Populating an XML Database

**Chapter 3, *Designing and Managing XML Storage Objects*,** first explains the tree representation of XML documents and how they are physically stored in DB2. Then it describes how to create and manage tables and table spaces for XML, including compression, reorganization, and partitioning.

**Chapter 4, *Inserting and Retrieving XML Data*,** looks at "full document" operations such as insert, delete, and retrieval of XML documents. This chapter also explains how to handle XML declarations, white space, and reserved characters in XML documents.

**Chapter 5, *Moving XML Data*,** looks at importing, exporting, loading, replicating, and federating XML data in DB2. A technique to split large XML documents into smaller ones is also demonstrated.

### Querying XML Data

**Chapter 6, *Querying XML Data: Introduction and XPath*,** is the first of four chapters on querying XML data. This chapter provides an overview of the different options for querying XML, introduces the XPath and XQuery data model, and describes the XPath language in detail. These concepts are fundamental for the subsequent chapters.

**Chapter 7, *Querying XML Data with SQL/XML***, explains how XPath can be included in SQL statements with the SQL/XML functions `XMLQUERY` and `XMLTABLE` and the `XMLEXISTS` predicate. The use of SQL/XML is illustrated through a rich collection of examples and a discussion of common mistakes and how to avoid them.

**Chapter 8, *Querying XML Data with XQuery*,** introduces the XQuery language, which is a superset of XPath. Among other things, this chapter describes XQuery FLWOR expressions, combinations of SQL and XQuery, and a comparison of XPath, XQuery, and SQL/XML.

**Chapter 9, *Querying XML Data: Advanced XML Queries and Troubleshooting*,** takes querying XML data to the expert level. It demonstrates how to perform grouping, aggregation, and joins over XML data or a mix of XML and relational data. The troubleshooting section discusses **"**bad**"** XML queries, common errors, and how to avoid both.

## Converting, Updating, and Transforming

**Chapter 10, *Producing XML from Relational Data*,** begins the discussion of converting, updating, and transforming data. This chapter explains how to read relational data from existing database tables and construct XML documents from it.

**Chapter 11, *Converting XML to Relational Data*,** describes the opposite of Chapter 10, that is, the process of decomposing or shredding XML documents into relational tables. Two shredding methods are discussed, one using the `XMLTABLE` function and the other using annotated XML Schemas.

**Chapter 12, *Updating and Transforming XML Documents*,** covers three techniques for updating XML documents: Full document replacement, XSLT transformations, and the XQuery Update Facility that allows you to modify, insert, delete, or rename individual elements and attributes within an XML document.

## Performance and Monitoring

**Chapter 13, *Defining and Using XML Indexes*,** is one of two chapters dedicated to performance. It describes how to create XML indexes to improve query performance and explains under which conditions query predicates can or cannot use XML indexes.

**Chapter 14, *Performance and Monitoring*,** looks at analyzing the performance of XML operations with particular emphasis on understanding XML query access plans. A summary of best practices for XML performance in DB2 is also provided.

## Ensuring Data Quality

**Chapter 15, *Managing XML Data with Namespaces*,** introduces XML namespaces and explains how they avoid naming conflicts and ambiguity, thus contributing to data quality. This chapter illustrates how to index, query, update, and construct XML documents that contain namespaces.

**Chapter 16, *Managing XML Schemas*,** first describes how XML Schemas can constrain XML documents in terms of their structure, element and attribute names, data types, and other characteristics. Then this chapter walks you through the concepts of registering, managing, and evolving XML Schemas in DB2.

**Chapter 17, *Validating XML Documents against XML Schemas*,** concentrates on the validation of XML documents to ensure XML data quality in DB2. You can validate XML documents in `INSERT` and `UPDATE` statements, queries, and import and load operations.

## Application Development

**Chapter 18, *Using XML in Stored Procedures, UDFs, and Triggers*,** demonstrates how you can implement application-specific processing logic with XML manipulation in SQL stored procedures, user-defined functions, and triggers.

**Chapter 19, *Performing Full-Text Search*,** describes how the *DB2 Net Search Extender* and *DB2 Text Search* support efficient full-text search in collections of XML documents.

**Chapter 20, *Understanding XML Data Encoding*,** explains internal and external XML encoding, how DB2 determines and handles XML encoding, and how you can avoid code page conversion.

**Chapter 21, *Developing XML Application with DB2*,** contains techniques and best practices for application programs that exchange XML data with the DB2 server. Code samples are provided for Java, .NET, C, COBOL, PL/1, PHP, and Perl programmers.

## Reference Material

**Chapter 22, *Exploring XML Information in the DB2 Catalog*,** is a guide to how XML storage objects, XML indexes, and XML Schemas are listed in the database catalog.

**Chapter 23, *Test Your Knowledge—The DB2 pureXML Quiz*,** offers 82 questions to revisit specific topic areas.

The Appendixes list supporting information and further reading for each chapter.

CHAPTER 11

# Converting XML to Relational Data

This chapter describes methods to convert XML documents to rows in relational tables. This conversion is commonly known as *shredding* or *decomposing* of XML documents. Given the rich support for XML columns in DB2 you might wonder in which cases it can still be useful or necessary to convert XML data to relational format. One common reason for shredding is that existing SQL applications might still require access to the data in relational format. For example, legacy applications, packaged business applications, or reporting software do not always understand XML and have fixed relational interfaces. Therefore you might sometimes find it useful to shred all or some of the data values of an incoming XML document into rows and columns of relational tables.

In this chapter you learn:

- The advantages and disadvantages of shredding and of different shredding methods (section 11.1)
- How to shred XML data to relational tables using `INSERT` statements that contain the `XMLTABLE` function (section 11.2)
- How to use XML Schema annotations that map and shred XML documents to relational tables (section 11.3)

## 11.1 Advantages and Disadvantages of Shredding

The concept of XML shredding is illustrated in Figure 11.1. In this example, XML documents with customer name, address, and phone information are mapped to two relational tables. The documents can contain multiple `phone` elements because there is a one-to-many relationship

between customers and phones. Hence, phone numbers are shredded into a separate table. Each repeating element, such as `phone`, leads to an additional table in the relational target schema. Suppose the customer information can also contain multiple email addresses, multiple accounts, a list of most recent orders, multiple products per order, and other repeating items. The number of tables required in the relational target schema can increase very quickly. Shredding XML into a large number of tables can lead to a complex and unnatural fragmentation of your logical business objects that makes application development difficult and error-prone. Querying the shredded data or reassembling the original documents may require complex multiway joins.

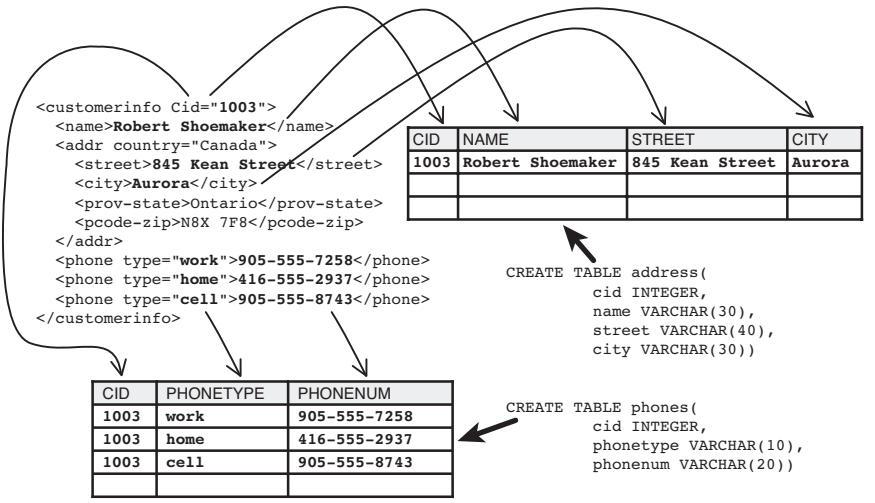



**Figure 11.1** Shredding of an XML document

Depending on the complexity, variability, and purpose of your XML documents, shredding may or may not be a good option. Table 11.1 summarizes the pros and cons of shredding XML data to relational tables.

**Table 11.1** When Shredding Is and Isn't a Good Option

| Shredding Can Be Useful When… | Shredding Is Not A Good Option When… |
|---|---|
| • Incoming XML data is just feeding an existing relational database. | • Your XML data is complex and nested, and difficult to map to a relational schema. |
| • The XML documents do not represent logical business objects that should be preserved. | • Mapping your XML format to a relational schema leads to a large number of tables. |
| • Your primary goal is to enable existing relational applications to access XML data. | • Your XML Schema is highly variable or tends to change over time. |
| • You are happy with your relational schema and would like to use it as much as possible. | • Your primary goal is to manage XML documents as intact business objects. |

**Table 11.1** When Shredding Is and Isn't a Good Option *(Continued)*

| Shredding Can Be Useful When… | Shredding Is Not A Good Option When… |
|---|---|
| • The structure of your XML data is such that it can easily be mapped to relational tables. | • You frequently need to reconstruct the shredded documents or parts of them. |
| • Your XML format is relatively stable and changes to it are rare. | • Ingesting XML data into the database at a high rate is important for your application. |
| • You rarely need to reconstruct the shredded documents. | |
| • Querying or updating the data with SQL is more important than insert performance. | |

In many XML application scenarios the structure and usage of the XML data does not lend itself to easy and efficient shredding. This is the reason why DB2 supports XML columns that allow you to index and query XML data without conversion. Sometimes you will find that your application requirements can be best met with *partial shredding* or *hybrid XML storage*.

- *Partial shredding* means that only a subset of the elements or attributes from each incoming XML document are shredded into relational tables. This is useful if a relational application does not require *all* data values from each XML document. In cases where shredding each document entirely is difficult and requires a complex relational target schema, partial shredding can simplify the mapping to the relational schema significantly.
- *Hybrid XML storage* means that upon insert of an XML document into an XML column, selected element or attribute values are extracted and redundantly stored in relational columns.

If you choose to shred XML documents, entirely or partially, DB2 provides you with a rich set of capabilities to do some or all of the following:

- Perform custom transformations of the data values before insertion into relational columns.
- Shred the same element or attribute value into multiple columns of the same table or different tables.
- Shred multiple different elements or attributes into the same column of a table.
- Specify conditions that govern when certain elements are or are not shredded. For example, shred the address of a customer document only if the `country` is Canada.
- Validate XML documents with an XML Schema during shredding.
- Store the full XML document along with the shredded data.

DB2 9 for z/OS and DB2 9.x for Linux, UNIX, and Windows support two shredding methods:

- SQL `INSERT` statements that use the `XMLTABLE` function. This function navigates into an input document and produces one or multiple relational rows for insert into a relational table.
- Decomposition with an annotated XML Schema. Since an XML Schema defines the structure of XML documents, annotations can be added to the schema to define how elements and attributes are mapped to relational tables.

Table 11.2 and Table 11.3 discuss the advantages and disadvantages of the `XMLTABLE` method and the annotated schema method.

**Table 11.2** Considerations for the `XMLTABLE` Method

| Advantages of the XMLTABLE Method | Disadvantages of the XMLTABLE Method |
|---|---|
| • It allows you to shred data even if you do not have an XML Schema.<br>• It does not require you to understand the XML Schema language or to understand schema annotations for decomposition.<br>• It is generally easier to use than annotated schemas because it is based on SQL and XPath.<br>• You can use familiar XPath, XQuery, or SQL functions and expressions to extract and optionally modify the data values.<br>• It often requires no or little work during XML Schema evolution.<br>• The shredding process can consume data from multiple XML and relational sources, if needed, such as values from DB2 sequences or look-up data from other relational tables.<br>• It can often provide better performance than annotated schema decompositions. | • For each target table that you want to shred into you need one `INSERT` statement.<br>• You might have to combine multiple `INSERT` statements in a stored procedure.<br>• There is no GUI support for implementing the `INSERT` statements and the required `XMLTABLE` functions. You need to be familiar with XPath and SQL/XML. |

**Table 11.3** Considerations for Annotated Schema Decomposition

| Advantages of the Annotated Schema Method | Disadvantages of the Annotated Schema Method |
|---|---|
| • The mapping from XML to relational tables can be defined using a GUI in IBM Data Studio Developer. | • It does not allow shredding without an XML Schema. |
| • If you shred complex XML data into a large number of tables, the coding effort can be lower than with the XMLTABLE approach. | • You might have to manually copy annotations when you start using a new version of your XML Schema. |
| • It offers a bulk mode with detailed diagnostics if some documents fail to shred. | • Despite the GUI support, you need to be familiar with the XML Schema language for all but simple shredding scenarios. |
| | • Annotating an XML Schema can be complex, if the schema itself is complex. |

## 11.2 Shredding with the XMLTABLE Function

The XMLTABLE function is an SQL table function that uses XQuery expressions to create relational rows from an XML input document. For details on the XMLTABLE function, see Chapter 7, *Querying XML Data with SQL/XML*. In this section we describe how to use the XMLTABLE function in an SQL INSERT statement to perform shredding. We use the shredding scenario in Figure 11.1 as an example.

The first step is to create the relational target tables, if they don't already exist. For the scenario in Figure 11.1 the target tables are defined as follows:

```
CREATE TABLE address(cid INTEGER, name VARCHAR(30),
                     street VARCHAR(40), city VARCHAR(30))

CREATE TABLE phones(cid INTEGER, phonetype VARCHAR(10),
                    phonenum VARCHAR(20))
```

Based on the definition of the target tables you construct the INSERT statements that shred incoming XML documents. The INSERT statements have to be of the form INSERT INTO … SELECT … FROM … XMLTABLE, as shown in Figure 11.2. Each XMLTABLE function contains a parameter marker ("?") through which an application can pass the XML document that is to be shredded. SQL typing rules require the parameter marker to be cast to the appropriate data type. The SELECT clause selects columns produced by the XMLTABLE function for insert into the address and phones tables, respectively.

```
INSERT INTO address(cid, name, street, city)
  SELECT x.custid, x.custname, x.str, x.place
  FROM XMLTABLE('$i/customerinfo' PASSING CAST(? AS XML) AS "i"
        COLUMNS
          custid    INTEGER     PATH '@Cid',
          custname  VARCHAR(30) PATH 'name',
          str       VARCHAR(40) PATH 'addr/street',
          place     VARCHAR(30) PATH 'addr/city' ) AS x ;


INSERT INTO phones(cid, phonetype, phonenum)
  SELECT x.custid, x.ptype, x.number
  FROM XMLTABLE('$i/customerinfo/phone'
                PASSING CAST(? AS XML) AS "i"
        COLUMNS
          custid    INTEGER     PATH '../@Cid',
          number    VARCHAR(15) PATH '.',
          ptype     VARCHAR(10) PATH './@type') AS x ;
```

**Figure 11.2** Inserting XML element and attribute values into relational columns

To populate the two target tables as illustrated in Figure 11.1, both `INSERT` statements have to be executed with the same XML document as input. One approach is that the application issues both `INSERT` statements in one transaction and binds the same XML document to the parameter markers for both statements. This approach works well but can be optimized, because the same XML document is sent from the client to the server and parsed at the DB2 server twice, once for each `INSERT` statement. This overhead can be avoided by combining both `INSERT` statements in a single stored procedure. The application then only makes a single stored procedure call and passes the input document once, regardless of the number of `INSERT` statements in the stored procedure. Chapter 18, *Using XML in Stored Procedures, UDFs, and Triggers*, demonstrates such a stored procedure as well as other examples of manipulating XML data in stored procedures and user-defined functions.

Alternatively, the `INSERT` statements in Figure 11.2 can read a set of input documents from an XML column. Suppose the documents have been loaded into the XML column `info` of the `customer` table. Then you need to modify one line in each of the `INSERT` statements in Figure 11.2 to read the input document from the `customer` table:

```
FROM customer, XMLTABLE('$i/customerinfo' PASSING info AS "i"
```

Loading the input documents into a staging table can be advantageous if you have to shred many documents. The `LOAD` utility parallelizes the parsing of XML documents, which reduces the time to move the documents into the database. When the documents are stored in an XML column in parsed format, the `XMLTABLE` function can shred the documents *without* XML parsing.

The `INSERT` statements can be enriched with XQuery or SQL functions or joins to tailor the shredding process to specific requirements. Figure 11.3 provides an example. The `SELECT` clause

contains the function RTRIM to remove trailing blanks from the column x.ptype. The row-generating expression of the XMLTABLE function contains a predicate that excludes home phone numbers from being shredded into the target table. The column-generating expression for the phone numbers uses the XQuery function normalize-space, which strips leading and trailing whitespace and replaces each internal sequence of whitespace characters with a single blank character. The statement also performs a join to the lookup table areacodes so that a phone number is inserted into the phones table only if its area code is listed in the areacodes table.

```
INSERT INTO phones(cid, phonetype, phonenum)
  SELECT x.custid, RTRIM(x.ptype), x.number
  FROM areacodes a,
       XMLTABLE('$i/customerinfo/phone[@type != "home"]'
                PASSING CAST(? AS XML) AS "i"
        COLUMNS
          custid    INTEGER     PATH '../@Cid',
          number    VARCHAR(15) PATH 'normalize-space(.)',
          ptype     VARCHAR(10) PATH './@type') AS x
  WHERE SUBSTR(x.number,1,3) = a.code;
```

**Figure 11.3** Using functions and joins to customize the shredding

### 11.2.1 Hybrid XML Storage

In many situations the complexity of the XML document structures makes shredding difficult, inefficient, and undesirable. Besides the performance penalty of shredding, scattering the values of an XML document across a large number of tables can make it difficult for an application developer to understand and query the data. To improve XML insert performance and to reduce the number of tables in your database, you may want to store XML documents in a *hybrid* manner. This approach extracts the values of selected XML elements or attributes and stores them in relational columns alongside the full XML document.

The example in the previous section used two tables, address and phones, as the target tables for shredding the customer documents. You might prefer to use just a single table that contains the customer cid, name, and city values in relational columns and the full XML document with the repeating phone elements and other information in an XML column. You can define the following table:

```
CREATE TABLE hybrid(cid INTEGER NOT NULL PRIMARY KEY,
                    name VARCHAR(30), city VARCHAR(25), info XML)
```

Figure 11.4 shows the INSERT statement to populate this table. The XMLTABLE function takes an XML document as input via a parameter marker. The column definitions in the XMLTABLE function produce four columns that match the definition of the target table hybrid. The row-generating expression in the XMLTABLE function is just $i, which produces the full input document. This expression is the input for the column-generating expressions in the COLUMNS clause of the XMLTABLE function. In particular, the column expression '.' returns the full input

document as-is and produces the XML column `doc` for insert into the `info` column of the target table.

```
INSERT INTO hybrid(cid, name, city, info)
  SELECT x.custid, x.custname, x.city, x.doc
  FROM XMLTABLE('$i' PASSING CAST(? AS XML) AS "i"
        COLUMNS
          custid   INTEGER     PATH 'customerinfo/@Cid',
          custname VARCHAR(30) PATH 'customerinfo/name',
          city     VARCHAR(25) PATH 'customerinfo/addr/city',
          doc      XML         PATH '.' ) AS x;
```

**Figure 11.4** Storing an XML document in a hybrid fashion

It is currently not possible to define check constraints in DB2 to enforce the integrity between relational columns and values in an XML document in the same row. You can, however, define `INSERT` and `UPDATE` triggers on the table to populate the relational columns automatically whenever a document is inserted or updated. Triggers are discussed in Chapter 18, *Using XML in Stored Procedures, UDFs, and Triggers*.

It can be useful to test such `INSERT` statements in the DB2 Command Line Processor (CLP). For this purpose you can replace the parameter marker with a literal XML document as shown in Figure 11.5. The literal document is a string that must be enclosed in single quotes and converted to the data type `XML` with the `XMLPARSE` function. Alternatively, you can read the input document from the file system with one of the UDFs that were introduced in Chapter 4, *Inserting and Retrieving XML Data*. The use of a UDF is demonstrated in Figure 11.6.

```
INSERT INTO hybrid(cid, name, city, info)
  SELECT x.custid, x.custname, x.city, x.doc
  FROM XMLTABLE('$i' PASSING
    XMLPARSE(document
     '<customerinfo 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>') AS "i"
        COLUMNS
          custid   INTEGER     PATH 'customerinfo/@Cid',
          custname VARCHAR(30) PATH 'customerinfo/name',
          city     VARCHAR(25) PATH 'customerinfo/addr/city',
          doc      XML         PATH '.' ) AS x;
```

**Figure 11.5** Hybrid insert statement with a literal XML document

```
INSERT INTO hybrid(cid, name, city, info)
  SELECT x.custid, x.custname, x.city, x.doc
  FROM XMLTABLE('$i' PASSING
    XMLPARSE(document
      blobFromFile('/xml/mydata/cust0037.xml')) AS "i"
        COLUMNS
          custid   INTEGER     PATH 'customerinfo/@Cid',
          custname VARCHAR(30) PATH 'customerinfo/name',
          city     VARCHAR(25) PATH 'customerinfo/addr/city',
          doc      XML         PATH '.' ) AS x;
```

**Figure 11.6** Hybrid insert statement with a "FromFile" UDF

The insert logic in Figure 11.4, Figure 11.5, and Figure 11.6 is identical. The only difference is how the input document is provided: via a parameter marker, as a literal string that is enclosed in single quotes, or via a UDF that reads a document from the file system.

### 11.2.2 Relational Views over XML Data

You can create relational views over XML data using `XMLTABLE` expressions. This allows you to provide applications with a relational or hybrid view of the XML data without actually storing the data in a relational or hybrid format. This can be useful if you want to avoid the overhead of converting large amounts of XML data to relational format. The basic `SELECT ... FROM ... XMLTABLE` constructs that were used in the `INSERT` statements in the previous section can also be used in `CREATE VIEW` statements.

As an example, suppose you want to create a relational view over the elements of the XML documents in the `customer` table to expose the customer identifier, name, street, and city values. Figure 11.7 shows the corresponding view definition plus an SQL query against the view.

```
CREATE VIEW custview(id, name, street, city)
AS
  SELECT x.custid, x.custname, x.str, x.place
  FROM customer,
       XMLTABLE('$i/customerinfo' PASSING info AS "i"
        COLUMNS
          custid    INTEGER     PATH '@Cid',
          custname  VARCHAR(30) PATH 'name',
          str       VARCHAR(40) PATH 'addr/street',
          place     VARCHAR(30) PATH 'addr/city' ) AS x;


SELECT id, name FROM custview WHERE city = 'Aurora';

ID          NAME
----------- ------------------------------
       1003 Robert Shoemaker

  1 record(s) selected.
```

**Figure 11.7** Creating a view over XML data

The query over the view in Figure 11.7 contains an SQL predicate for the `city` column in the view. The values in the `city` column come from an XML element in the underlying XML column. You can speed up this query by creating an XML index on `/customerinfo/addr/city` for the `info` column of the `customer` table. DB2 9 for z/OS and DB2 9.7 for Linux, UNIX, and Windows are able to convert the relational predicate `city = 'Aurora'` into an XML predicate on the underlying XML column so that the XML index can be used. This is not possible in DB2 9.1 and DB2 9.5 for Linux, UNIX, and Windows. In these previous versions of DB2, include the XML column in the view definition and write the search condition as an XML predicate, as in the following query. Otherwise an XML index cannot be used.

```
SELECT id, name
FROM custview
WHERE XMLEXISTS('$INFO/customerinfo/addr[city = "Aurora"]')
```

## 11.3 Shredding with Annotated XML Schemas

This section describes another approach to shredding XML documents into relational tables. The approach is called *annotated schema shredding* or *annotated schema decomposition* because it is based on annotations in an XML Schema. These annotations define how XML elements and attributes in your XML data map to columns in your relational tables.

To perform annotated schema shredding, take the following steps:

- Identify or create the relational target tables that will hold the shredded data.
- Annotate your XML Schema to define the mapping from XML to the relational tables.
- Register the XML Schema in the DB2 XML Schema Repository.
- Shred XML documents with Command Line Processor commands or built-in stored procedures.

Assuming you have defined the relational tables that you want to shred into, let's look at annotating an XML Schema.

### 11.3.1 Annotating an XML Schema

Schema annotations are additional elements and attributes in an XML Schema to provide mapping information. DB2 can use this information to shred XML documents to relational tables. The annotations do not change the semantics of the original XML Schema. If a document is valid for the annotated schema then it is also valid for the original schema, and vice versa. You can use an annotated schema to validate XML documents just like the original XML Schema. For an introduction to XML Schemas, see Chapter 16, *Managing XML Schemas*.

The following is one line from an XML Schema:

```
<xs:element name="street" type="xs:string" minOccurs="1"/>
```

This line defines an XML element called `street` and declares that its data type is `xs:string` and that this element has to occur at least once. You can add a simple annotation to this element definition to indicate that the element should be shredded into the column `STREET` of the table `ADDRESS`. The annotation consists of two additional attributes in the element definition, as follows:

```
<xs:element name="street" type="xs:string" minOccurs="1"
  db2-xdb:rowSet="ADDRESS" db2-xdb:column="STREET"/>
```

The same annotation can also be provided as schema elements instead of attributes, as shown next. You will see later in Figure 11.8 why this can be useful.

```
<xs:element name="street" type="xs:string" minOccurs="1">
  <xs:annotation>
    <xs:appinfo>
      <db2-xdb:rowSetMapping>
        <db2-xdb:rowSet>ADDRESS</db2-xdb:rowSet>
        <db2-xdb:column>STREET</db2-xdb:column>
      </db2-xdb:rowSetMapping>
    </xs:appinfo>
  </xs:annotation>
<xs:element/>
```

The prefix `xs` is used for all constructs that belong to the XML Schema language, and the prefix `db2-xdb` is used for all DB2-specific schema annotations. This provides a clear distinction and ensures that the annotated schema validates the same XML documents as the original schema.

There are 14 different types of annotations. They allow you to specify what to shred, where to shred to, how to filter or transform the shredded data, and in which order to execute inserts into the target tables. Table 11.4 provides an overview of the available annotations, broken down into logical groupings by user task. The individual annotations are further described in Table 11.5.

**Table 11.4** Overview and Grouping of Schema Annotations

| If You Want to | Use This Annotation |
|---|---|
| Specify the target tables to shred into | `db2-xdb:rowSet`<br>`db2-xdb:column`<br>`db2-xdb:SQLSchema`<br>`db2-xdb:defaultSQLSchema` |
| Specify what to shred | `db2-xdb:contentHandling` |
| Transform data values while shredding | `db2-xdb:expression`<br>`db2-xdb:normalization`<br>`db2-xdb:truncate` |
| Filter data | `db2-xdb:condition`<br>`db2-xdb:locationPath` |

*(continues)*

**Table 11.4** Overview and Grouping of Schema Annotations *(Continued)*

| If You Want to | Use This Annotation |
|---|---|
| Map an element or attribute to multiple columns | `db2-xdb:rowSetMapping` |
| Map several elements or attributes to the same column | `db2-xdb:table` |
| Define the order in which rows are inserted into the target table, to avoid referential integrity violations | `db2-xdb:rowSetOperationOrder`<br>`db2-xdb:order` |

**Table 11.5** XML Schema Annotations

| Annotation | Description |
|---|---|
| `db2-xdb:defaultSQLSchema` | The default relational schema for the target tables. |
| `db2-xdb:SQLSchema` | Overrides the default schema for individual tables. |
| `db2-xdb:rowSet` | The table name that the element or attribute is mapped to |
| `db2-xdb:column` | The column name that the element or attribute is mapped to. |
| `db2-xdb:contentHandling` | For an XML element, this annotation defines how to derive the value that will be inserted into the target column. You can chose the text value of just this element (`text`), the concatenation of this element's text and the text of all its descendant nodes (`stringValue`), or the serialized XML (including all tages) of this element and all descendants (`serializeSubtree`). If you omit this annotation, DB2 chooses an appropriate default based on the nature of the respective element. |
| `db2-xdb:truncate` | Specifies whether a value should be truncated if its length is greater than the length of the target column. |
| `db2-xdb:normalization` | Specifies how to treat whitespace—valid values are `whitespaceStrip`, `canonical`, and `original` |
| `db2-xdb:expression` | Specifies an expression that is to be applied to the data before insertion into the target table. |

**Table 11.5** XML Schema Annotations *(Continued)*

| Annotation | Description |
|---|---|
| db2-xdb:locationPath | Filters based on the XML context. For example, if it is a customer address then shred to the cust table; if it is an employee address then shred to the employee table. |
| db2-xdb:condition | Specifies value conditions so that data is inserted into a target table only if all conditions are true. |
| db2-xdb:rowSetMapping | Enables users to specify multiple mappings, to the same or different tables, for an element or attribute. |
| db2-xdb:table | Maps multiple elements or attributes to a single column. |
| db2-xdb:order | Specifies the insertion order of rows among multiple tables. |
| db2-xdb:rowSetOperationOrder | Groups together multiple db2-xdb:order annotations. |

To demonstrate annotated schema decomposition we use the shredding scenario in Figure 11.1 as an example. Assume that the target tables have been defined as shown in Figure 11.1. An annotated schema that defines the desired mapping is provided in Figure 11.8. Let's look at the lines that are highlighted in bold font. The first bold line declares the namespace prefix db2-xdb, which is used throughout the schema to distinguish DB2-specific annotations from regular XML Schema tags. The first use of this prefix is in the annotation db2-xdb:defaultSQLSchema, which defines the relational schema of the target tables. The next annotation occurs in the definition of the element name. The two annotation attributes db2-xdb:rowSet="ADDRESS" and db2-xdb:column="NAME" define the target table and column for the name element. Similarly, the street and city elements are also mapped to respective columns of the ADDRESS table. The next two annotations map the phone number and the type attribute to columns in the PHONES table. The last block of annotations belongs to the XML Schema definition of the Cid attribute. Since the Cid attribute value becomes the join key between the ADDRESS and the PHONE table, it has to be mapped to both tables. Two row set mappings are necessary, which requires the use of *annotation elements* instead of *annotation attributes*. The first db2-xdb:rowSetMapping maps the Cid attribute to the CID column in the ADDRESS table. The second db2-xdb:rowSet Mapping assigns the Cid attribute to the CID column in the PHONES table.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified"
 xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2/xdb1" >
  <xs:annotation>
    <xs:appinfo>
    <db2-xdb:defaultSQLSchema>db2admin</db2-xdb:defaultSQLSchema>
    </xs:appinfo>
  </xs:annotation>
```

**Figure 11.8** Annotated schema to implement the shredding in Figure 11.1 *(continues)*

```
  <xs:element name="customerinfo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string" minOccurs="1"
         db2-xdb:rowSet="ADDRESS" db2-xdb:column="NAME"/>
        <xs:element name="addr" minOccurs="1"
         maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="street" type="xs:string"
               minOccurs="1" db2-xdb:rowSet="ADDRESS"
               db2-xdb:column="STREET"/>
              <xs:element name="city" type="xs:string"
               minOccurs="1" db2-xdb:rowSet="ADDRESS"
               db2-xdb:column="CITY"/>
              <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" minOccurs="0"
         maxOccurs="unbounded" db2-xdb:rowSet="PHONES"
         db2-xdb:column="PHONENUM">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="type" form="unqualified"
                 type="xs:string" db2-xdb:rowSet="PHONES"
                 db2-xdb:column="PHONETYPE"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="Cid" type="xs:integer">
        <xs:annotation>
          <xs:appinfo>
            <db2-xdb:rowSetMapping>
              <db2-xdb:rowSet>ADDRESS</db2-xdb:rowSet>
              <db2-xdb:column>CID</db2-xdb:column>
            </db2-xdb:rowSetMapping>
            <db2-xdb:rowSetMapping>
              <db2-xdb:rowSet>PHONES</db2-xdb:rowSet>
              <db2-xdb:column>CID</db2-xdb:column>
            </db2-xdb:rowSetMapping>
          </xs:appinfo>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

**Figure 11.8** Annotated schema to implement the shredding in Figure 11.1 *(Continued)*

### 11.3.2 Defining Schema Annotations Visually in IBM Data Studio

You can add annotations to an XML Schema manually, using any text editor or XML Schema editor. Alternatively, you can use the Annotated XSD Mapping Editor in IBM Data Studio Developer. To invoke the editor, right-click on an XML Schema name and select `Open With`, `Annotated XSD Mapping Editor`. A screenshot of the mapping editor is shown in Figure 11.9. The left side of the editor shows the hierarchical document structure defined by the XML Schema (`Source`). The right side shows the tables and columns of the relational target schema (`Target`). You can add mapping relationships by connecting source items with target columns. There is also a discover function to find probable relationships. Mapped relationships are represented in the mapping editor by lines drawn between source elements and target columns.

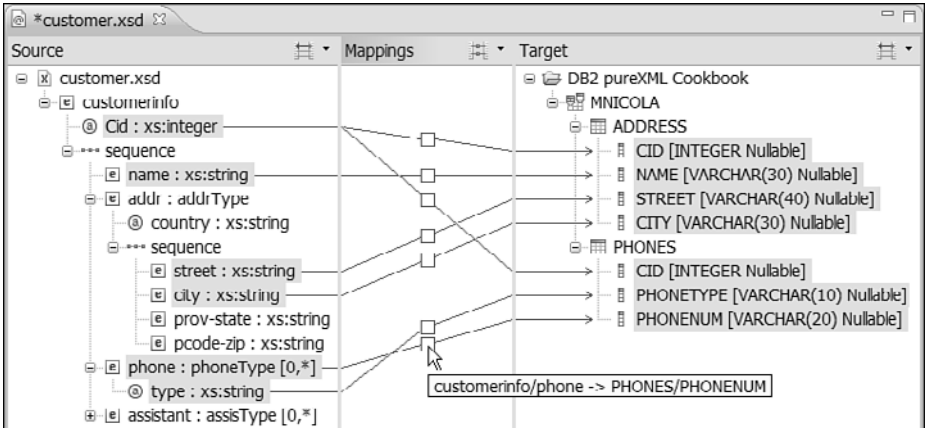



**Figure 11.9** Annotated XSD Mapping Editor in Data Studio Developer

### 11.3.3 Registering an Annotated Schema

After you have created your annotated XML Schema you need to register it in the XML Schema Repository of the database. DB2's XML Schema Repository is described in detail in Chapter 16, *Managing XML Schemas*. For the annotated schema in Figure 11.8 it is sufficient to issue the `REGISTER XMLSCHEMA` command with its `COMPLETE` and `ENABLE DECOMPOSITION` options as shown in Figure 11.10. In this example the XML Schema is assumed to reside in the file `/xml/myschemas/cust2.xsd`. Upon registration it is assigned the SQL identifier `db2admin.cust2xsd`. This identifier can be used to reference the schema later. The `COMPLETE` option of the command indicates that there are no additional XML Schema documents to be added. The option `ENABLE DECOMPOSITION` indicates that this XML Schema can be used not only for document validation but also for shredding.

```
REGISTER XMLSCHEMA 'http://pureXMLcookbook.org'
FROM '/xml/myschemas/cust2.xsd'
AS db2admin.cust2xsd COMPLETE ENABLE DECOMPOSITION;
```

**Figure 11.10** Registering an annotated XML schema

Figure 11.11 shows that you can query the DB2 catalog view `syscat.xsrobjects` to determine whether a registered schema is enabled for decomposition (`Y`) or not (`N`).

```
SELECT SUBSTR(objectname,1,10) AS objectname,
       status, decomposition
FROM syscat.xsrobjects ;


OBJECTNAME STATUS DECOMPOSITION
---------- ------ -------------
CUST2XSD   C      Y
```

**Figure 11.11** Checking the status of an annotated XML schema

The `DECOMPOSITION` status of an annotated schema is automatically changed to `X` (*inoperative*) and shredding is disabled, if any of the target tables are dropped or a target column is altered. No warning is issued when that happens and subsequent attempts to use the schema for shredding fail. You can also use the following commands to disable and enable an annotated schema for shredding:

```
ALTER XSROBJECT cust2xsd DISABLE DECOMPOSITION;
ALTER XSROBJECT cust2xsd ENABLE DECOMPOSITION;
```

### 11.3.4 Decomposing One XML Document at a Time

After you have registered and enabled the annotated XML Schema you can decompose XML documents with the `DECOMPOSE XML DOCUMENT` command or with a built-in stored procedure. The `DECOMPOSE XML DOCUMENT` command is convenient to use in the DB2 Command Line Processor (CLP) while the stored procedure can be called from an application program or the CLP. The CLP command takes two parameters as input: the filename of the XML document that is to be shredded and the SQL identifier of the annotated schema, as in the following example:

```
DECOMPOSE XML DOCUMENT /xml/mydocuments/cust01.xml
  XMLSCHEMA db2admin.cust2xsd VALIDATE;
```

The keyword `VALIDATE` is optional and indicates whether XML documents should be validated against the schema as part of the shredding process. While shredding, DB2 traverses both the XML document and the annotated schema and detects fundamental schema violations even if the `VALIDATE` keyword is not specified. For example, the shredding process fails with an error if a

mandatory element is missing, even if this element is not being shredded and the VALIDATE keyword is omitted. Similarly, extraneous elements or data type violations also cause the decomposition to fail. The reason is that the shredding process walks through the annotated XML Schema and the instance document in lockstep and therefore detects many schema violations "for free" even if the XML parser does not perform validation.

To decompose XML documents from an application program, use the stored procedure XDBDECOMPXML. The parameters of this stored procedure are shown in Figure 11.12 and described in Table 11.6.

```
>>-XDBDECOMPXML--(--rschema--,--xmlschemaname--,--xmldoc--,---->

>--documentid--,--validation--,--reserved--,--reserved--,------>

>--reserved--)-------------------------------------------------><
```

**Figure 11.12** Syntax and parameters of the stored procedure XDBDECOMPXML

**Table 11.6** Description of the Parameters of the Stored Procedure XDBDECOMPXML

| Parameter | Description |
|---|---|
| rschema | The relational schema part of the two-part SQL identifier of the annotated XML Schema. For example, if the SQL identifier of the XML Schema is db2admin.cust2xsd, then you should pass the string 'db2admin' to this parameter. In DB2 for z/OS this value must be either 'SYSXSR' or NULL. |
| xmlschemaname | The second part of the two-part SQL identifier of the annotated XML Schema. If the SQL identifier of the XML Schema is db2admin.cust2xsd, then you pass the string 'cust2xsd' to this parameter. This value cannot be NULL. |
| xmldoc | In DB2 for Linux, UNIX, and Windows, this parameter is of type BLOB(1M) and takes the XML document to be decomposed. In DB2 for z/OS this parameter is of type CLOB AS LOCATOR. This parameter cannot be NULL. |
| documentid | A string that the caller can use to identify the input XML document. The value provided will be substituted for any use of $DECOMP_DOCUMENTID specified in the db2-xdb:expression or db2-xdb:condition annotations. |
| validation | Possible values are: 0 (no validation) and 1 (validation is performed). This parameter does not exist in DB2 for z/OS. |
| reserved | Parameters reserved for future use. The values passed for these arguments must be NULL. These parameters do not exist in DB2 for z/OS. |

A Java code snippet that calls the stored procedure using parameter markers is shown in Figure 11.13

```
CallableStatement callStmt = con.prepareCall(
"call SYSPROC.XDBDECOMPXML(?,?,?,?,?, null, null, null)");

File xmldoc = new File("c:\mydoc.xml");
FileInputStream xmldocis = new FileInputStream(xmldoc);

callStmt.setString(1, "db2admin" );
callStmt.setString(2, "cust2xsd" );

// document to be shredded:
callStmt.setBinaryStream(3,xmldocis,(int)xmldoc.length() );

callStmt.setString(4, "mydocument26580" );

// no schema validation in this call:
callStmt.setInt(5, 0);

callStmt.execute();
```

**Figure 11.13** Java code that invokes the stored procedure `XDBDECOMPXML`

While the input parameter for XML documents is of type `CLOB AS LOCATOR` in DB2 for z/OS, it is of type `BLOB(1M)` in DB2 for Linux, UNIX, and Windows. If you expect your XML documents to be larger than 1MB, use one of the stored procedures listed in Table 11.7. These stored procedures are all identical except for their name and the size of the input parameter `xmldoc`. When you call a stored procedure, DB2 allocates memory according to the *declared* size of the input parameters. For example, if all of your input documents are at most 10MB in size, the stored procedure `XDBDECOMPXML10MB` is a good choice to conserve memory.

**Table 11.7** Stored Procedures for Different Document Sizes (DB2 for Linux, UNIX, and Windows)

| Stored Procedure | Document Size | Supported since |
|---|---|---|
| `XDBDECOMPXML` | ≤1MB | DB2 9.1 |
| `XDBDECOMPXML10MB` | ≤10MB | DB2 9.1 |
| `XDBDECOMPXML25MB` | ≤25MB | DB2 9.1 |
| `XDBDECOMPXML50MB` | ≤50MB | DB2 9.1 |
| `XDBDECOMPXML75MB` | ≤75MB | DB2 9.1 |
| `XDBDECOMPXML100MB` | ≤100MB | DB2 9.1 |
| `XDBDECOMPXML500MB` | ≤500MB | DB2 9.5 FP3 |

**Table 11.7** Stored Procedures for Different Document Sizes (DB2 for Linux, UNIX, and Windows) *(Continued)*

| Stored Procedure | Document Size | Supported since |
|---|---|---|
| XDBDECOMPXML1GB | ≤1GB | DB2 9.5 FP3 |
| XDBDECOMPXML1_5GB | ≤1.5GB | DB2 9.7 |
| XDBDECOMPXML2GB | ≤2GB | DB2 9.7 |

For platform compatibility, DB2 for z/OS supports the procedure XDBDECOMPXML100MB with the same parameters as DB2 for Linux, UNIX, and Windows, including the parameter for validation.

### 11.3.5 Decomposing XML Documents in Bulk

DB2 9.7 for Linux, UNIX, and Windows introduces a new stored procedure called XDB_DECOMP_XML_FROM_QUERY. It uses an annotated schema to decompose one or multiple XML documents selected from a column of type XML, BLOB, or VARCHAR FOR BIT DATA. The main difference to the procedure XDBDECOMPXML is that XDB_DECOMP_XML_FROM_QUERY takes an SQL query as a parameter and executes it to obtain the input documents from a DB2 table. For a large number of documents, a LOAD operation followed by a "bulk decomp" can be more efficient than shredding these documents with a separate stored procedure call for each document. Figure 11.14 shows the parameters of this stored procedure. The parameters commit_count and allow_access are similar to the corresponding parameters of DB2's IMPORT utility. The parameters total_docs, num_docs_decomposed, and result_report are output parameters that provide information about the outcome of the bulk shredding process. All parameters are explained in Table 11.8.

```
>>--XDB_DECOMP_XML_FROM_QUERY--(--rschema--,--xmlschema--,-->

>--query--,--validation--,--commit_count--,--allow_access--,---->

>--reserved--,--reserved2--,--continue_on_error--,-------------->

>--total_docs--,--num_docs_decomposed--,--result_report--)-->
```

**Figure 11.14** The stored procedure XDB_DECOMP_XML_FROM_QUERY

**Table 11.8** Parameters for `XDB_DECOMP_XML_FROM_QUERY`

| Parameter | Description |
|---|---|
| `rschema` | Same as for `XDBDECOMPXML`. |
| `xmlschema` | Same as `xmlschemaname` for `XDBDECOMPXML`. |
| `query` | A query string of type `CLOB(1GB)`, which cannot be `NULL`. The query must be an SQL or SQL/XML `SELECT` statement and must return two columns. The first column must contain a unique document identifier for each XML document in the second column of the result set. The second column contains the XML documents to be shredded and must be of type `XML`, `BLOB`, `VARCHAR FOR BIT DATA`, or `LONG VARCHAR FOR BIT DATA`. |
| `validation` | Possible values are: `0` (no validation) and `1` (validation is performed). |
| `commit_count` | An integer value equal to or greater than `0`. A value of `0` means the stored procedure does not perform any commits. A value of `n` means that a commit is performed after every `n` successful document decompositions. |
| `allow_access` | A value of `1` or `0`. If the value is `0`, then the stored procedure acquires an exclusive lock on all tables that are referenced in the annotated XML Schema. If the value is `1`, then the stored procedure acquires a shared lock. |
| `reserved`, `reserved2` | These parameters are reserved for future use and must be `NULL`. |
| `continue_on_error` | Can be `1` or `0`. A value of `0` means the procedure stops upon the first document that cannot be decomposed; for example, if the document does not match the XML Schema. |
| `total_docs` | An output parameter that indicates the total number of documents that the procedure *tried* to decompose. |
| `num_docs_decomposed` | An output parameter that indicates the number of documents that were *successfully* decomposed. |
| `result_report` | An output parameter of type `BLOB(2GB)`. It contains an XML document that provides diagnostic information for each document that was not successfully decomposed. This report is not generated if all documents shredded successfully. The reason this is a `BLOB` field (rather than `CLOB`) is to avoid codepage conversion and potential truncation/data loss if the application code page is materially different from the database codepage. |

Figure 11.15 shows an invocation of the `XDB_DECOMP_XML_FROM_QUERY` stored procedure in the CLP. This stored procedure call reads all XML documents from the `info` column of the `customer` table and shreds them with the annotated XML Schema `db2admin.cust2xsd`. The procedure commits every 25 documents and does not stop if a document cannot be shredded.

```
call SYSPROC.XDB_DECOMP_XML_FROM_QUERY
('DB2ADMIN', 'CUST2XSD', 'SELECT cid, info FROM customer',
  0, 25, 1, NULL, NULL, '1',?,?,?) ;

  Value of output parameters
  --------------------------
  Parameter Name  : TOTALDOCS
  Parameter Value : 100

  Parameter Name  : NUMDOCSDECOMPOSED
  Parameter Value : 100

  Parameter Name  : RESULTREPORT
  Parameter Value : x''

  Return Status = 0
```

**Figure 11.15** Calling the procedure SYSPROC.XDB_DECOMP_XML_FROM_QUERY

If you frequently perform bulk shredding in the CLP, use the command DECOMPOSE XML DOCUMENTS instead of the stored procedure. It is more convenient for command-line use and performs the same job as the stored procedure XDB_DECOMP_XML_FROM_QUERY. Figure 11.16 shows the syntax of the command. The various clauses and keywords of the command have the same meaning as the corresponding stored procedure parameters. For example, ***query*** is the SELECT statement that provides the input documents, and ***xml-schema-name*** is the two-part SQL identifier of the annotated XML Schema.

```
>>-DECOMPOSE XML DOCUMENTS IN----'query'----XMLSCHEMA------->

                                 .-ALLOW NO ACCESS-.
>--xml-schema-name--+----------+--+-----------------+---------->
                    '-VALIDATE-'  '-ALLOW ACCESS----'

>--+----------------------+--+-------------------+-------------->
   '-COMMITCOUNT--integer-'  '-CONTINUE_ON_ERROR-'

>--+--------------------------+-------------------------------><
   '-MESSAGES--message-file-'
```

**Figure 11.16** Syntax for the DECOMPOSE XML DOCUMENTS command

Figure 11.17 illustrates the execution of the DECOMPOSE XML DOCUMENTS command in the DB2 Command Line Processor.

```
DECOMPOSE XML DOCUMENTS IN 'SELECT cid, info FROM customer'
XMLSCHEMA db2admin.cust2xsd MESSAGES decomp_errors.xml ;

DB216001I  The DECOMPOSE XML DOCUMENTS command successfully
  decomposed all "100" documents.
```

**Figure 11.17** Example of the DECOMPOSE XML DOCUMENTS command

If you don't specify a ***message-file*** then the error report is written to standard output. Figure 11.18 shows a sample error report. For each document that failed to shred, the error report shows the document identifier (`xdb:documentId`). This identifier is obtained from the first column that is produced by the SQL statement in the `DECOMPOSE XML DOCUMENTS` command. The error report also contains the DB2 error message for each document that failed. Figure 11.18 reveals that document 1002 contains an unexpected XML attribute called `status`, and that document 1005 contains an element or attribute value `abc` that is invalid because the XML Schema expected to find a value of type `xs:integer`. If you need more detailed information on why a document is not valid for a given XML Schema, use the stored procedure `XSR_GET_PARSING_DIAGNOSTICS`, which we discuss in section 17.6, *Diagnosing Validation and Parsing Errors*.

```
<?xml version='1.0' ?>
<xdb:errorReport
     xmlns:xdb="http://www.ibm.com/xmlns/prod/db2/xdb1">
   <xdb:document>
      <xdb:documentId>1002</xdb:documentId>
      <xdb:errorMsg>SQL16271N Unknown attribute "status" at or
       near line “1" in document "1002".</xdb:errorMsg>
   </xdb:document>
   <xdb:document>
      <xdb:documentId>1005</xdb:documentId>
      <xdb:errorMsg> SQL16267N An XML value "abc" at or near
        line "1" in document "1005" is not valid according to
        its declared XML schema type "xs:integer" or is outside
        the supported range of values for the XML schema type
      </xdb:errorMsg>
   </xdb:document>
</xdb:errorReport>
```

**Figure 11.18** Sample error report from bulk decomp

## 11.4 Summary

When you consider shredding XML documents into relational tables, remember that XML and relational data are based on fundamentally different data models. Relational tables are flat and unordered collections of rows with strictly typed columns, and each row in a table must have the same structure. One-to-many relationships are expressed by using multiple tables and join relationships between them. In contrast, XML documents tend to have a hierarchical and nested structure that can represent multiple one-to-many relationships in a single document. XML allows elements to be repeated any number of times, and XML Schemas can define hundreds or thousands of optional elements and attributes that may or may not exist in any given document. Due to these differences, shredding XML data to relational tables can be difficult, inefficient, and sometimes prohibitively complex.

If the structure of your XML data is of limited complexity such that it can easily be mapped to relational tables, and if your XML format is unlikely to change over time, then XML shredding can sometimes be useful to feed existing relational applications and reporting software.

DB2 offers two methods for shredding XML data. The first method uses SQL `INSERT` statements with the `XMLTABLE` function. One such `INSERT` statement is required for each target table and multiple statements can be combined in a stored procedure to avoid repetitive parsing of the same XML document. The shredding statements can include XQuery and SQL functions, joins to other tables, or references to DB2 sequences. These features allow for customization and a high degree of flexibility in the shredding process, but require manual coding. The second approach for shredding XML data uses annotations in an XML Schema to define the mapping from XML to relational tables and columns. IBM Data Studio Developer provides a visual interface to create this mapping conveniently with little or no manual coding.

# Index

## Symbols



## A

## B

## C

# D

## E

## F

## J



## K



## L

## M



## N

## O

## P

## Q

## R

## S

## T

## U



## V

## W

## X

## Y–Z