Information Management software

IBM

The Ins and Outs of XML and DB2 for i5/OS

Understand how XML and DB2 work together

Master XML and relational database mapping

Leverage current programming skills to process XML

Jarek Miszczyk
Robert Andrews
David Andruchuk
Hernando Bedoya
Emily Chun

Redbooks

ibm.com/redbooks

International Technical Support Organization

The Ins and Outs of XML and DB2 for i5/OS

October 2006

SG24-7258-00

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (October 2006)

This edition applies to Version 5, Release 4, Modification 0 of IBM i5/OS, product number 5722-SS1.

Contents

Notices . . . vii
Trademarks . . . viii

Preface . . . ix
The team that wrote this redbook . . . ix
Become a published author . . . xi
Comments welcome . . . xii

Part 1. Introduction to DB2 for i5/OS and the scenario . . . 1

Chapter 1. Introduction to XML integration with DB2 for i5/OS . . . 3
1.1 Overview of relevant XML technologies . . . 4
1.1.1 Hierarchical structure of XML documents . . . 4
1.1.2 Document type definition . . . 5
1.1.3 XML schema . . . 6
1.1.4 XPath and Location Path . . . 9
1.1.5 XSL and XSL Transformations . . . 14
1.1.6 Tools for working with XML . . . 14
1.2 Advantages of DB2 . . . 15
1.3 XML to relational database mapping . . . 15
1.3.1 The challenge . . . 16
1.3.2 XML to RDB mapping concepts . . . 17
1.4 Strategies for integrating XML with DB2 for i5/OS . . . 18
1.4.1 Programmatic approach . . . 18
1.4.2 Middleware (XML Extender) approach . . . 19

Chapter 2. Scenario overview . . . 21
2.1 Model of the fictional scenario . . . 22
2.2 Database model . . . 24
2.3 XML document models . . . 27

Part 2. Programmatic approach . . . 31

Chapter 3. Using SQL to compose XML . . . 33
3.1 Overview of stored procedures . . . 34
3.2 Mapping the database to XML logically . . . 35
3.3 Coding the SQL stored procedure . . . 36
3.4 Compiling the SQL stored procedure . . . 38
3.5 Running the SQL stored procedure . . . 40
3.6 Deploying stored procedures . . . 41

Chapter 4. Using XSL Transformation and SQL . . . 43
4.1 Decomposing XML using XSLT, Java, and SQL . . . 44
4.2 Prerequisites . . . 44
4.3 Decomposition planning and design . . . 44
4.4 Decomposition coding . . . 44
4.4.1 Program flow . . . 45
4.4.2 Required modules . . . 46
4.4.3 Input documents . . . 46
4.4.4 Target table . . . 47

4.4.5 XSL transform document 48
4.4.6 XSL document coding 49
4.4.7 Transform processor 51
4.5 Decomposition module deployment 55
4.6 Decomposition execution 55
4.7 Some considerations for XSL decomposition 56

Chapter 5. Using RPG for XML processing 59
5.1 Composing XML using RPG 60
5.1.1 Prerequisites 60
5.1.2 Composition planning and design 61
5.1.3 Composition coding 61
5.1.4 Deployment of RPG composition modules 72
5.1.5 Composition execution 73
5.1.6 Considerations for RPG composition 74
5.2 Decomposing XML using RPG 75
5.2.1 Prerequisites 76
5.2.2 Decomposition planning and design 76
5.2.3 Decomposition coding 76
5.2.4 Deployment of the RPG decomposition modules 85
5.2.5 Decomposition execution 86
5.2.6 Considerations for RPG decomposition 87

Chapter 6. Using SAX and Java to decompose XML 89
6.1 Setting up the development environment 90
6.2 Defining the XML to RDB mapping 92
6.3 Implementing the mapping in Java 94
6.4 Deploying the solution to the System i machine 101
6.5 Running the deployed application 103

Chapter 7. Advantages and disadvantages of the programmatic approach 105
7.1 Advantages 106
7.2 Disadvantages 106

Part 3. Middleware approach 107

Chapter 8. Overview of DB2 XML Extender 109
8.1 Product overview 110
8.2 Setup and common administration tasks 110
8.2.1 Prerequisites 111
8.2.2 Preparing your database for XML Extender 111
8.3 XML Extender storage and access methods 114
8.3.1 XML Column method 116
8.3.2 XML Collection method 127
8.4 SQL mapping DAD composition example 129
8.4.1 SQL mapping DAD methodology 129
8.4.2 Building the SQL view 131
8.4.3 Creating the SQL mapping DAD 134
8.4.4 Executing the SQL mapping DAD composition 137
8.5 Handling very large documents 139
8.5.1 Redefining user-defined types 139
8.5.2 Redefining stored procedure parameters 140
8.5.3 Application program and system changes 142
8.6 Additional considerations 143

8.6.1 SQL naming convention 143
8.6.2 Migration 143
8.6.3 Commitment control 144
8.6.4 Journal management 144
8.6.5 Save and restore. 144
8.6.6 Troubleshooting and error messages 145

Chapter 9. Performance-related information regarding XML Extender 147
9.1 Operating system levels and enhancements 148
9.1.1 XML parser performance increase: V5R3 and V5R4 PTF 148
9.1.2 XML Extender performance increase: V5R3 PTF. 148
9.1.3 XML Extender Quick Pool memory management: V5R4 PTF 148
9.1.4 Technote 148
9.1.5 PTF quick reference 149
9.2 XML Extender performance measurements 149
9.3 XSLT and XML Extender compared to XSLT and SQL 151
9.4 Performance summary 153

Chapter 10. Shredding methodology 155
10.1 Setting up the development environment 156
10.2 Analyzing the XML document structure and determining the mapping. 157
10.3 XSL Transform to insert unique keys for repeating elements 161
10.4 Modifying DTD and XSD to reflect the transformed document 166
10.5 Building the DDL script and adding RI constraints for the keys 167
10.6 Creating tables on the database server 169
10.7 Graphically mapping transformed DTD to relational tables (generating DAD) 170
10.8 Creating the wrapper stored procedure 176
10.9 Deploying the XSL, DAD, Java, and SQL to the integrated file system on the System i machine 177
10.10 Running the deployed application. 179

Chapter 11. Composing methodology 181
11.1 Setting up the development environment 182
11.2 Analyzing the database structure and outbound XML hierarchy. 183
11.3 Creating the SQL view to project the data 186
11.4 Graphically mapping the relational database to XML (generating DAD). 186
11.5 Creating the wrapper stored procedure 192
11.6 Deploying DAD, SQL script to the integrated file system on the System i machine . 193
11.7 Running the deployed application. 194

Chapter 12. Advantages and disadvantages of the middleware approach 197
12.1 Advantages. 198
12.2 Disadvantages 198

Part 4. Moving forward 199

Chapter 13. A look into the future 201
13.1 Future XML technologies 202
13.1.1 XQuery 202
13.1.2 SQL/XML 203
13.1.3 Hybrid and multistructured database management systems 203
13.1.4 XLink. 204
13.1.5 XML Encryption. 204
13.2 Future XML technologies on the System i platform 205

Appendix A. Additional material 207
Locating the Web material 207
Using the Web material 207
System requirements for downloading the Web material 209
How to use the Web material 209

Related publications 211
IBM Redbooks 211
Other publications 211
Online resources 211
How to get IBM Redbooks 211
Help from IBM 212

Index 213

Notices

Trademarks

Preface

XML represents a fundamental change in computing. It allows applications to move away from proprietary file and data formats to a world of open data interchange. XML has become ubiquitous not only because of its range of applications, but also because of its ease of use.

Although XML solves many problems by providing a standard format for data interchange, some challenges remain. In the real world, applications need reliable services to store, retrieve, and manipulate data. These services have traditionally been offered by DB2 Universal Database for iSeries, which is now called DB2® for i5/OS®.

In this IBM® Redbook, we discuss the challenges of representing XML hierarchies in the relational database model. We provide an in-depth explanation of the three most popular approaches to bridge the hierarchy, the relational model dichotomy:

- Programmatically process the XML documents and map their hierarchy into a relational database.

  Typically these solutions leverage one of the standard application programming interfaces (APIs) to manipulate XML, such as Simple API for XML (SAX) or Document Object Model (DOM). Java™ Database Connectivity (JDBC™) is then used to write the extracted data into the database.
- Use database middleware to handle the XML parsing and XML-to-relational database mapping.

  DB2 for i5/OS provides DB2 XML Extender. We focus on the advanced features of XML Extender that allow us to handle complex or large XML documents with little or no programming effort. We demonstrate how to take advantage of XML Extender and WebSphere® Studio Application Developer tooling to dramatically speed the process of building robust XML-based applications.
- Use Extensible Stylesheet Language (XSL) Transformation to transform inbound XML documents directly to SQL scripts.

  The use of XSL is useful to decompose an XML document into an SQL Script to populate database tables.

We also share best practices and techniques aimed at streamlining the XML and DB2 for i5/OS integration.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Jarek Miszczyk is a Senior Software Engineer for the ISV Enablement organization in Rochester, MN. He has been with IBM for 15 years. His mission is to provide hands-on consulting services to independent software vendors (ISVs), large IBM Clients, and other IBM organizations on DB2 for i5/OS-related issues. He also writes extensively and teaches IBM classes in all areas of the IBM System i™ database. Before joining the ISV Enablement organization in 2000, Jarek worked for three years at the IBM ITSO in Rochester, where he was the leading author of several popular IBM Redbooks™ covering the topic of database. Jarek holds a Master in Computer Science degree and has over 18 years of experience in the computer field. His areas of expertise include cross-platform database programming, database performance tuning, and database integration with emerging technologies (XML, Java, and Microsoft® .NET).

Robert Andrews is a Staff Software Engineer working on the Database team in the Rochester Support Center. While his focus is database, he specializes in SQL, journaling, and remote communications. He has been a member of, and is the education coordinator for, the Database team since he joined IBM five years ago. Robert also worked for eight months with the DB2 Development team focusing on performance in the initial release of SQL Query Engine (SQE) in R520. Robert has contributed to several publications and published articles in trade magazines. Robert is currently finishing a Master in Information Systems and Management degree.

David Andruchuk is a System Architect with SunGard Futures Systems in Chicago, Illinois. He has over 20 years of application development and architecture experience in all areas of business and the IBM Midrange platform. Currently he is working on XML integration implementation for SunGard and their installed client base worldwide.

Hernando Bedoya is an IT Specialist at the IBM ITSO, in Rochester, Minnesota. He writes extensively and teaches IBM classes worldwide in all areas of DB2 for i5/OS. Before joining the ITSO more than five years ago, he worked for IBM Colombia as an AS/400® IT Specialist doing presales support for the Andean countries. He has 20 years experience in the computing field and has taught database classes in Colombian universities. He holds a Master in Computer Science degree from EAFIT, Colombia. His areas of expertise are database technology, application development, and data warehousing.

Emily Chun is an Advisory Software Engineer working on the XML Extender development team at the IBM Silicon Valley Lab. She has been with the XML Extender product since it was first released for V5R1 on the IBM eServer™ iSeries™ server. She has more than 20 years of experience with software and application development within IBM on mainframe, midrange, and workstation platforms. She has a Bachelor of Arts degree in Computer Science and a Master in Business Administration (Management Information Systems) degree.

The XML and DB2 team: David Andruchuk, Hernando Bedoya, Emily Chun, Robert Andrews, and Jarek Miszczyk

Thanks to the following people for their contributions to this project:

Thomas Gray
Joanna Pohl-Miszczyk
Linda Robinson
ITSO, Rochester Center

Aaron Albertson
Jay Hansen
Scott Hanson
Linda Swan
Jon Triebenbach
Jing Wang
Eric Will
IBM Rochester Development Lab

Kent Milligan
DB2 for i5/OS Technology team IBM Rochester

Jennifer Haugen
David Summers
IBM Rochester Support Center

Felicity Hendricks
Angel Reyda
IBM Silicon Valley Development Lab

Debra Yendrzeski

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Part 1

Introduction to DB2 for i5/OS and the scenario

In this part, we introduce basic information about DB2 for i5/OS and integration to XML technology. We explain the different strategies for integrating XML with DB2 for i5/OS. We also describe the different scenarios that will be used in this book.

This part of the book includes the following chapters:

- Chapter 1, “Introduction to XML integration with DB2 for i5/OS” on page 3
- Chapter 2, “Scenario overview” on page 21

1

Introduction to XML integration with DB2 for i5/OS

XML represents a fundamental change in computing. It allows applications to move away from proprietary file and data formats to a world of open data interchange. XML has become ubiquitous not only because of its range of applications, but also because of its ease of use. The text-based nature of XML makes creating tools easy. Also since it is an open, license-free, cross-platform standard, anyone can create, develop, and use tools for XML. In short, XML means portable data.

In this IBM Redbook, we discuss all aspects of XML integration with DB2 for i5/OS. By presenting a simple retail store scenario, we discuss a broad range of technologies that can be employed to manipulate XML documents so that they can be stored in or retrieved from DB2. We also document end-to-end implementation methodologies that were hardened in production environments. Our goal is to make this redbook a reference publication for both traditional System i developers who use CL, RPG, and native APIs and those who prefer Java and graphical user interface (GUI)-based tooling.

In this chapter, we introduce you to the related XML technologies, the advantages of DB2, XML to relational data mapping, and strategies for integrating XML with DB2 for i5/OS.

1.1 Overview of relevant XML technologies

Before we explain the details of the XML integration with DB2 for i5/OS, we must discuss several basic concepts used throughout this book.

1.1.1 Hierarchical structure of XML documents

Figure 1-1 shows a graphical representation of a sample XML document called StoreSales.xml. As you can see in Figure 1-1, an XML document has a tree-like structure, with the *root element* StoreSales at the top of the tree. The document must contain one and only one root element. An element is the *parent* of all the elements it contains. The elements that are inside a parent element are called its *children*. Similarly, the elements that have the same parent element are called *siblings*.

In our example, StoreSales is the parent of all other elements, Address is a child of StoreSales, and Sales and Returns are siblings. Going down the element tree, each child element must be fully contained within its parent element. Sibling elements may not overlap.

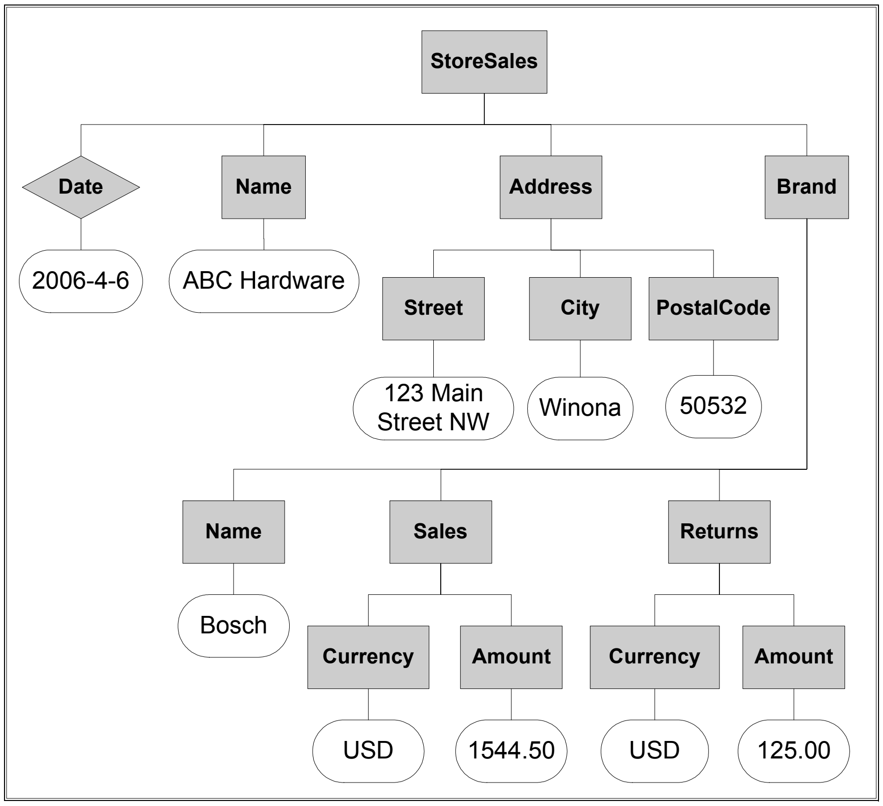


Figure 1-1 Tree-like hierarchy of an XML document

For your reference, Example 1-1 shows the content of the sample StoreSales.xml document.

Example 1-1 Source of StoreSales.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<StoreSales Date="2006-04-06">
   <Name>ABC Hardware</Name>
   <Address>
      <Street>123 Main Street NW</Street>
      <City>Winona</City>
      <PostalCode>50532</PostalCode>
   </Address>
   <Brand>
      <Name>Bosch</Name>
      <Sales>
         <Currency>USD</Currency>
         <Amount>1544.50</Amount>
      </Sales>
      <Returns>
         <Currency>USD</Currency>
         <Amount>125.00</Amount>
      </Returns>
   </Brand>
</StoreSales>
```

1.1.2 Document type definition

The document type definition (DTD) specifies the structure of an XML document, thereby allowing XML parsers to understand and interpret the document's contents. The DTD contains the list of tags that are allowed within the XML document and their types and attributes. More specifically, the DTD defines how elements relate to one another within the document's tree structure and specifies which attributes may be used with which elements. Therefore, the DTD also constraints the element types that can be included in the document and determines its conformance. When an XML document conforms to its DTD, it is said to be *valid*.

Example 1-2 shows the content of the StoreSales.dtd that defines the grammar for our sample StoreSales.xml document.

Example 1-2 Source of StoreSales.dtd

```
<?xml version='1.0' encoding="UTF-8"?>
<!ELEMENT StoreSales (Name,Address,Brand+)>
<!ATTLIST StoreSales Date CDATA #REQUIRED>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Address (Street,City,PostalCode)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT PostalCode (#PCDATA)>
<!ELEMENT Brand (Name,Sales?,Returns?)>
<!ELEMENT Sales (Currency,Amount)>
<!ELEMENT Returns (Currency,Amount)>
<!ELEMENT Currency (#PCDATA)>
<!ELEMENT Amount (#PCDATA)>
```

We must point out some things in the DTD shown in Example 1-2. As you can see, a DTD tag that defines an XML element starts with `<!ELEMENT`. The tag name is followed by the name of the element that is defined. If the element is a parent for other elements, the children elements are listed inside parentheses after the parent element's name.

You may have noticed that some children in the list may end with an additional qualifier. The question mark (?) qualifier means that the given element is optional. A parent element contains zero or one occurrences of a given child element. An asterisk (*) means zero or more occurrences, while a plus sign (+) indicates one or more occurrences of a child element.

Similarly, a DTD tag that defines an XML attribute starts with `<!ATTLIST` and is followed by the element name for which the attribute is defined. Each attribute is described by three, space-separated values: the name of the attribute, the type of data, and the attribute's default value.

1.1.3 XML schema

The purpose of the XML schema is to define the legal building blocks of an XML document, similar to DTD. While DTD may be easier to work with, the XML schema has several important advantages over DTD. One advantage is that the XML schema supports all data types that are used in most programming languages. The most common types are:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

In addition, the XML schema element complexType allows you to define an element type that can consist of subelements. Example 1-3 shows a complex data type.

Example 1-3 Sample XML schema with complexType

```
<xs:element name="StoreAddress" Type="Address"/>
   <xs:complexType name="Address">
      <xs:sequence>
         <xs:element name="Street" type="xs:string"/>
         <xs:element name="City" type="xs:string"/>
         <xs:element name="State" type="xs:string"/>
         <xs:element name="Zip" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Notice how the complex type Address is used in the definition of the StoreAddress element.

Another advantage is that the XML schema provides better support for XML namespaces. The validated document can use multiple namespaces and reuse constructs that are already defined in a different namespace.

With the XML schema, the built-in data types can be modified to set constraints or data content restrictions. Restrictions on XML elements are called *facets*. Example 1-4 shows how the base data type xsd:string can be restricted by a regular expression.

Example 1-4 Sample XML schema with restricted data types

```
<xs:element name="pwd">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="[a-zA-Z]{6}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

The `pwd` element is a simple type with a restriction. There must be exactly six characters in a row, and those characters must be in lowercase or uppercase letters.

Armed with the basic XML schema skills, we can review the contents of the StoreSales.xsd file shown in Example 1-5. This XML schema document can be used to validate the example StoreSales.xml from Example 1-1 on page 5.

Example 1-5 Source of StoreSales.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="StoreSales">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="Name"/>
            <xsd:element ref="Address"/>
            <xsd:element maxOccurs="unbounded" minOccurs="1" ref="Brand"/>
         </xsd:sequence>
         <xsd:attribute name="Date" type="xsd:string" use="required"/>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="Name" type="xsd:string"/>
   <xsd:element name="Address">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="Street"/>
            <xsd:element ref="City"/>
            <xsd:element ref="PostalCode"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="Street" type="xsd:string"/>
   <xsd:element name="City" type="xsd:string"/>
   <xsd:element name="PostalCode" type="xsd:string"/>
   <xsd:element name="Brand">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="Name"/>
            <xsd:element ref="Sales"/>
            <xsd:element maxOccurs="1" minOccurs="0" ref="Returns"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="Sales">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="Currency"/>
            <xsd:element ref="Amount"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="Returns">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="Currency"/>
            <xsd:element ref="Amount"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="Currency" type="xsd:string"/>
   <xsd:element name="Amount" type="xsd:string"/>
</xsd:schema>
```

In addition to showing the XML Schema Definition (XSD) as an XML document itself, several tools can help to visually display what is defined inside the XSD. Using these tools can produce graphics such as the one shown in Figure 1-2.

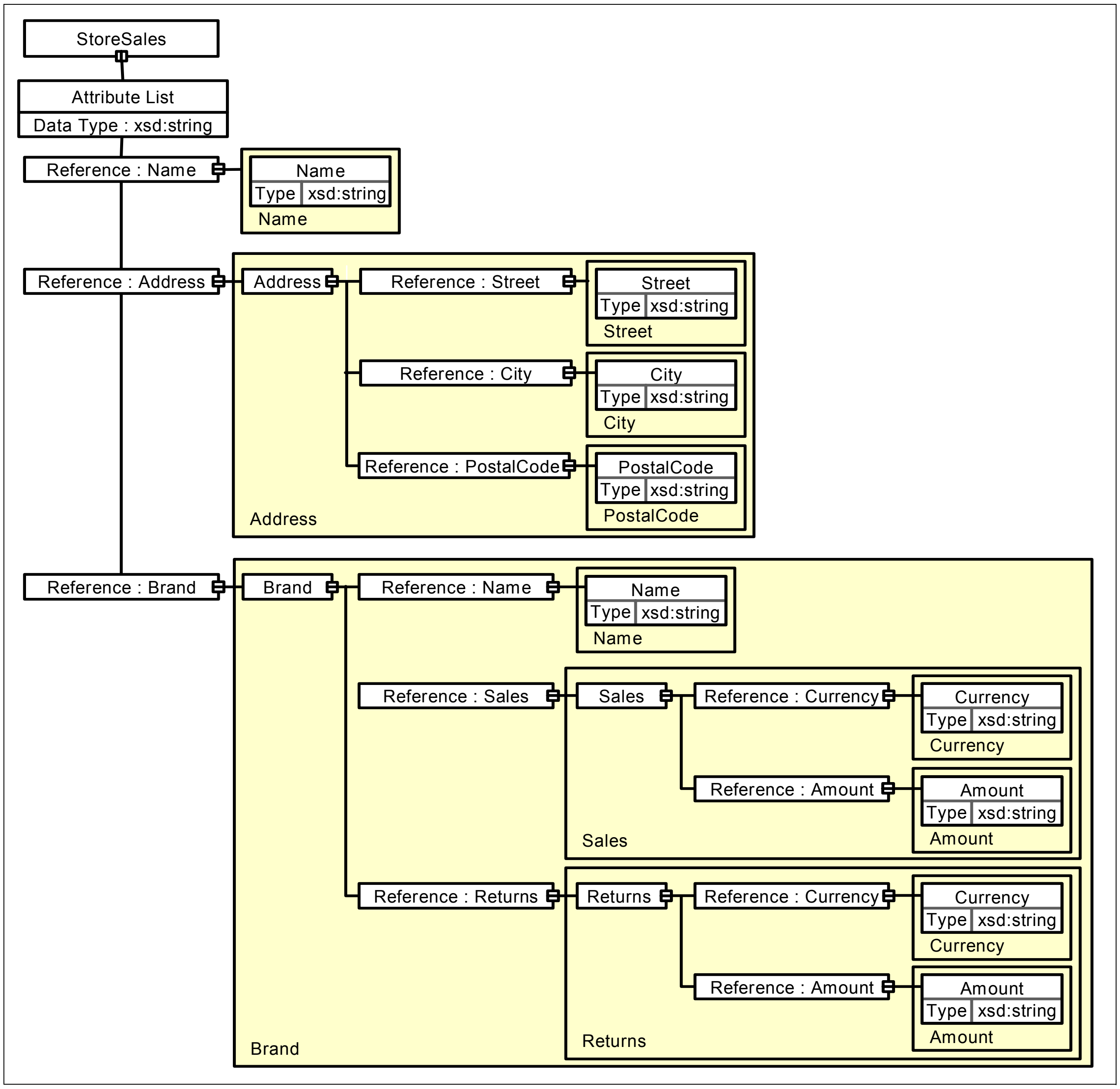


Figure 1-2 Sample graphical representation of XSD

1.1.4 XPath and Location Path

XPath is one of the corner stones of the World Wide Web Consortium (W3C) Extensible Stylesheet Language Transformation (XSLT) standard. It is a language that describes how to locate specific elements (attributes, comments, processing instructions, and so on) in an XML document. XPath allows you to locate specific content within an XML document.

An XPath expression is used to navigate the XML hierarchy and return parts of the XML documents. Generally, an XPath expression can return one of the following data types:

- Node set
- Boolean
- Number
- String

In most cases, we use XPath expressions to return node sets. A node set can be empty or contain single node or many nodes. There are several kinds of nodes:

- Element
- Attribute
- Text
- Namespace
- Processing-instruction
- Comment
- Document (root)

XPath treats an XML document as a logical ordered tree of nodes. The root of the tree is called the *document node* (or root node). Do not confuse the XPath root node with XML root element. In XPath, the root node is represented by a forward slash (/).

An XPath expression contains one or more location steps, separated by slashes. Each location step has the following form:

```
axis-name::node-test[predicate]
```

The XPath *axis* contains a part of the document, defined from the perspective of the "context node". The XPath specification defines a number of different axes:

- **ancestor**: Holds all ancestors of the current node
- **ancestor-or-self**: Holds all ancestors of the current node and the current node itself
- **attribute**: Holds all attributes of the current node
- **child**: Holds all children of the current node
- **descendant**: Holds all descendants of the current node
- **descendant-or-self**: Holds all descendants of the current node and the current node itself
- **following**: Holds everything in the document after the closing tag of the current node
- **following-sibling**: Holds all siblings after the current node
- **namespace**: Holds all namespace nodes of the current node
- **parent**: Holds the parent of the current node
- **preceding**: Holds everything in the document that is before the start tag of the current node
- **preceding-sibling**: Holds all siblings before the current node
- **self**: Selects the current node

The *node test* makes a selection from the nodes on that axis. The two most useful node tests are:

- **node()**: Selects any type of node
- **text():** Selects a text node

By adding *predicates*, it is possible to select a subset from these nodes. A predicate contains an XPath expression. If the expression in the predicate returns true, the node remains in the selected set; otherwise it is removed.

XPath expressions are extensively used in many scenarios and examples that are discussed in this redbook. We wrote a simple Java utility that can be helpful in testing the correctness of XPath expressions. Example 1-6 shows the source listing of our XPathTester program.

Example 1-6 XPathTester.java

```
import java.io.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.xpath.*;
import org.w3c.dom.NodeList;

public class XPathTester {
   private Document document;

   private String query;

   public XPathTester(String filename) {
      try {
         InputSource inputSource = new InputSource(filename); 1
         BufferedReader in = new BufferedReader(new InputStreamReader(
               System.in));

         do {
            System.out.print("XPath> ");
            query = in.readLine(); 2
            try {
               XPath xpath = XPathFactory.newInstance().newXPath(); 3

               NodeList nodes = (NodeList) xpath.evaluate(query,
                     inputSource, XPathConstants.NODESET); 4

               if (nodes == null) {
                  System.out
                        .println("XPath query must return a node-set");
               } else {
                  for (int i = 0; i < nodes.getLength(); i++) { 5
                     System.out.println("Name:  "
                           + nodes.item(i).getNodeName());
                     System.out.println("Value: "
                           + nodes.item(i).getNodeValue());
                  }
               }
            } catch (XPathExpressionException e) {
               System.out.println("Invalid XPath query: " + query);
            }
         } while (!(query.equals("quit")));
      } catch (IOException e) {
         System.err.println("Could not load '" + filename + "': "
               + e.toString());
      }
   }
   public static void main(String[] args) {
      if (args.length == 1)
         new XPathTester(args[0]);
      else
         System.err.println("Usage: java XPathTester <filename>");
   }

} // end of class XPathTester
```

The program requires one input parameter, which is the name of the XML document to be searched by the XPath expression. In line **1**, we use the org.xml.sax.InputSource class to encapsulate information about the input XML document. This information includes a public identifier, a system identifier, a byte stream, or a character stream. In line **2**, the user provides the XPath expression to be tested.

In line **3**, the XPathFactory is used create an XPath object. In line **4**, the XPath class is used to evaluate the XPath expression entered by the user. In line **5**, the resulting node set is displayed on the standard output device.

Important: The XPath interface and the XPathFactory abstract class were first defined in Java 2 Platform Standard Edition (J2SE™) 5.0. Therefore, you need Java SDK 1.5 or later to compile the XPathTester program.

We use the XPathTester to evaluate several XPath expressions. We use the sample StoreSales.xml document as input. The following example shows the input XPath expressions and the results produced by the program:

```
XPath> /
Name:  #document
Value: null
```

As mentioned the forward slash selects the root node of a document:

```
XPath> /StoreSales/Name
Name:  Name
Value: null
```

In this case, we select Name element, which is a child of the StoreSales element. You may be surprised that the value for that element is null. In this case, in XPath, the text contained within an element is a child of the containing element. The type of this child node is text; see the following example:

```
XPath> /StoreSales/Name/text()
Name:  #text
Value: ABC Hardware
```

Now we test less trivial expressions. We want to retrieve the dollar amounts that are descendants of Brand:

```
XPath> //Brand/descendant::*/Amount/text()
Name:  #text
Value: 1544.50
Name:  #text
Value: 125.00
```

We retrieve the dollar amount only for these Amount elements that are children of Sales that in turn are descendants of Brand:

```
XPath> //Brand/descendant::*[name() = 'Sales']/Amount/text()
Name:  #text
Value: 1544.50
```

Location Path

A *Location Path* is a special case of XPath expression. It is used by many middleware products including DB2 XML Extender. A Location Path is a sequence of XML tags separated by a forward slash that identifies an XML element or attribute.

Location Path uses the following syntax types:

/tag1	Represents the element tag1under root
/tag1/tag2/..../tagn	Represents an element with the name tagn as the child of the descending chain from root, tag1, tag2, through tagn-1
//tagn	Represents any element with the name tagn, where double slashes (//) denote zero or more arbitrary tags
/tag1//tagn	Represents any element with the name tagn, a child of an element with the name tag1under root, where double slashes denote zero or more arbitrary tags
/tag1/tag2/@attr1	Represents the attribute attr1 of an element with the name tag2, which is a child of element tag1under root
/tag1/tag2[@attr1="5"]	Represents an element with the name tag2 whose attribute attr1 has the value 5; tag2is a child of element with the name tag1under root
/tag1/tag2[@attr1="5"]/.../tagn	Represents an element with the name tagn, which is a child of the descending chain from root, tag1, tag2, through tagn-1, where the attribute attr1 of tag2 has the value 5

Figure 1-3 illustrates how to navigate from the root element StoreSales to the nested child element Amount following the XML document's hierarchy. The resulting Location Path is /StoreSales/Brand/Returns/Amount.

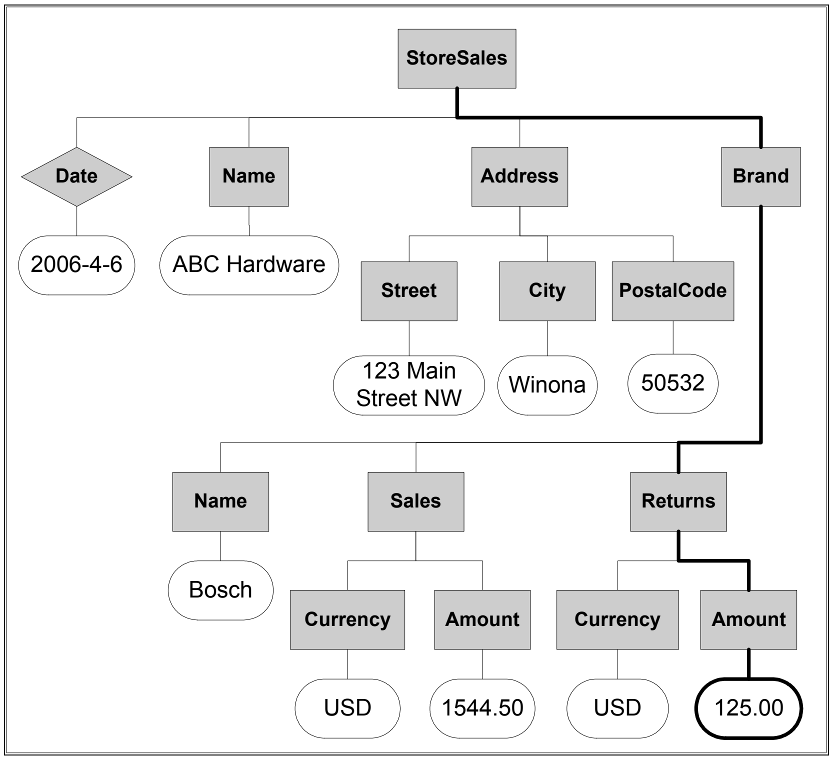


Figure 1-3 Location Path

1.1.5 XSL and XSL Transformations

The concept of stylesheets in most widely known in the Web usage of cascading style sheets (CSS). A stylesheet provides formatting and presentation processing for raw underlying data. This means that XML, in its raw, tagged text data format, is a perfect candidate to take advantage of stylesheets.

XSL is a standard for stylesheets that can be applied to XML documents. The XSL family contains three different types of statements. The first category is *XML Path Language* (XPath). Similar to how Location Path works, XPath defines a method for locating elements inside an XML document. The second category is *XSL Formatting Object* (XML-FO) statements. XSL-FO provides methods of visually formatting the data inside an XML document.

The third and most useful category is *XSL Transformations* statements. XSLT is a language that allows for powerful manipulation of the data inside an XML document. Where XSL-FO provides formatting to the screen, XSLT can change the contents of the elements and more. It is important to note that XSLT does not change the original XML document. It simply outputs the transformed copy to a new file. This new file can be an XML document or any other type of file such as an HTML document or PDF file. Later in this book, we show how these transformations can be useful in connecting XML and DB2.

1.1.6 Tools for working with XML

Several IBM and OEM tools exist for working with XML and other related technologies. Since this is not the main focus of this redbook, we do not go into details here. However, you can find a discussion about XML tools in Part 3, “XML tools for database systems,” of the IBM Redbook *XML for DB2 Information Integration*, SG24-6994.

1.2 Advantages of DB2

When you begin looking at how to work with XML in your company, many questions arise. The most obvious of these is: How do we handle XML documents?

Specific database products can store the XML documents in a hierarchical model. At first glance, some might say that this should be the direction taken. However, many of these products are new and not yet proven. Alternatively, DB2 for i5/OS is a mature technology that has been in development for 20 years. After reviewing the advantages of DB2 for i5/OS, we are sure that you will agree that DB2 is the best choice for a back-end system to handle XML documents.

Since DB2 has been around for several years, it is field proven. There is a wealth of query optimization code that allows for data processing in a fast and efficient method. Such tools as multiple formats of indexes and internal statistics allow the system to make the best choice possible based on the request, the data, and the system resources. The System i platform is an extremely stable one that requires little downtime. Some companies measure their time between initial program loads (IPLs) in months. If a system should fail, the System i platform has many options to ensure high availability. These include cross geographic mirroring and several IBM and OEM applications that provide data replication.

DB2 for i5/OS incorporates many technologies to take full advantage of the hardware platform. These include memory management, single-level storage, and multiple processors. Adding the DB2 Symmetric Multiprocessing program provides functionality for using multiple processors to handle a single operation. As new hardware is added, it is instantly taken advantage of with no application changes.

The System i platform is known for being easy to use. All storage management tasks, such as page sizes and proper data distribution, are handled automatically by the operating system and DB2. Closely integrated in the operating system is the concept of *object-level security*, which allows the administrator to set the security once and have it apply to all interfaces to that file. The System i platform provides a complex security model including user, group, and public levels of security, as well as granular detail down to field or column level security in tables.

Since the System i platform is designed to handle multiple processes that access the same data, DB2 for i5/OS provides strong and varied locking mechanisms. This allows multiple users to access different parts of the file with different levels of locks on the data. Data can be locked simply when it is changed, or it can be reserved so that no other processes can read the data while another one is working on it. The key is that there is a high degree of flexibility to allow your process to provide the level of concurrent access that is acceptable in your environment.

While the list of System i strengths can comprise a book of its own, our goal here is to point out some the major database related advantages of DB2. We are confident that using DB2 as the backend for XML documents is a good choice. We want you to understand that DB2 and XML can work together and give you all the benefits that you have come to expect from DB2 for i5/OS when working with XML documents.

1.3 XML to relational database mapping

Although XML solves many problems by providing a standard format for data interchange, some challenges remain. In the real world, applications need reliable services to store, retrieve, and manipulate data.

On the System i platform, these services were traditionally offered by DB2 for i5/OS. It seems natural to use DB2 for i5/OS to persist and manipulate XML documents. The problem is that the relational model of DB2 and the hierarchical model of XML documents are different.

In the relational model, the data is stored in rows of two-dimensional tables where the physical order of rows is insignificant. XML is a highly hierarchical model where the order of elements is significant, and the relationship among elements is described by a given document. Using a relational model to express the hierarchy of elements in a complex XML document is a non-trivial task.

1.3.1 The challenge

As stated previously, two distinct methods of data models are used. These methods are the relational data model and the hierarchical data model. The challenge introduced here is how to make these two interchange data back and forth. How can data be taken from one data model and transposed into the other? That is the focus of this redbook. First, we look more closely at these models.

Relational data model

Note: For those of you who have been using the System i platform for a while, this is a review.

In the relational data model, data is stored in flat, two-dimensional tables. We create these tables as having columns or fields that label and define the data breakdown. Think of these as the header row of a spreadsheet. Into the columns, rows or records of data are inserted. This is the actual data that we must store.

However, we often run into situations where the data does not fit into these two-dimensional tables. For example, assume that we have a database of contact information. It is easy to start defining such items as address, city, and phone number. But say we also want to store the names of the children in the family. Each family can have none, one, or more children.

We can approach this problem in many ways. The first way is to create one field for children and place the names of all the children in that one field. However, there is no easy way to obtain more information about each child. Assume that we also want to record the age of each child. The result is a mess of data that is difficult to understand.

Another approach is to create multiple fields, such as child1, child2, and so on. But how many do you put in? What if there are more children than slots? What if they have no children? We would have wasted space.

To work around this problem, the relational model introduces the concept of creating multiple tables and then defining a relationship between them, hence the name *relational data model*. In our main table, if we assign a unique identifier, a primary key, that allows us to select each particular row and then put that value along with the child’s name into a second table, we can use that key value to put the pieces of information together. This eliminates the problem of how many fields to put in since we continue to add rows with only columns of the family ID and the child’s name. This would easily allow us to add columns to the child table for items like age or birth date. If there are no children, then no rows are added to the child table. Since the primary key from the parent file is used in the child file, it is referred to as a *foreign key*. This means that the value is not unique, since we could have multiple children from the same family, but links back to a unique row in the family table. Because the previous example models what is happening so well, the terms *parent* and *children* are used to describe any tables that match this dependency.

Taking our analogy one step further, we cannot have children without a parent. How can we assign a child to a family with a unique family key if that family is not defined in the parent table? To help enforce this, the concept of *referential integrity* (RI) is introduced. RI is the way that the database management software prevents a child from being added unless it can find and verify the foreign key, the family identifier, in the parent table.

A real-world example of RI can be thought of in terms of shipping addresses. If we view a company as only shipping to eight states, it can create a table with those values. Each has a unique identifier. The shipping table that stores the address of where the item is to be shipped has the ship-to state as a foreign key to the valid state table. That way, if an order is attempted to be shipped to Pennsylvania and that state is not in the states table, the order is not allowed.

These relationships can be nested at many levels to provide a cascading effect where one master, parent table refers to many children and those in turn are parents to other children tables. By matching the primary and foreign keys, all the data can be related to one another. Because we add these layers of links and remove repetitive data, we are said to be normalizing the database.

Hierarchical data model

In addition to the relational data model, the other major representation of data is the hierarchical model. This model closely matches the structure of an XML document. The best way to think of data represented in a hierarchical model is to think of a collection of upside down trees. The top element represents the highest level for that chunk of data.

Using the family concept explained in “Relational data model” on page 16, we can have one tree for each family. At the top of the tree, we store all our pieces of information for that family such as the address, city, and postal code. In addition to the data that hangs off of the top level, lower levels are added. We can have a model so that each child in the family is placed on a layer one down from the top. Under each child, we can then attach the information that relates to that child.

The hierarchical model is concerned with the physical relationship of the data. The children elements are often duplicated under each parent element. The children entries have no key structure like the relational model does to link them to their parents. They simply exist directly under them.

A common example of this model is a purchase order. In a classic order example, an order header contains information at the order level. In addition to the order header, there are several order items with their own pieces of information. In the hierarchical model, the main information is at the highest level, while each item that is ordered is a child of the parent. In the relational model, we need a type of order key to link the order items to the order header information.

One drawback of the hierarchical model is that there cannot be links horizontally; rather there can only be vertical links. Each tree can only connect up and not across. We can think in the terms of a “one to many” relationship going down the tree. A child cannot have two parents.

1.3.2 XML to RDB mapping concepts

Now that you understand the two different methodologies for storing data, how can you bring the two together so that data can be transformed from one format to the other? The key is to develop a mapping between the two different models so we can see at a high level what needs to be done to move from one to the other. We call this broad topic *XML (hierarchical) to RDB (relational) mapping*. To begin, you must to understand the basic concepts.

Containment

Containment is the XML equivalent of adding layers. When an element is a child to a parent, all children and siblings are contained inside the parent element. We can say that all elements, except the root element itself, are contained inside the root element.

Optional elements

When looking at how an XML document should look, we must go to the DTD or XSD. These definitions show all the possible valid items and structure. Some of the elements in the definitions may be optional. When using the family database model indicated earlier, keep in mind that a family is not required to have children.

Repeating elements

With a family database, not only is the existence of children optional, but they can repeat. There can be multiple children in one family.

Wrapper elements

Wrappers are used to help group elements of the same type together. Wrapper elements do not contain any text nodes or attributes. They only hold other children elements. It is possible to create a "children" element that only contains "child" elements for each child. If there is an attribute, such as the number of children in the children element, then the element is not a wrapper element.

Same named elements

Inside an XML document, the same element name can be used in many places. The XSD allows us to define an element with a data type and reference that element name in several places. In the example in 1.1, "Overview of relevant XML technologies" on page 4, an element called "Name" is used inside both the root StoreSales element and the Brand element. The element "Name" is only defined once as being of data type string.

You cannot have an element at another location that has a different data type with the same name. However, same named elements are powerful for that reason. After the Name element is defined, it is a fairly generic element that we can use as a children element to other items, as we have done in the example.

1.4 Strategies for integrating XML with DB2 for i5/OS

You have seen that DB2 for i5/OS is extremely powerful. However, it is based on a relational model that does not match up well to the hierarchical model of XML. To allow these two technologies to work together, we can use the concept of XML to RDB mapping. To implement this methodology on the System i platform, there are two approaches.

1.4.1 Programmatic approach

The first approach to integrating XML into DB2 for i5/OS is the *programmatic approach*. In this methodology, the application development team uses traditional programming methods in languages such as RPG, CL, and SQL to do the integration. Standard Simple API for XML (SAX) and Document Object Model (DOM) parsers are available to assist.

The advantage of the programmatic approach is that programs can be created and implemented quickly. However, this also leads to the primary problem. Since it is quick, it often lacks reusability and flexibility. We often see developers creating unique programs with the business logic for each XML document that comes in or goes out.

For complete details about the advantages and disadvantages of the programmatic approach, see Chapter 7, “Advantages and disadvantages of the programmatic approach” on page 105.

1.4.2 Middleware (XML Extender) approach

In addition to the programmatic approach, middleware applications exist to help bridge the gap between programs and processes for you. The major middleware product for the System i platform is *DB2 UDB XML Extender for iSeries*. This product helps programs and processes work more easily with XML documents. It takes care of the complex logic that goes into the underlying handling of XML documents so that you can focus better on your business.

Because of this flexibility, the XML Extenders product is powerful. It allows for document mapping that can be used in both composing and decomposing without modification. This flexibility and power add another layer that may take additional time to fully use.

For complete details about the advantages and disadvantages of the middleware approach, see Chapter 12, “Advantages and disadvantages of the middleware approach” on page 197.

2

Scenario overview

In this chapter, we describe the scenario that we used throughout this redbook to provide examples and a basis for processing information. Specifically, we discuss the following topics:

- Overview of the scenario
- Database architecture
- XML architecture

2.1 Model of the fictional scenario

To provide a consistent flow for this book, our team developed a fictional scenario that mimics many real-world scenarios that you may encounter. This model contains many complex concepts but is illustrated with a limited set of data. The key is to understand the structure of our fictional company and the concepts that we are discussing. When you understand our structure, you can draw similarities to your own environment.

Our fictional company, A^{2}BCM Corp, is a retail shop. The data that we provide is based on a hardware or home improvement store, but the structure is universal to any retail environment. The company structure is multinational with a set of regions in each country. Each region within a country has several stores.

The purchasing department is looking to leverage volume discounts from various brand distributors. To do this, they need hard data for daily sales by country and by brand of product. Our data model is one where each store has individual transactions that need to be rolled up (summarized) to the country level. The country data is then rolled up to the corporate level, and at that level, the data is prepared. Finally the data is presented to purchasing with only the information that was requested by the purchasing department, which is a breakdown by sales per country per brand. Figure 2-1 shows an overview of this flow.

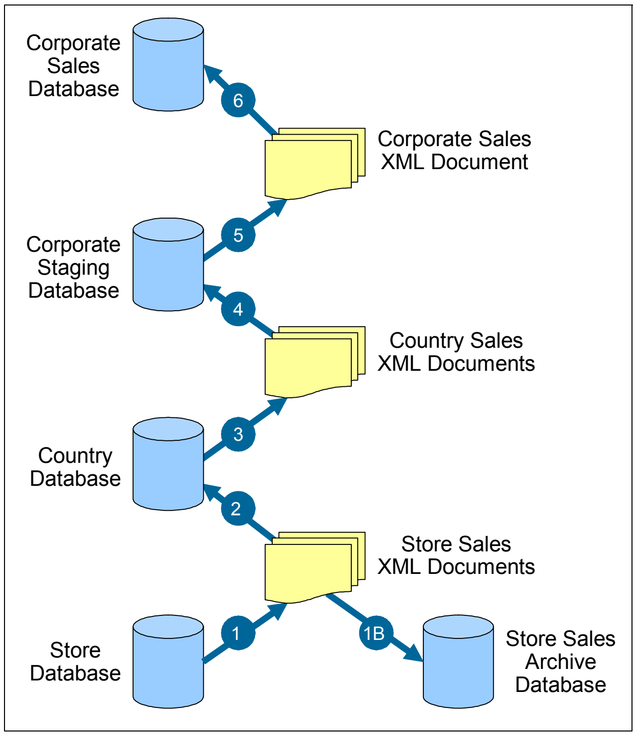

Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 2-1 Operations overview

To show various techniques for integrating XML into DB2 for i5/OS, we provide various compositions and decompositions at each level. Our point of sale (POS) system is already running natively on a System i machine using a standard relational database (RDB) architecture.

The steps shown in Figure 2-1 are explained as follows. For each step, we indicate where further information is provided in this redbook:

1 To start, we compose (generate) an XML document at the store transaction level to send to the country office.

In Chapter 3, "Using SQL to compose XML" on page 33, we show step 1 with SQL only. In 5.1, "Composing XML using RPG" on page 60, we show another way to perform step 1 with Common Gateway Interface (CGI) in RPG.

1B We archive the XML documents intact in case we need to reference them in the future.

This step is an aside that shows how to use the XML Column method to archive XML documents intact. You can learn more about this step in 8.3.1, "XML Column method" on page 116.

2 The country offices must decompose the store sales XML document into their database that contains data from all stores in the country, which are broken down by regions.

In Chapter 4, "Using XSL Transformation and SQL" on page 43, we show this step with an Extensible Stylesheet Language (XSL) decompose.

3 The country office level composes an XML document with summary information for that country, by region, to the corporate level.

In 8.4, "SQL mapping DAD composition example" on page 129, we show the XML Extender SQL mapping document access definition (DAD) composition for this step.

4 At the corporate office, the country data is decomposed into a staging database.

In Chapter 10, "Shredding methodology" on page 155, we show the XML Extender RDB node mapping DAD decomposition.

5 At the corporate staging database, an XML document is composed with the brand information per country.

In Chapter 11, "Composing methodology" on page 181, we show the RDB node mapping DAD that is used for the composition of this step.

6 The XML document is decomposed into the corporate sales system, which provides the purchasing department with the information it needs.

In 5.2, "Decomposing XML using RPG" on page 75, we show how RPG does a decomposition. Another method involving the use of Java to complete this step is presented in Chapter 6, "Using SAX and Java to decompose XML" on page 89.

Table 2-1 summarizes the locations in this redbook for the different scenarios and techniques.

Table 2-1 Location in book of scenario implementations

Step	Description	Location in this redbook
1	Compose via SQL	Chapter 3, "Using SQL to compose XML" on page 33
	Compose via CGI in RPG	Section 5.1, "Composing XML using RPG" on page 60
1B	Store XML documents intact in a table	Section 8.3.1, "XML Column method" on page 116
2	Decompose via XSL	Chapter 4, "Using XSL Transformation and SQL" on page 43
3	Compose via SQL mapping DAD	Section 8.4, "SQL mapping DAD composition example" on page 129
4	Decompose via RDB node mapping DAD	Chapter 10, "Shredding methodology" on page 155

Step	Description	Location in this redbook
5	Compose via RDB node mapping DAD	Chapter 11, “Composing methodology” on page 181
6	Decompose via RPG	Section 5.2, “Decomposing XML using RPG” on page 75
	Decompose via Java	Chapter 6, “Using SAX and Java to decompose XML” on page 89

In the following chapters, we detail the techniques and methodologies that are used for each of the different compositions and decompositions. While a company may choose to use a single method in their shop, we switch methods to illustrate the various techniques. Keep in mind that this is only one example. While we demonstrate complex concepts, you must customize the techniques to match your particular requirements.

2.2 Database model

This company is using the power of DB2 for i5/OS and RDB tables. Using a normalized data format with referential integrity, the setup of the databases is such that multiple tables are linked by a series of primary and foreign keys.

The lowest level of the database model is the store level. In the company, each store’s data is saved in their own schema. It is possible that these schemas all exist on the same system or on systems at each store. However, the data at this level is isolated to an individual store. For our example, we have seven stores, which each have a unique StoreID value. The names of the sample schemas are *StoreXXX*, where *XXX* is the StoreID value. Our first store is StoreID number 7. The data for store 7 is stored in schema Store7.

Within StoreXXX, there are three tables, which are shown in Figure 2-2.

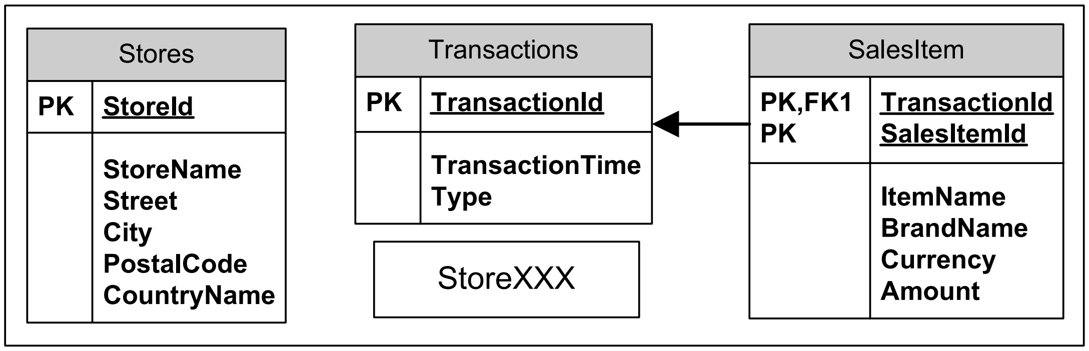


Figure 2-2 StoreXXX schema layout

The *Stores table* provides a directory of all stores in the company. This table is in common among all the stores and is already populated with data from the corporate office. The primary key, a unique value that identifies a row, is the StoreID value. For transactions at that particular store, a standard sales header and detailed table setup are used. This means that for each transaction, there is only one row in the Transactions header table, while there is a row for every item in a transaction in the SalesItem table. There can be one or many rows in the SalesItem table for each sale recorded in the Transactions table (many to one relationship).

The *Transactions table* has a primary key of TransactionID and a time stamp for that transaction. It also indicates the type of transaction, either a sale or a return.

The *SalesItem table* has a compound primary key of TransactionId and SalesItemId. The SalesItemID is the ordinal position of the item purchased. For each transaction, there must be an item for which we create duplicate keys. By combining both the TransactionId and SalesItemId together, a unique value is created. TransactionId is also a foreign key back to the Transactions table. A *foreign key* is a requirement that a value must be defined elsewhere before it is used here. In our example, a TransactionId cannot be added to the SalesItem table without a matching TransactionID in the Transactions table. The last fields in the SalesItem table indicate the brand name of the item sold, a descriptive item name, the currency used for the purchase (since this is a multinational company), and the amount paid.

At the country level, there is one schema per country. As with the store level, the schema can exist on the same system or on different physical systems. The names of our sample schemas are *CountryXXX*, where *XXX* is either USA for the United States of America, POL for Poland, or COL for Colombia.

In each CountryXXX schema, there are three tables as shown in Figure 2-3.

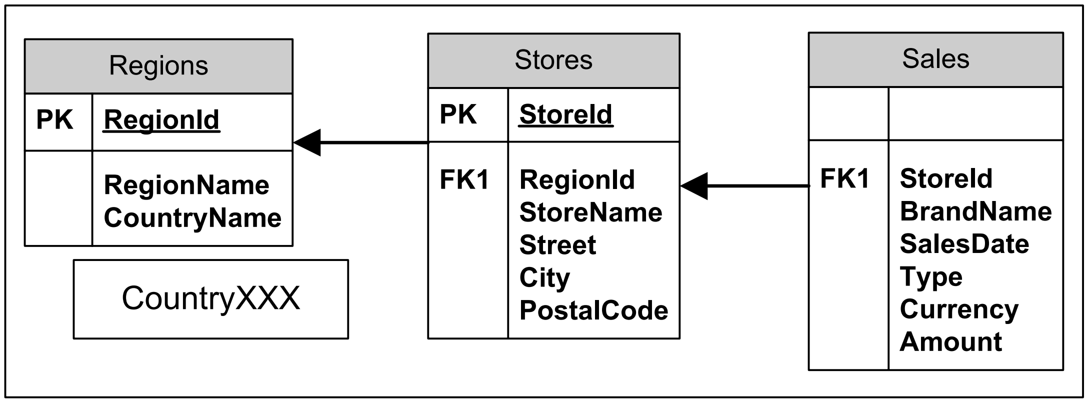


Figure 2-3 CountryXXX schema layout

The *Regions table* contains a list of all regions and their associated countries. Similar to the Stores table in the StoreXXX schema, the Regions table contains all regions, not just those in the country for the schema. The primary key is the RegionId, which means that each region and country combination must have a unique identifying value.

The *Stores table* is similar to the Stores table in the StoreXXX schema. However, it contains a RegionId and not the country. It also has RegionID as a foreign key to the Regions table. Both the Regions and Stores tables are populated with data from the corporate office.

The *Sales table* contains a summary of sales per store, brand, date, type, currency, and amount.

The third schema in our example is for the corporate staging level. At this point, the schema needs to hold information about each region from each country. This schema is called *CorpStage* in our sample. The corporate staging database contains three tables as shown in Figure 2-4.

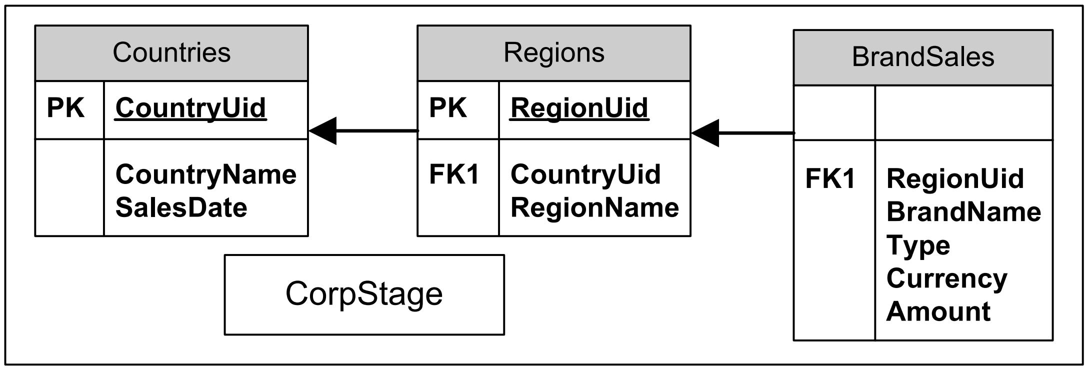


Figure 2-4 CorpStage schema layout

The *Countries table* contains a primary key of CountryUid, a country name, and sales date. If there are multiple sales dates for the same country, there will be multiple rows for that country, one for each date.

The *Regions table* contains a primary key of a RegionUid, a foreign key link back to CountryUid, and the region name. It is important to note here that we are calling them CountryUid and RegionUid. The Uid stands for *unique identifier*. These tables are not populated with data at the start. As data is placed into these tables, a unique value is assigned to them to allow for standard RDB joining.

The *BrandSales table* contains the RegionUid, brand name, type of sale, currency, and amount.

The last schema is the final corporate sales database is called *CorpSales*. This database holds only one table, which has one row per country, date, brand, type of sale and currency, as shown Figure 2-5. Since the values can be duplicated, there is no primary key. The data provided in this table meets what the purchasing department requires to leverage their buying power.

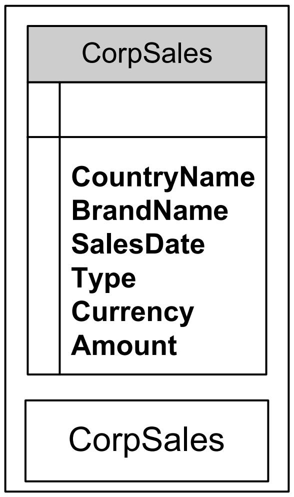


Figure 2-5 CorpSales schema layout

2.3 XML document models

In addition to the database structure, three different XML documents are used to pass the data between the layers. The major difference between the DB2 database representations of the data and the XML representations of the data is the format. There must be a switch from relational thinking to hierarchical thinking.

The first XML document is generated from the StoreXXX that feeds the CountryXXX database. This document has the root element named StoreSales and contains information that is still at the transaction level but only for a particular date and store. Example 2-1 shows the XML layout of StoreSales.

In Example 2-1, the values inside the percent symbols, such as `%YYYY-MM-DD%`, are used as placeholders for real data. The StoreSales element contains the attributes of Date, the XML schema instance definition, and the XML schema definition.

There are two elements inside the StoreSales element. The StoreID element only appears once and contains the store identifier number. The Transactions element is a wrapper for other elements. You might recall that a wrapper element is one that only contains other elements; it has no attributes or data of its own. Inside of Transactions, there are multiple Transaction elements. A Transaction has an attribute of Type that is either a SALE or RETURN. Inside a Transaction, there can be one or more SalesItem elements. Inside of SalesItem there are four elements: Brand, Name, Currency, and Amount. Brand has an attribute of name, while the other three elements have data inside the elements.

Example 2-1 StoreSales XML layout

```
<?xml version="1.0" encoding="UTF-8" ?>
<StoreSales date="%YYYY-MM-DD%"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="StoreSales.xsd">
   <StoreId>%#%</StoreId>
   <Transactions>
      <Transaction type="%SALE/RETURN%">
         <SalesItem>
            <Brand name="%BrandName%" />
            <Name>%ItemName%</Name>
            <Currency>%Cur%</Currency>
            <Amount>%XXXXX.XX%</Amount>
         </SalesItem>
      </Transaction>
   </Transactions>
</StoreSales>
```

In addition to the sample document in Example 2-1, XML can be described by either a document type definition (DTD) or an XML Schema Definition (XSD). See 1.1.2, “Document type definition” on page 5, or 1.1.3, “XML schema” on page 6, for more details. To help describe these XML documents, the complete DTD and XSD are provided in the additional materials that you can download for this redbook. For details about downloading this material, see Appendix A, “Additional material” on page 207.

The second XML document is the CountrySalesByRegion document. Example 2-2 is a sample XML file showing the CountrySalesByRegion layout. This document contains information for one country that is broken down by regions, brand, currency, and the total amount of sales and returns. This is the next level of rollup where we lose the transaction detail level but now have data per region.

The root element in this document is CountrySalesByRegion, which contains the two elements CountryInfo and Regions. Inside CountryInfo is one element called Name, which contains the name of the country. Regions is a wrapper element that contains multiple copies of the Region element. Inside each Region element is a Name element and multiple Brand elements. The Name element at this level is the name of the region. Inside each Brand element are a Name element, an optional Sales element, and an optional Returns element. The Name element at this level is the name of the brand. The optional Sales and Returns elements contain the same two elements. First is a Currency element, which contains the name of the currency. Second is the Amount element, which is the numeric total for the transaction type in that currency, in that brand, in that region, for that country.

Example 2-2 CountrySalesByRegion XML layout

```
<?xml version="1.0" encoding="UTF-8" ?>
<CountrySalesByRegion Date="%YYYY-MM-DD%"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="CountrySalesByRegion.xsd">
   <CountryInfo>
      <Name>%Country%</Name>
   </CountryInfo>
   <Regions>
      <Region>
         <Name>%Region%</Name>
         <Brand>
            <Name>%Brand%</Name>
            <Sales>
               <Currency>%Cur%</Currency>
               <Amount>%XXXXX.XX%</Amount>
            </Sales>
            <Returns>
               <Currency>%Cur%</Currency>
               <Amount>%XXXXX.XX%</Amount>
            </Returns>
         </Brand>
      </Region>
   </Regions>
</CountrySalesByRegion>
```

The third XML document is the CorpSales document. This contains the final rollup where we drop the region level of detail and replace it with the country level as the lowest item. Example 2-3 is a sample XML file showing the CorpSales layout. The CorpSales element has an attribute of date and contains two elements of CountryInfo and SalesByBrand. CountryInfo is a wrapper element that contains just a Name element. SalesByBrand is a wrapper element that contains multiple Brand elements. Each Brand element has a Name element and optional Sales and Returns elements. Each Sales or Returns element has Currency and Amount elements. While this is similar to what we have seen before, they are different in their values and the meanings behind the data.

Example 2-3 CorpSales XML layout

```
<?xml version="1.0" encoding="UTF-8" ?>
<CorpSales Date="%YYYY-MM-DD%"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="CorpSales.xsd">
   <CountryInfo>
      <Name>%Country%</Name>
   </CountryInfo>
   <SalesByBrand>
      <Brand>
         <Name>%Brand%</Name>
         <Sales>
            <Currency>%Cur%</Currency>
            <Amount>%XXXXX.XX%</Amount>
         </Sales>
         <Returns>
            <Currency>%Cur%</Currency>
            <Amount>%XXXXX.XX%</Amount>
         </Returns>
      </Brand>
   </SalesByBrand>
</CorpSales>
```

Part 2

Programmatic approach

The first approach to integrating XML into DB2 for i5/OS is the programmatic approach. In this methodology, the application development team uses traditional programming methods in languages, such as RPG, CL, and SQL, to do the integration. The standard parsers Simple API for XML (SAX) and Document Object Model (DOM) are available to assist.

The advantages of the programmatic approach are that programs can be created and implemented quickly. However, this also leads to the primary problem. Since it is quick, it often lacks reusability and flexibility. We often see developers who must create unique programs with the business logic for each XML document that comes in or goes out.

In this part, we explain the programmatic approach by using examples from our fictional scenario. This part includes the following chapters:

- Chapter 3, “Using SQL to compose XML” on page 33
- Chapter 4, “Using XSL Transformation and SQL” on page 43
- Chapter 5, “Using RPG for XML processing” on page 59
- Chapter 6, “Using SAX and Java to decompose XML” on page 89
- Chapter 7, “Advantages and disadvantages of the programmatic approach” on page 105

3

Using SQL to compose XML

In this chapter, we describe how to use an SQL stored procedure in DB2 for i5/OS that will allow an XML document to be created without the use of any other product. We illustrate step 1 of our scenario. You can only use this method for generating XML.

In this chapter, we discuss the following topics:

- Stored procedure concepts
- Manual database to XML mapping
- Sample stored procedure and its usage

3.1 Overview of stored procedures

A *stored procedure* is a program that can be called from an SQL interface. There are two major categories of stored procedures:

- An *external stored procedure* is a high-level language program that simply gets a wrapper around it that allows it to be called from an SQL interface.
- An *SQL stored procedure* is one where the body of the program is written in the SQL Persistent Stored Module (SQL PSM) language.

The SQL PSM language is extremely powerful and provides much of the functionality found in traditional languages such as C. For full details about SQL PSM, see Chapter 3, “Introduction to the SQL Persistent Stored Module in DB2 UDB for iSeries,” in *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503.

An SQL stored procedure allows a set of operations to be carried out like any traditional program. In addition, input and output parameters can be passed in and out to allow key items to be “parameterized”. This means that we can create one stored procedure with parameters such as store ID and date. We can then call this one procedure with different parameters to achieve different output. The advantage is that we have only one copy of code instead of constantly updating multiple copies that do similar workloads.

In this chapter, we explain how to compose an XML document from a DB2 database using an SQL stored procedure. This is a quick method to generate an XML document. However, it lacks robustness. The code that we use is created specifically knowing how the database and the desired XML document look. The step that we cover is the first composition going from the StoreXXX database into the StoreSales XML document as shown in Figure 3-1.

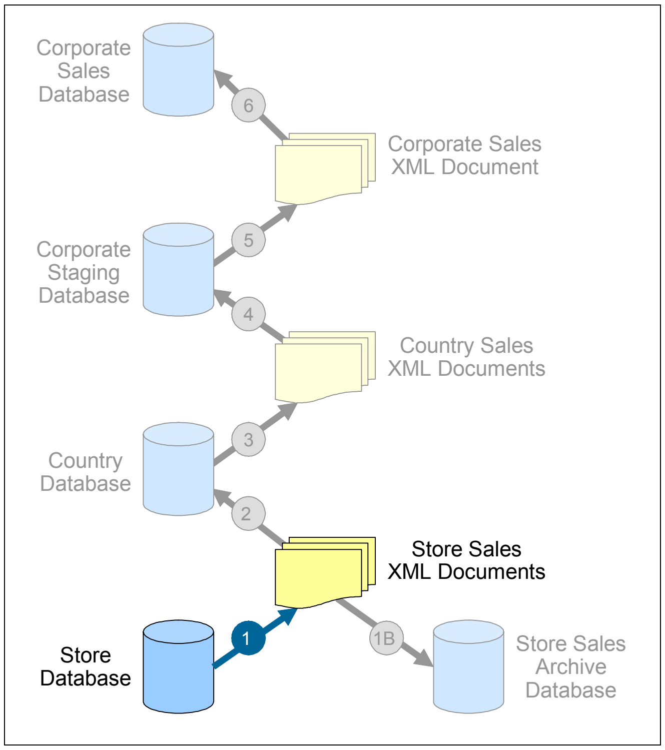

Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 3-1 Scenario step 1

3.2 Mapping the database to XML logically

Our goal is to build the StoreSales XML document from the StoreXXX database. Looking at Example 2-1 on page 27, the StoreSales XML document contains information about a single store. In that example, only seven values need to be filled in. These values are the date and store number at the highest level, the transaction type at the header level, and brand, item, currency, and amount at the item level. Where does this information come from?

The store ID and the date are the keys for building this document. To make this stored procedure as useful as possible, we make these two values parameters that are passed in. This also means that we have the values to insert into the XML document.

This leaves the header and item levels to be completed. Based on our knowledge of how the StoreXXX database is laid out, we only have one level of nesting. This nesting is from the TransactionID into the SalesItem table, where we need to join the individual items to the header information.

Using this setup, we can fill in the header level by selecting from the Transactions table where the date equals the value passed in. Since the data for each store is kept in its own schema, we select from that schema only. Reading each transaction, we can pull out the TransactionId, which allows us to obtain the individual items from the SalesItem table. Armed with this TransactionId, we can then read each item in SalesItem and write the last four values.

If we continue to loop through the Transactions table and nest loop though SalesItem, we complete all the required items in the StoreSales XML document. Table 3-1 illustrates the relational to hierarchical mapping.

Table 3-1 Relational to hierarchical mapping

Location path	Table name	Column name
/StoreSales/@date	(Passed in as a parameter)	
/StoreSales/StoreID	(Passed in as a parameter)	
/StoreSales/Transactions/Transaction/@type	Transactions	Type
/StoreSales/Transactions/Transaction/SalesItem/Brand/@name	SalesItem	BrandName
/StoreSales/Transactions/Transaction/SalesItem/Name	SalesItem	ItemName
/StoreSales/Transactions/Transaction/SalesItem/Currency	SalesItem	Currency
/StoreSales/Transactions/Transaction/SalesItem/Amount	SalesItem	Amount

3.3 Coding the SQL stored procedure

Now that we have worked through the thought process of how to map the StoreXXX database to the StoreSales XML document, we must write the code for this. Example 3-1 shows the code of the SQL stored procedure to execute this logic. A detailed description follows the code.

Example 3-1 Source of GenStoreXML SQL stored procedure

```
create procedure xmlredbook/GenStoreXML
   (IN dateval Char(10), IN storeval varchar(10))                    1
   language SQL
   begin
      declare string char(30000);                                    2
      declare transid INTEGER;
      declare libval varchar(50);
      declare liblen integer;

      set libval = 'Store' || storeval;                              3
      set liblen = (length(libval) + 18);
      set string = 'call qsys/qcmdexc(''CHGCURLIB CURLIB(' || libval ||
         ')'', 00000000' || cast(liblen as decimal(15,5)) || ')';    4
      prepare s1 from string;
      execute s1;                                                    5
      create table qtemp/outfile(char1 char(1000));                  6

      insert into qtemp/outfile values('<?xml version="1.0" encoding="UTF-8"?>'); 7
      set string = '<StoreSales date="' || dateval ||
         '" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="StoreSales.xsd">';           8
      insert into qtemp/outfile values(string);
      set string = '<StoreId>' || storeval || '</StoreId>';          9
      insert into qtemp/outfile values(string);
      insert into qtemp/outfile values('<Transactions>');            10

      FOR each_transaction AS cursor1 CURSOR FOR                     11
         select TransactionId, Type from Transactions
         where dateval = char(date(TransactionTime), ISO)            12
      DO set string = '<Transaction type="' || Type || '">';         13
         insert into qtemp/outfile values(string);
         set transid = TransactionId;                                14
         FOR each_salesitem AS cursor2 CURSOR FOR
            select ItemName, BrandName, Currency, Amount from SalesItem
            where TransactionID = transid                            15
         DO set string = '<SalesItem><Brand name="' || replace(BrandName, '&', '&amp;')
               || '" /><Name>' || ItemName || '</Name><Currency>' || Currency ||
               '</Currency><Amount>' || rtrim(char(Amount)) || '</Amount></SalesItem>';
            insert into qtemp/outfile values(string);                16
         END FOR;
         insert into qtemp/outfile values('</Transaction>');         17
      END FOR;

      insert into qtemp/outfile values('</Transactions>');           18
      insert into qtemp/outfile values('</StoreSales>');

      set liblen = (length(libval) + 147);                           19
      set string = 'call qsys/qcmdexc(''CPYTOIMPF FROMFILE(QTEMP/OUTFILE)
         TOSTMF(''''/XMLRedbook/StoreXML/' || libval || substring(dateval,9,2) ||'.xml'''')
         MBROPT(*REPLACE) STMFCODPAG(819) RCDDLM(*CRLF) DTAFMT(*FIXED)
         STRDLM(*NONE)'', 0000000' || cast(liblen as decimal(15,5)) || ')'; 20
      prepare s2 from string;
      execute s2;                                                  21
      drop table qtemp/outfile;                                    22
   end;
```

We now break down the code of this stored procedure to explain it. At line 1 in Example 3-1, we define the parameters of the stored procedure. This procedure has two input parameters. These input parameters are the date, in ISO format, for example ‘2006-04-06’, and the store ID, such as 7.

At line 2, we start the main body of the stored procedure. We must declare all our variables at the top of the procedure. We declared a character variable (string) to help us build our character strings. We also have a variable (transid) to hold our TransactionId so that we can manually do our joining between Transactions and SalesItem.

At line 3, we start to build our variables. Since we only passed in the StoreID value, we must concatenate ‘Store’ on the front to get the complete store schema. Since each of the different stores are kept in their own schema, we need a way to switch the schemas dynamically. Dynamic SQL statements allow us to do that. However, when we reach the FOR loops later, we cannot use dynamic SQL. This means that we must make the switch from SQL naming to SYS naming to compile this. (See 3.5, “Running the SQL stored procedure” on page 40, for details.) If we do not use SYS, the schema is set at compile time and not at run time.

At line 4, we assign the string to execute the Change Current Library (CHGCURLIB) CL command to set the correct library list for our store. When the string is set, at line 5, we run the prepared statement to change the current library.

To help build the XML document, we create a temporary table at line 6 in Example 3-1 with only one column. This table must be long enough to hold our longest insert. After the table is created, we can start building our XML document. At line 7, we insert the standard XML document header. At line 8, we continue to build our XML document. We can also build strings to insert where we put text and variables together. At line 9, we build the StoreID element with the tags and the real text node data. At line 10, we enter the Transactions wrapper element that holds the Transaction elements.

Now that we are into the main part of the document, we must build our recursion, which we do by using the SQL FOR statement. At line 11, we define a cursor that will pull the data from our Transactions table.

Important: This is not a traditional cursor as we think of in embedded SQL programming. In this case, the cursor builds a temporary table with the results of the statement. The DO section of the FOR loop then operates on each row of this temporary table.

At line 12 in Example 3-1, we add the WHERE clause that limits the selection to just the date that was passed in. Since the table contains a timestamp data type, we need to do some conversion to make the timestamp field match the character string that was passed into the stored procedure.

Since we are into the DO section of the FOR loop, we found at least one transaction. Therefore, at line 13, we build the string to stick in the transaction header with the transaction type for that transaction. Since we need to do this join manually, we save the value of the

TransactionID into our transid global variable at line **14** so that we can limit the selection from SalesItem.

We then start our second FOR loop to select from the SalesItem file at line **15** in Example 3-1 on page 36. At this level, we use the transid global variable to limit the cursor only to the items that were a part of that transaction. When we have this second cursor with just the items for the specific TransactionId, we can build the complete string for the item and insert that into our temporary table at line **16**.

There are two special items inside this string at line **16**. The first is on the BrandName field. Since we know our data, we know there are brands, such as ITSO Electronics, that contain a special XML character. These characters are used as special items in XML documents, such as <, >, and &, and cannot be used directly. Instead, an XML entity is used. An *entity* is a replace string that signals one of these special characters. For example, instead of using an ampersand (&), use `&amp;`. Using the SQL replace function, we can easily substitute `&amp;` for `&` in our data.

The second special item is the Amount field. Since this is stored as a decimal value, the character conversion of the decimal may include extra spaces inside the tags. To remove these, the right trim (rtrim) function is used on the decimal to character (char) function on the Amount field.

At line **17**, we end our inside loop and return to the outside loop. Here we must close our Transaction element. We then exit our outside loop and print the closing lines of the XML document, which is done at line **18**. At this point, the XML document is completely built inside the temporary table.

The key to using the QCMDEXC command is to pass the proper length of the first parameter as the second parameter. At line **19**, we calculate the proper length since the StoreId can be one, two, or three digits long. We did a similar calculation at line **3**.

To obtain the content from the temporary table, we use the Copy to Imported File (CPYTOIMPF) command. To use this, we start building the string at line **20**. We build the integrated file system file name by placing the static part with the dynamic parts of the StoreID and the day value from the date value that was passed in as the first parameter of the stored procedure. At line **21**, we run this statement to copy the temporary database table to the integrated file system where the XML documents are normally stored. Finally, at line **22**, we clean up the temporary table and end the stored procedure.

Overall, you can see that the flow of the stored procedure is to build an XML document from the top-down by following the mapping discussed in 3.2, “Mapping the database to XML logically” on page 35. When it is built, we copy the XML document from the database file to an integrated file system file for further use.

3.4 Compiling the SQL stored procedure

Now that we see how to code the SQL stored procedure, we must compile it. While it may seem straight forward, there are a few points to watch for. As mentioned, the SQL stored procedure needs to be compiled with SYS naming and not SQL naming. The source for this SQL stored procedure, as well as all other items, is available in the additional materials that are available for download. For information about downloading this code, see Appendix A, “Additional material” on page 207.

As part of the information that is extracted, several files in the /XMLRedbook/SQL Source folder are SQL files. If your PC has iSeries Navigator installed, which is part of iSeries Access for Windows®, all files that have the SQL extension should be associated with the Run SQL Scripts program. You simply double-click the **Create GenStoreXML.sql** file to open the Run SQL Scripts program with our sample stored procedure.

We cover the Run SQL Scripts interface in this section. If you use another method to run SQL statements, follow the steps required for that interface to make these changes.

To connect to your system of choice and then configure the SQL Scripts connection to have *SYS naming convention:

1. On your Windows Desktop, double-click the **iSeries Navigator** icon.
2. Click the server name and then sign on with a valid user ID and password.
3. Expand **Databases** and select ***your database***. Right-click and select **Run SQL Scripts...** to launch the SQL Script Center.
4. In the new Run SQL Scripts window, from the menu bar, click **Connection** → **JDBC Setup**.
5. Click the **Format** tab, and under Naming convention, verify that ***SYS** is specified. Otherwise specify ***SYS**, and click **OK**. A naming convention of *SYS requires that you use a forward slash (/) as the separator (for example, library/object).

Since we are using a system naming convention, the compiler needs to find a copy of the files to build the stored procedure correctly. Therefore, we add one of the StoreXXX libraries, such as Store7, to the library list. In Run SQL Scripts, we can run any CL command by typing a `CL:` before the command.

To add the Store7 library to our library list in Run SQL Scripts, we run the following statement:

```
CL: ADDLIBLE LIB(Store7);
```

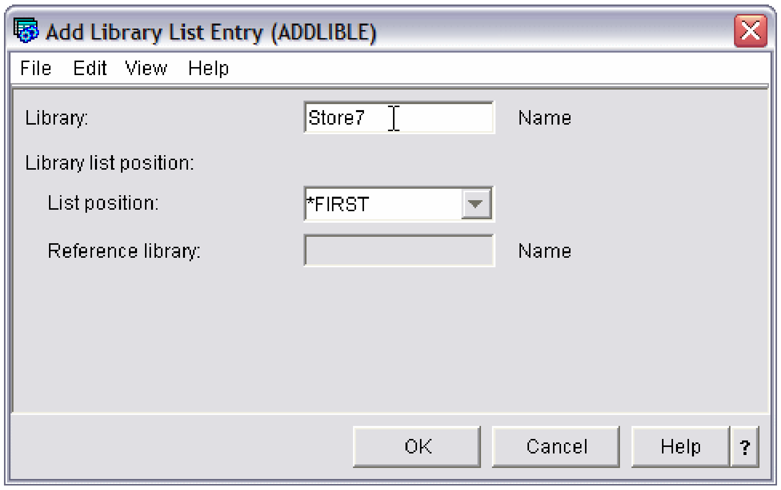
Running this statement before creating the stored procedure adds the library and allows the stored procedure to compile correctly. There are several methods of running statements in Run SQL Scripts. Some of the methods to run an SQL statement in Run SQL Scripts are:

- Place your cursor in the statement and click **Run** → **Selected**.
- Place your cursor in the statement and press Ctrl + Y.
- If **Options** → **Run Statement on Double-Click** is selected, double-click the SQL statement.
- Place your cursor in the statement and click the **Run Selected** (⌛) icon.

In regard to `CL:` functionality, while some CL commands were prompted in the past, now all CL commands can be graphically prompted by typing the command and pressing the F4 key. For example, type the following command and then press F4:

```
CL: ADDLIBLE
```

Figure 3-2 shows a sample of this prompting.



Library: Store7 Name

Library list position:

List position: *FIRST

Reference library: Name

Figure 3-2 Sample of CL prompting in Run SQL Scripts

3.5 Running the SQL stored procedure

Now that the stored procedure is created, we must run it. Earlier in our discussion earlier, we built in parameters of the store ID and the date. This allows us to use the same procedure multiple times. In our scenario, since we have seven stores with two dates, we must call the stored procedure 14 times. Because we created this as a stored procedure, we use the SQL Call from any SQL interface, such as Run SQL Scripts.

In the downloaded material in /XMLRedbook/SQL Source, in addition to the source of the stored procedure, is a file named Run GenStoreXML.sql. Example 3-2 shows the source of this file. From this file, we can see that we can simply call the same stored procedure multiple times and get the 14 resulting XML documents.

From the source of the stored procedure in Example 3-1 on page 36, we can see the location of the XML documents is /XMLRedbook/StoreXML/Store<StoreId><Day portion of date>.xml, such as /XMLRedbook/StoreXML/Store706.xml.

Example 3-2 Source of Run GenStoreXML.sql

```
call xmlredbook.GenStoreXML('2006-04-06', '7');
call xmlredbook.GenStoreXML('2006-04-06', '42');
call xmlredbook.GenStoreXML('2006-04-06', '71');
call xmlredbook.GenStoreXML('2006-04-06', '87');
call xmlredbook.GenStoreXML('2006-04-06', '172');
call xmlredbook.GenStoreXML('2006-04-06', '207');
call xmlredbook.GenStoreXML('2006-04-06', '210');

call xmlredbook.GenStoreXML('2006-04-07', '7');
call xmlredbook.GenStoreXML('2006-04-07', '42');
call xmlredbook.GenStoreXML('2006-04-07', '71');
call xmlredbook.GenStoreXML('2006-04-07', '87');
call xmlredbook.GenStoreXML('2006-04-07', '172');
call xmlredbook.GenStoreXML('2006-04-07', '207');
call xmlredbook.GenStoreXML('2006-04-07', '210');
```

3.6 Deploying stored procedures

Stored procedures contain two parts, a catalog entry and a program object. These two items can exist separate from each other but only work when both parts are there at run time. The operating system allows for stored procedures to be registered into the catalog when the program object is restored. This allows for easy propagation of stored procedures. However, there are some finer points to make this work.

For full details about how to properly save and restore stored procedures, refer to:

- "Moving into production (save and restore)" in *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503
- KnowledgeBase document 390520958 "Stored Procedures Restore Overview"

  `http://www-912.ibm.com/s_dir/slkbase.nsf/slkbase`

If we are using SQL stored procedures only, it is possible to simply run the create procedure statement on each box. This way we do not have to worry about the program objects. By running the SQL script, the program object and the catalog entry are created automatically.

4

Using XSL Transformation and SQL

In this chapter, we introduce the use of Extensible Stylesheet Language (XSL) to decompose an XML document into a table using Extensible Stylesheet Language Transform (XSLT) and Java. A stylesheet can only be used for decomposition because it accepts only XML as input.

We provide a solution for step 2 (Figure 4-1) from our fictional scenario. Specifically, we discuss decomposing XML using XSLT, Java, and SQL.

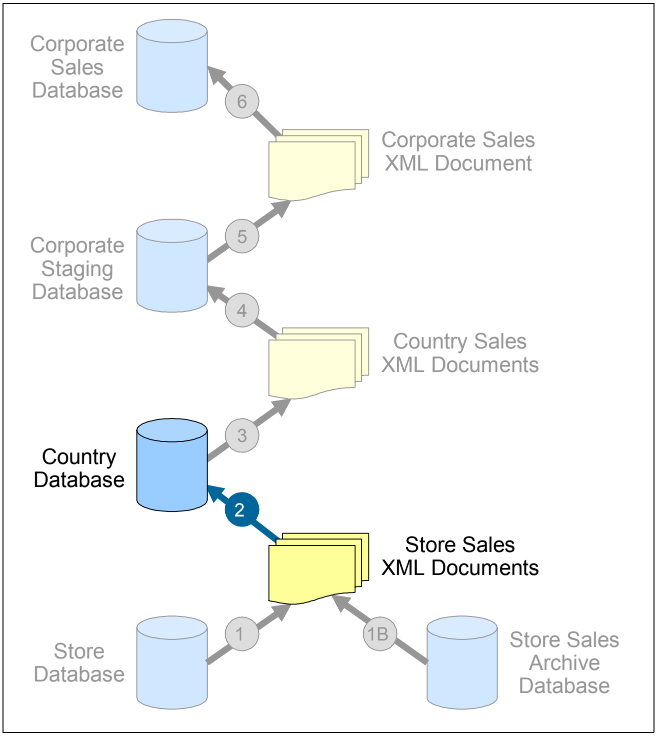

Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 4-1 Scenario step 2

4.1 Decomposing XML using XSLT, Java, and SQL

A stylesheet can transform or modify an XML document into another XML or text-based type of document that is recognized by a browser such as HTML or Extensible Hypertext Markup Language (XHTML). With XSL Transformation, you can add or remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests, and make decisions about which elements to keep, remove, and a lot more. Typically an XSL document is used to reformat XML for browser output (HTML), but in our scenario, we used XSL to transform an XML document into SQL script to populate our Sales table.

For more information about the basics of XSL, XSL Transformations, and its applications, refer to the following Web address:

http://www.w3schools.com/xsl/

Using this functionality, we can decompose our StoreSales XML document.

4.2 Prerequisites

In the examples presented in this section, we use Integrated Language Environment® (ILE) CL, CL user commands, and Panel Groups (PNLGRP). To code the examples in this section, we must meet the following requirements:

- Verify that the default version of Java is 1.4 by entering the following command in Qshell:

  ```
  java -version
  ```

- Verify that WebSphere Development Studio - Option 21 (5722WDS) is installed on the System i platform.

4.3 Decomposition planning and design

The StoreSales XML document is the solution for step 2 of our scenario shown in Figure 4-1. It is the first process that requires decomposition of our A^{2}BCM company scenario.

Since this decomposition is for several stores that are reporting their sales data for a day, each contained in a separate XML document, we require input parameters to be passed to the decomposition processor. This is accomplished by providing a CMD interface to our ILE CL module along with an associated help PNLGRP. This flexibility allows the same decomposition processor to be used for any number of inbound StoreSales XML documents that potentially reside in different integrated file system directories or have different document names.

In addition, an ILE CL program was developed to validate the user input prior to processing to prevent any errors in user input at the point of origination. This approach grants us flexibility in reuse of a one-coded solution for any number of document locations and naming conventions that are provided.

4.4 Decomposition coding

In this section, we explain the program flow, the different modules, and the XSL document coding.

4.4.1 Program flow

Figure 4-2 illustrates the processing flow of the StoreSales XML document decomposition.

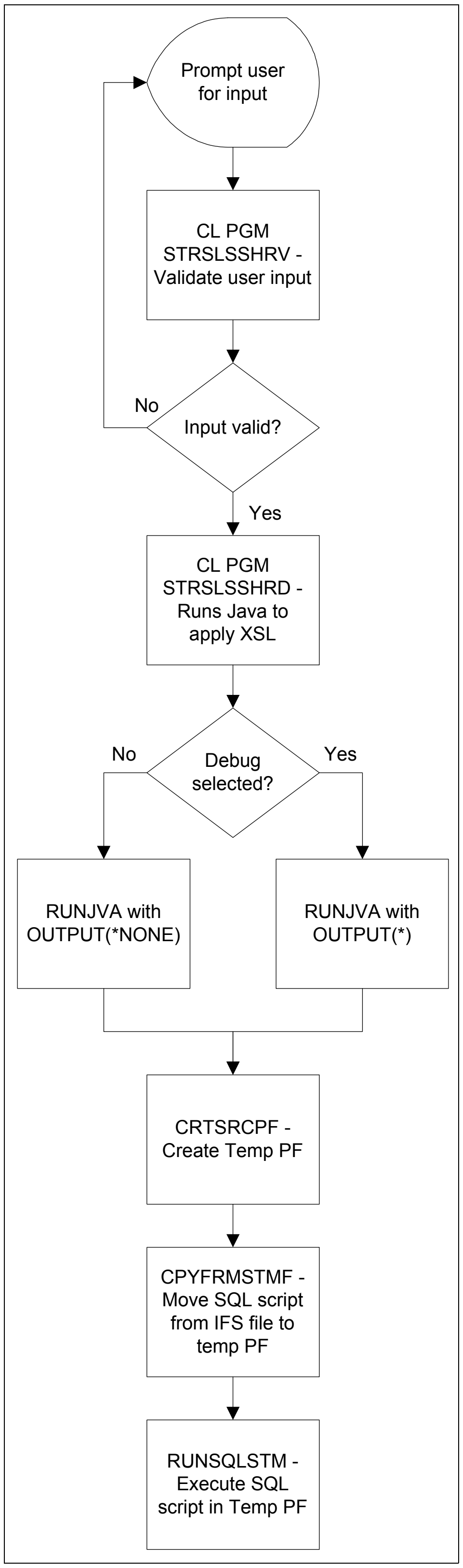

Prompt user for input
CL PGM STRSLSSHRV - Validate user input
Input valid?
No
Yes
CL PGM STRSLSSHRD - Runs Java to apply XSL
Debug selected?
No
Yes
RUNJVA with OUTPUT(*NONE)
RUNJVA with OUTPUT(*)
CRTSRCPF - Create Temp PF
CPYFRMSTMF - Move SQL script from IFS file to temp PF
RUNSQLSTM - Execute SQL script in Temp PF

Figure 4-2 StrSlsShrd program flow

4.4.2 Required modules

The following modules were created for decomposition processing via XSL:

- STRSLSSHRV Control Language Integrated Language Environment (CLLE) to perform command parameter validity checking
- STRSLSSHRD CLLE to decompose an XML document and to populate the Sales table using the RUNJVA command to call Java and the RUNSQLSTM command to process the generated SQL script
- STRSLSSHRD Command (CMD) to provide a user interface for input variables
- STRSLSSHRD PNLGRP to provide for help text for each command parameter

The source code for the associated CMD, PNLGRP, and CLLE validity checking program, as well as all other items, is available in the additional materials that are available for download with this IBM Redbook. Module generation instructions are contained in each of the sample source members. For information about downloading this code, see Appendix A, "Additional material" on page 207.

4.4.3 Input documents

Example 4-1 and Example 4-2 are of the inbound XML and document type definition (DTD) documents respectively. These two inputs are the basis for the mapping of the XML document to our target Sales table using the XSL Transformation document. We begin by reviewing our StoreSales XML document to determine the tree structure of the XML XPATH nodes noting any repeating or wrapper elements.

Example 4-1 shows the StoreSales XML document.

Example 4-1 StoreSales XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<StoreSales Date="2006-04-06"                                       2
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="StoreSales.xsd">
   <StoreId>7</StoreId>                                             2
   <Transactions>
      <Transaction type="SALE">                                     3
         <SalesItem>                                                4
            <Brand name="Pepsi" />
            <Name>Mt. Dew 20 oz.</Name>
            <Currency>USD</Currency>
            <Amount>1.19</Amount>
         </SalesItem>
         <SalesItem>
            <Brand name="General Electric" />
            <Name>60 watt bulbs, 4 pk.</Name>
            <Currency>USD</Currency>
            <Amount>.98</Amount>
         </SalesItem>
      </Transaction>
   </Transactions>
</StoreSales>
```

Example 4-2 shows the StoreSales DTD document.

Example 4-2 StoreSales DTD document

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT StoreSales (StoreId,Transactions) >
<!ATTLIST StoreSales
  Date CDATA #IMPLIED>

<!ELEMENT StoreId (#PCDATA)>
<!ELEMENT Transactions (Transaction+) >                          3

<!ELEMENT Transaction (SalesItem+) >                             4
<!ATTLIST Transaction
  type CDATA #IMPLIED>

<!ELEMENT SalesItem (Brand,Name,Currency,Amount) >

<!ELEMENT Brand EMPTY>
<!ATTLIST Brand
  name CDATA #IMPLIED>

<!ELEMENT Name (#PCDATA)>
<!ELEMENT Currency (#PCDATA)>
<!ELEMENT Amount (#PCDATA)>
```

4.4.4 Target table

Table 4-1 shows the existing Sales table into which we will decompose our XML document. This table layout, along with the previous examples of the XML and DTD documents, gives us all the input that is required to begin mapping in the XSL to decompose our XML document.

Refer to 10.2, “Analyzing the XML document structure and determining the mapping” on page 157, for more information about XML document structure and mappings.

Table 4-1 Sales table

Column name	Data type	Length	Allocated
StoreId	Integer		
BrandName	VarChar	80	50
SalesDate	Date		
Type	VarChar	30	10
Currency	VarChar	30	10
Amount	Decimal	9.2	

4.4.5 XSL transform document

Example 4-3 shows the StoreSales XSL document that was developed to decompose an XML document by transforming the contents to an SQL script.

Example 4-3 StoreSales XSL document

```
<?xml version="1.0"?>
<!-- ***************************************************************** -->
<!-- *  This XSL is used to transform the StoreSales XML document to    * -->
<!-- *  SQL script that will insert the decomposed data to our Sales    * -->
<!-- *  table.                                                          * -->
<!-- ***************************************************************** -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="text" indent = "no" encoding="UTF-8"/>     1

<!-- Process complete tree starting from root-->
<xsl:template match="/">

<!-- Set sales date variable with XML data from Date attribute -->
   <xsl:variable name="date">                                      2
      <xsl:value-of select="StoreSales/@Date"/>
   </xsl:variable>

<!-- Set store id variable with XML data from StoreId attribute -->
   <xsl:variable name="id">
      <xsl:value-of select="StoreSales/StoreId"/>
   </xsl:variable>

<!-- Process each repeating Transaction element -->
   <xsl:for-each select="StoreSales/Transactions/Transaction">     3
      <xsl:variable name="type">
         <xsl:value-of select="@type"/>
      </xsl:variable>

<!-- Process each repeating SalesItem element -->
      <xsl:for-each select="SalesItem">                            4
         insert into Sales                                         5
         (salesdate,storeid,type,brandname,currency,amount)
         values (
            &apos;<xsl:value-of select="$date"/>&apos;,            6
            &apos;<xsl:value-of select="$id"/>&apos;,
            &apos;<xsl:value-of select="$type"/>&apos;,
            &apos;<xsl:value-of select="Brand/@name"/>&apos;,
            &apos;<xsl:value-of select="Currency"/>&apos;,
                  <xsl:value-of select="Amount"/>);
      </xsl:for-each>

   </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

The function of the StoreSales XSL is to:

- Decompose the XML data using the default Xalan parser
- Generate an SQL script to insert XML data into our Sales table

Use the following guidelines when coding the XSL:

- Beginning with the Root element of the XML document, code the transformation stylesheet following the top-down hierarchy flow.
- Match the XML element and attribute names, noting the case, spelling, and tree structure path.
- Use variables (identified by a dollar sign ($) prefix) to save XML data for later use and reuse.
- Attributes in the XML data are identified with an @ prefix.
- Elements in the XML data have no prefix identification.

The concepts illustrated in Example 4-1 on page 46 through Example 4-3 show the mapping of XML element and attribute names in an XML document and associated DTD to the target Sales table (Table 4-1 on page 47) with XSL element coding. In those examples, the numbers correspond to each other to indicate this mapping. For example, 2 corresponds with 2. Using XSL Transformation, the tree structure of the XML document is followed, and we can now break down the components of the XSL coded document in relationship to the input and target documents.

4.4.6 XSL document coding

In this section, we explain the XSL coding using the reverse type numbers in Example 4-1 on page 46, Example 4-2 on page 47, and Example 4-3.

Since we are creating an SQL script to be run following the transformation, the output method used in the stylesheet is set to text as shown at line 1 in Example 4-3. This assures that we produce a document that we can process via an SQL engine.

The Date attribute and StoreID element of the StoreSales parent element must be saved for later processing since they will not be required at this level of the document. Using the XSL element `xsl:variable`, we create a global variable named *date* and populate the variable with the content of the StoreSales Date attribute using the XSL element `xsl:value-of select`:

```
<xsl:variable name="date">
   <xsl:value-of select="StoreSales/@date"/>
</xsl:variable>
```

The correct identification of an attribute is to prefix the attribute name with an @ sign. Therefore, `xsl:value-of select` is coded as `"StoreSales/@date"`, which identifies that the attribute date (@date) is part of the StoreSales element. The slash following the `"StoreSales/@date"` closes out the `xsl:value-of` instruction. Our global variable *date* is now populated and can be used later in the XSLT processing as shown at line 2.

The same applies for the StoreId element (StoreSales/StoreId) value that will be retrieved and stored in the *id* variable for later XSLT processing.

As defined in the DTD, the Transactions element is a wrapper element that can repeat since it is coded with a plus sign (+) at the end of line 3 in Example 4-2 on page 47. The XLS element to repeat is `for-each`. In reviewing the code in the XSL document that is shown, a repeat loop is created for the Transaction element contained within the Transactions wrapper element that is contained in the StoreSales element (StoreSales/Transactions/Transaction). As was the case with the StoreSales date attribute and StoreId element, the Transaction type attribute needs to be stored in a global variable for later XSLT processing. We again create a global variable named *type* that will contain the Transaction Type attribute value.

Following the hierarchy of the XML document, the next element encountered is SalesItem. Again this is a wrapper element that can repeat (as indicated by the + sign in the DTD), so the XSL must begin a `for-each` repeat as indicated at line **4** as well. Since this is the last element to repeat, it contains our lowest level to be reached within the XML document, so we begin generating our SQL script that is output.

Up to this point, all of our XSL coding has been to instruct the XSL on how to process our XML document. Beginning with `insert into Sales` at line **5**, we start coding what is to be placed in our SQL script file that is generated. The text is in a free format so what we type is what is output to the generated file. Following the standard SQL INSERT statement as INSERT INTO TABLE_NAME (COLUMNS) VALUE(), we begin formatting the SQL script line with the following literals:

```
insert into Sales
(salesdate,storeid,type,brandname,currency,amount)
values (
```

This literal is out put to our generated SQL script file as shown at line **5** in Example 4-3 on page 48.

Using the global variables that we set previously, we populate the *values* section of the SQL INSERT statement with the actual data decomposed from the XML document. We populate the values to be inserted into our columns, as a row, in our Sales table.

The following example shows a mixture of literals, character entities, and XSL instructions to populate the salesdate value (see line **6**):

```
&apos;<xsl:value-of select="$date"/>&apos;,
```

In this example, note the following explanation:

- *&apos;* is a character entity that substitutes with an apostrophe in the output. In XML, like HTML, some characters have special meaning (<, >, &, and so on) and are not output if used as is, so a substitution must be done. For more information and examples of character entities, see the following Web site:

  http://www.w3schools.com/tags/ref_entities.asp

- *<xsl:value-of select =”$date”/>* substitutes the value found in the global variable *date* (identified by the $ preceeding the date) that we populated earlier for the salesdate data value in the VALUE() clause that pertains to the salesdate column.
- The closing comma (,) is the comma literal that is inserted to separate the individual values in the VALUE clause.

The same formatting takes place for the storeid column, using the global variable *id* ($id), and the type column, using the global variable *type*($type), that we populated earlier for later XSLT processing previously.

The *value-of select* is again used to extract the XML data for insert. However this time, the data to be processed is from the XML document and not a saved variable. The same formatting rules apply using the apostrophe character entity to enclose our data and the literal comma to end our value. The following statements show our final two character XML data values to be extracted, the attribute name contained in the Brand element and the Currency element:

```
&apos;<xsl:value-of select="Brand/@name"/>&apos;,
&apos;<xsl:value-of select="Currency"/>&apos;,
```

The resulting output generated so far looks like the following example:

```
insert into Sales
(salesdate,storeid,type,brandname,currency,amount)
values (
'2006-04-06',
'7',
'SALE',
'Pepsi',
'Mt. Dew 20 oz.',
'USD',
```

The final XML element to be processed is the Amount:

```
<xsl:value-of select="Amount"/>);
```

The Amount column in our table, being numeric, does not require the extracted value be enclosed in apostrophes as does an alphanumeric value, so the &apos; is omitted. After the `value-of select` XSL instruct closing bracket, the closing parenthesis literal ends our VALUES clause, and the semicolon literal ends the SQL statement.

The resulting output that is generated now looks like the following example:

```
insert into Sales
(salesdate,storeid,type,brandname,currency,amount)
values (
'2006-04-06',
'7',
'SALE',
'Pepsi',
'USD',
      1.19)';
```

Since these XSL instructs are within the *for-each* loops (SalesItem and Transaction), they are run until all the XML data is decomposed and the SQL script to perform the table inserts is generated.

Table 4-2 shows a mapping of the XML element/attribute names to our Sales table columns.

Table 4-2 XML names to Sales table columns mapping

XML name	Column name
StoreSales/Date	salesdate
StoreSales/StoreId	storeid
StoreSales/Transactions/Transaction/type	type
StoreSales/Transactions/Transaction/SalesItem/Brand/name	brandname
StoreSales/Transactions/Transaction/SalesItem/Currency	currency
StoreSales/Transactions/Transaction/SalesItem/Amount	amount

4.4.7 Transform processor

Now that we have coded our XSL to transform the XML document, we must apply the XSL to our XML document and then process the generated SQL script to populate our Sales table. Using the Java Transform class (see Example 10-4 on page 164 for more information about the Java Transform), we can apply the stylesheet to the XML document. The execution of

Java is handled in CLLE with the use of the RUNJVA command to run the XSL transform. We then use the RUNSQLSTM command to execute the generated SQL script after the decomposition process is complete.

To begin our decomposition process, we provide a CMD interface that prompts a user to enter these parameters:

- The directory name that contains the XML document and the XSL that transforms the XML document to SQL script
- The name of the XML document itself
- The name of the schema that contains our Sales table
- The directory used as the Classpath to locate the Transform class
- A debug flag to control interactive display of the Transform class execution messages

Using standard industry accepted CLLE programing methods, we accept the CMD input as variables to our CLLE program. We process the variables as required internal to the program to make them available to our decomposition processing (not be illustrated here). Example 4-4 shows the STRSLSSHRD CLLE program code for the RUNJVA and RUNSQLSTM execution.

Example 4-4 STRSLSSHRD CLLE program code

```
IF         COND(&DEBUG *EQ '*YES') THEN( +                 1
   RUNJVA     CLASS(Transform) +
               PARM(&XMLFILE    +
                   &XSLFILE    +
                   &SQLFILE    +
                   ‘ ‘) +
            CLASSPATH(&CLASSPATH) +
            PROP((java.version 1.4)))
IF         COND(&DEBUG *EQ '*NO') THEN( +
   RUNJVA     CLASS(Transform) +
               PARM(&XMLFILE    +
                   &XSLFILE    +
                   &SQLFILE    +
                   ‘ ‘)    +
               CLASSPATH(&CLASSPATH) +
               PROP((java.version 1.4)) +
               OUTPUT(*NONE))

CRTSRCPF   FILE(QTEMP/QSQLSRC) RCDLEN(80)                  2
MONMSG     MSGID(CPF0000)

CHGVAR     VAR(&FRMSTRMF) VALUE(&DIRECTORY |< &DOCUMENT |< '.sql')

CPYFRMSTMF FROMSTMF(&FRMSTRMF)
            TOMBR('/qsys.lib/qtemp.lib/qsqlsrc.file/StoreSales.mbr')
            MBROPT(*REPLACE)

RUNSQLSTM  SRCFILE(QTEMP/QSQLSRC) SRCMBR(STORESALES)       3
            DFTRDBCOL(&SCHEMA)

CHGVAR     VAR(&SQLFILE) VALUE(&DIRECTORY |< &DOCUMENT |< '.sql')

RMVLNK     OBJLNK(&SQLFILE)                                4
```

The execution of the RUNJVA command is conditioned based on the user input supplied in the DEBUG parameter. The following input is valid:

- DEBUG=*YES indicates to display the Java execution status window if it is run in the interactive environment.
- DEBUG=*NO suppresses the status window (see section 1 in Example 4-4).

The RUNJVA command requires the following parameters for our execution:

- CLASS is the name of the class to be executed. Transform is the class we will use.
- PARM refers to the parameters that we pass to the Transform class (explained later).
- CLASSPATH is the location on the integrated file system where the class is found. This parameter (&CLASSPATH) is supplied during user input at program initialization.
- PROP refers to the properties to be used at execution. We require that the minimum Java version that is used is 1.4, so this is hardcoded in our example.
- OUTPUT determines if the Java execution status window is displayed during execution. This parameter is set based on the value supplied by the user in the DEBUG parameter.

The Transform class requires four parameters that are passed to the class using the PARM parameter on the RUNJVA command:

- XMLFILE name and the location of the XML document to be transformed

  The &XMLFILE parameter is populated using the user supplied directory and document name specified at program initialization.
- XSLFILE name and location of the XSL document to be applied to the XML document

  The &XSLFILE parameter is populated using the user-supplied directory name and our XSL document name STORESALES.XSL.
- SQLFILE name and location of the generated SQL script file generated during transformation processing

  The &SQLFILE parameter is populated using the user-supplied directory name and the document name with .sql appended to it.
- The Guaranteed Universal Unique Identifier (GUUID) value, which is not used for our transformation, so it can default to blank

The following example shows how the RUNJVA command looks with the command parameters displayed and the DEBUG parameter as *YES:

```
RUNJVA     CLASS(Transform) +
              PARM('/XMLRedbook/StoreXML/Store706.xml' +
                  '/XMLRedbook/StoreXML/StoreSales.xsl' +
                  '/XMlRedbook/StoreXML/Store706.xml.sql' +
                  ' ')    +
              CLASSPATH('/XMLRedbook/classes') +
              PROP((java.version 1.4))
```

After the Java program applies the XSL to the XML document, the generated SQL script resides in the integrated file system directory specified by the user who has the same name as the input XML document with an .sql suffix appended to it for uniqueness.

Example 4-5 shows a sample of the generated SQL script.

Example 4-5 Generated StoreSales SQL script

```
insert into Sales
    (salesdate,storeid,type,brandname,currency,amount)
    values (
     '2006-04-06',
     '7',
     'SALE',
     'Pepsi',
     'USD',
           1.19);

    insert into Sales
    (salesdate,storeid,type,brandname,currency,amount)
    values (
     '2006-04-06',
     '7',
     'SALE',
     'General Electric',
     'USD',
           .98);
```

Now that we have generated the SQL script, we must run it to populate the Sales table with the XML data that has been decomposed. We use the RUNSQLSTM command to process the SQL. The RUNSQLSTM command requires the SQL script that is to be executed to be stored as a member in a source physical file. To move the generated SQL script from the integrated file system to an executable form in a source physical file, we copy the contents of the stream file that contains the SQL script to a source member in a source physical file. Since this application only uses the generated SQL script on a temporary basis, we can create a source physical file in the QTEMP library to hold our copied source (see **2** in Example 4-4 on page 52). Next the CPYFRMSTMF command is used to convert the stream file script to a temporary source member so that it may be executed.

Using the RUNSQLSTM command, we can execute SQL from our temporary member in the source file (see **3** in Example 4-4 on page 52). The parameter value for DFTRDBCOL (Default RDB collection) is supplied by the user at program initialization. It instructs the SQL as to the schema in which the Sales table is located since our decomposition module will run for multiple stores, reporting for different countries.

Note: Refer to 4.7, “Some considerations for XSL decomposition” on page 56, for considerations for using the RUNSQLSTM command.

The final step is to remove any temporary work integrated file system objects that we created so we do no clutter the system with orphaned objects (see line **4** in Example 4-4 on page 52). This process can be conditioned by using the DEBUG parameter setting and in the event that we want to keep the generated SQL script for troubleshooting purposes.

4.5 Decomposition module deployment

All CMD, CLLE, and PNLGRP modules can be saved using the SAVOBJ command and restored on a target system using the RSTOBJ command.

The following example shows how to deploy our StoreSales decomposition modules from our source system to a target system:

1. On the source system, create a save file to store our saved objects:
   ```
   CRTSAVF SAVF(XMLREDBOOK/STRSLSMODS)
   ```
2. On the source system, save the StoreSales decomposition modules to the STRSLSMODS save file:
   ```
   SAVOBJ OBJ(STRSLSSHR*) LIB(XMLREDBOOK) DEV(*SAVF) SAVF(XMLREDBOOK/STRSLSMODS)
   ```
3. On the target system, create a save file to receive the saved objects to be deployed:
   ```
   CRTSAVF SAVF(XMLREDBOOK/STRSLSMODS)
   ```
4. On the target system, restore the StoreSales decomposition modules to the target schema:
   ```
   RSTOBJ OBJ(*ALL) SAVLIB(XMLREDBOOK) DEV(*SAVF) SAVF(XMLREDBOOK/STRSLSMODS)
   ```

The StoreSales XSL document and the Java Transform class can be saved by using the SAV command and restored on a target system by using the RST command.

The following example shows how to deploy our StoreSales XSL module and Java Transform class from our source system to a target system:

1. On the source system, create a save file to store our saved module:
   ```
   CRTSAVF SAVF(XMLREDBOOK/STRSLSIFSS)
   ```
2. On the source system, save the StoreSales XSL module and Java Transform class to the STRSLSIFSS save file:
   ```
   SAV DEV(‘/QSYS.LIB/XMLREDBOOK.LIB/STRSLSIFSS.FILE’)
   OBJ((‘/XMLREDBOOK/STOREXML/StoreSales.xsl’)
         (‘/XMLREDBOOK/CLASSES/Transform.class’)) DTACPR(*YES)
   ```
3. On the target system, create a save file to receive the saved module to be deployed:
   ```
   CRTSAVF SAVF(XMLREDBOOK/STRSLSIFSS)
   ```
4. On the target system, restore the StoreSales XSL module to the target schema:
   ```
   RST DEV(‘/QSYS.LIB/XMLREDBOOK.LIB/STRSLSIFSS.FILE’)
   OBJ((‘/XMLREDBOOK/STOREXML/StoreSales.xsl’)) DTACPR(*YES)
   ```
5. On the target system, restore the Java Transform class to the target schema:
   ```
   RST DEV(‘/QSYS.LIB/XMLREDBOOK.LIB/STRSLSIFSS.FILE’)
   OBJ((‘/XMLREDBOOK/CLASSES/Transform.class’)) DTACPR(*YES)
   ```

4.6 Decomposition execution

Now that we have coded and deployed our solution, we must execute the modules and decompose our XML documents. The modules must be executed for each reporting store that provided an XML document that we want to decompose into our Sales table.

The STRSLSSHRD CMD is used to initiate the decomposition process for each Store's sales XML document we want to handle. On a command line, type the following command and press F4 to prompt for user input:

```
STRSLSSHRD
```

We describe the parameters for this command as follows; cursor-sensitive help is available by pressing F1:

- INPUT

  This parameter is composed of two parts, a directory and a document. Enter the integrated file system directory that contains the XML document that you want to decompose. The document is the actual XML document that you are decomposing.

- OUTPUT

  This parameter is the schema that contains our sales database table.

- CLASSPATH

  This parameter is the directory that contains the Transform Class that is required to apply the XSL to the XML document.

- DEBUG

  This parameter controls the display of the Java status window.

The following example shows how the command looks after the user enters the necessary parameters to decompose store seven's sales for day six with debugging:

```
STRSLSSHRD INPUT('/XMLRedbook/StoreXML' 'Store706.xml')
            OUTPUT('CountryXML') +
            CLASSPATH('/XMLRedbook/classes') +
            DEBUG(*YES)
```

After the decompose has completed for the document specified, the command line returns. Validate that the XML data was decomposed correctly and inserted into the Sales table in the schema specified.

4.7 Some considerations for XSL decomposition

As of the writing of this IBM Redbook, you must take into account the following considerations when using XSL to decompose an XML document:

- The decomposition module provided in this chapter does not provide examples of error handling processing. When coding your decomposition solution, keep this in mind to trap any errors that may occur during decomposition and report them back to the user or log for later retrieval.
- Use of the Java virtual machine (JVM™) on the System i platform requires simple tuning to optimize performance. Since we use JVM to run our Transform class, it is important to note the memory pool that your JVM runs in be set appropriately to improve performance. This tuning comes into play more when you are processing large or complex XML documents. Refer to the paper *Tuning Garbage Collection for Java and WebSphere on iSeries* for additional information:

  http://www.ibm.com/servers/eserver/iseries/perfmgmt/pdf/tuninggc.pdf

- The RUNSQLSTM function has a limit in regard to the size of an SQL script that can be handled. Currently this limIt is 16 Mb and must be taken into account if you are attempting to decompose large XML documents.

There are two workarounds for this limitation:

- Create an high-level language (HLL) application that reads the generated SQL script and processes each insert one at a time.
- Use the DB2 command as part of Qshell to run the SQL script.

5

Using RPG for XML processing

In this chapter, we introduce using RPG to decompose XML documents into our relational tables. This chapter provides a solution for step 1 and step 6 from our fictional scenario.

We address the following topics in this chapter:

- Composing XML using RPG
- Decomposing XML using RPG

5.1 Composing XML using RPG

As of V5R4 of IBM i5/OS, the Integrated Language Environment (ILE) RPG compiler has new native operations codes and built-in functions that facilitate composition of an XML document. Through the Client Technology Center (CTC), IBM has provided an open source solution for Web development, via a high-level language (HLL), based on the Common Gateway Interface (CGI) language.

While the main focus of the CGI functionality is to provide Web access to an HLL, in this example we used the CGI toolkit to produce an XML document using RPG. This is an alternate method to perform step 1 in our scenario (Figure 5-1). The first method was presented in Chapter 3, “Using SQL to compose XML” on page 33.

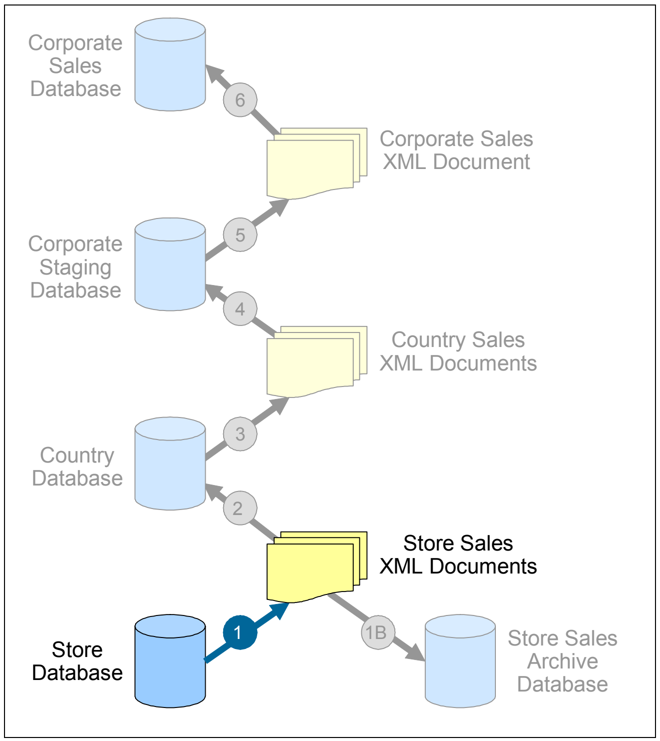

Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 5-1 Scenario step 1: Alternate method

5.1.1 Prerequisites

In the examples in this section, we use ILE CL, CL user commands, and Panel Groups (PNLGRP). We also use CGI, which is an Open Source product. To write the code for the examples in this section, you must meet the following requirements:

- Verify that you have installed WebSphere Development Studio - option 21 (5722WDS) on your System i machine.
- Verify that you have installed WebSphere Development Studio - option 34 (5722WDS) on your System i machine. It is needed for the example done in RPG to compose XML documents.
- Download the CGIDEV2 library from Easy400, an Open Source site, and install it on the System i machine. For instructions and to download the library, go to the following Web address:

  `http://www-922.ibm.com/`

Note: Control Language Integrated Language Environment (CLLE), Command (CMD), UIM PNLGRP, and Report Program Generator (RPG) module development are part of the WebSphere Development Studio for iSeries license program.

5.1.2 Composition planning and design

The StoreSales XML document is the first XML document that required composition at the store level of our A^2BCM company scenario. In Chapter 3, “Using SQL to compose XML” on page 33, we provide an SQL-based solution to generate XML from our database tables for the StoreSales XML document. Using RPG, we provide another programmatic solution that shows how to generate the same XML document using RPG, embedded SQL, and the CGI toolkit.

Since this composition is for several stores to report their data, each contained in a separate XML document, we require input parameters to be passed to the composition modules to provide flexibility and reuse of coded solutions. To accomplish this, we provided a CMD interface to our RPG module, along with an associated help panel (PNLGRP). This interface allows the same composition module to be used for any number of outbound StoreSales XML documents that potentially reside in different integrated file system directories with different document names. In addition, a CLLE program was developed to validate the user input prior to processing to prevent any errors in user input. This approach grants us flexibility in reuse of a one-coded solution for any number of document locations and naming conventions that are provided.

The biggest advantage of using the CGI toolkit is that it removes the complexity of building an HTML interface, or in our case, an XML document, and publishing it. One service program supplied in the toolkit does all the work and provides simple procedure interfaces to access the features supplied by CGI. The developer can just concentrate on business logic required to generate an XML document.

5.1.3 Composition coding

In this section, we explain the program flow, the different modules, and the Extensible Stylesheet Language (XSL) document coding.

Program flow

Figure 5-2 illustrates the processing flow of the StoreSales XML document composition.

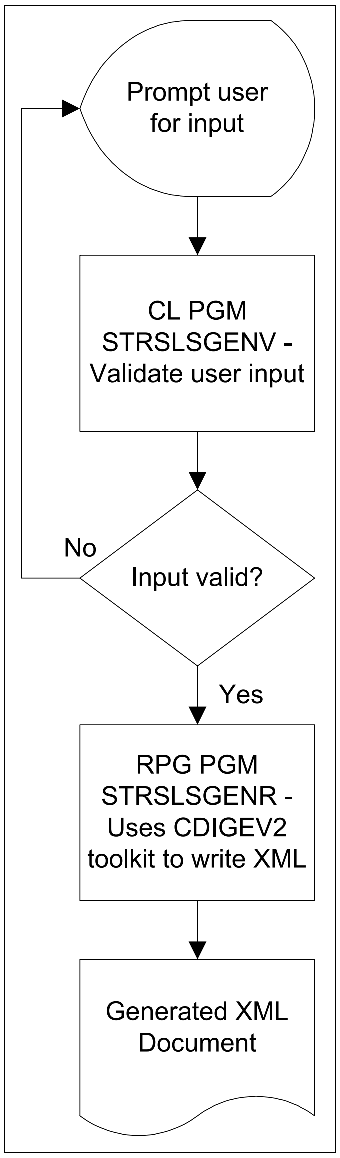

Prompt user for input
CL PGM STRSLSGENV - Validate user input
Input valid?
No
Yes
RPG PGM STRSLSGENR - Uses CDIGEV2 toolkit to write XML
Generated XML Document

Figure 5-2 StrSlsGenr program flow

Required modules

The following modules were created for composition processing via RPG:

- STRSLSGENV CLLE to perform command parameter validity checking
- STRSLSGENR SQLRPGLE to compose an XML document from our database using the CGI toolkit
- STRSLSGENR CGI script template that defines our XML document to be produced
- STRSLSGENR CMD to provide a user interface for input variables
- STRSLSGENR PNLGRP to provide for help text for each command parameter

The source code for the associated CMD, PNLGRP, and CLLE program, as well as all other items, is available in the additional materials that are available for download. Module generation instructions are contained in each of the sample source members. For information about downloading this code, see Appendix A, “Additional material” on page 207.

Input documents

To begin our composition process, we create a member in the QXMLSRC source physical file to contain our CGI XML template. This source member is a representation of our desired XML document that contains literals and section and data tags used by the CGI toolkit. The developer codes how the XML document should look when it is finished along with the appropriate CGI controls and saves the template when complete.

Example 5-1 shows the CGI XML template for the StoreSales XML document.

Example 5-1 StoreSales CGI XML template

```
/$Header                                                   1
<?xml version="1.0" encoding="UTF-8"?>
/$StoreSales
<StoreSales date="/%salesdate%/"                           2
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="StoreSales.xsd">
 <StoreId>/%storeid%/</StoreId>
 <Transactions>                                            3
/$Transaction
  <Transaction type="/%type%/">
/$SalesItem
   <SalesItem>
    <Brand name="/%brand%/"/>
    <Name>/%name%/</Name>
    <Currency>/%currency%/</Currency>
    <Amount>/%amount%/</Amount>
   </SalesItem>
/$EndTransaction
  </Transaction>
/$Trailer
</Transactions>
</StoreSales>
```

The section tag in line 1 of Example 5-1 instructs the CGI toolkit what to output to the memory buffer established by the toolkit when the CGI **`WrtSection`** procedure is called. The memory buffer contains a work image of the final XML document until it is written to a stream file using the CGI **`WrtHtmlToStmf`** procedure. All source in the template that follows a section tag (prefixed by the /$ qualifier) up to the next section tag found is written to the memory buffer when the **`WrtSection`** procedure is called. A section tag is case sensitive when referenced in your HLL.

The data tag in line 2 instructs the CGI toolkit to substitute in its place the actual data contained in an HLL variable when the CGI **`UpdHTMLVar`** procedure is called. In Example 5-1, the data tag `/%type%/` (data tags are prefixed/suffixed by /% and %/ respectively) might contain SALE in our XML document if our HLL variable *type* contained SALE. A data tag is case sensitive when referenced in your HLL.

The literal at line 3 is actual XML text that we want placed in the document at output time. It is output as typed in our source member template.

Input tables

Table 5-1 shows the existing Transactions table that we use to compose our XML document.

Table 5-1 Transaction table

Column name	Data type	Length	Allocated
Transaction Id	Integer		
Transaction Time	Timestamp		
Transaction Type	VarChar	30	10

Table 5-2 shows the existing SalesItem table that we use to compose our XML document. These table layouts, along with the CGI XML template, give us all the input that is required to begin our mapping to compose our XML document.

Table 5-2 SalesItem table

Column name	Data type	Length	Allocated
Transaction Id	Integer		
SalesItem Id	Integer		
ItemName	VarChar	80	50
BrandName	VarChar	80	50
Currency	VarChar	30	10
Amount	Decimal	9.2	

Table 5-3 shows a mapping of the Transactions and SalesItem table columns to the StoreSales XML element and attribute names.

Table 5-3 Column name to XML name

Column name	XML name
User date input	StoreSales/Date
User store id input	StoreSales/StoreId
Transaction.transaction type	StoreSales/Transactions/Transaction/type
SalesItem.brandname	StoreSales/Transactions/Transaction/SalesItem/Brand/name
SalesItem.currency	StoreSales/Transactions/Transaction/SalesItem/Currency
SalesItem.amount	StoreSales/Transactions/Transaction/SalesItem/Amount

Header specifications coding

The coding of the SQLRPGLE module begins by specifying a binding directory in our control specifications. Since we use the procedures supplied in the CGI toolkit service program CGISRVPGM2 in our module, we must specify the binding directory that points to the location of the service program object. We add the following BndDir code to our H spec:

```
h BndDir('CGIDEV2/TEMPLATE2')
```

Data definition specifications coding

Example 5-2 shows the SQLRPGLE source for the data definition specifications (DDS).

Example 5-2 StrSlsGenr SQLRPGLE DDS

```
D STRSLSGENR      pr                  extpgm('STRSLSGENR')        1
D Input                       33
D Output                     163
D CodePage                     5  0
D Debug                        4

D STRSLSGENR      pi
D Input                       33
D Output                     163
D CodePage                     5  0
D Debug                        4

/Copy cgidev2/qrpglesrc,PrototypeB                                2

/Copy cgidev2/qrpglesrc,usec

D  ExecuteSql     PR          10i 0                               3
D                           5000       value

D QRtvJOBI        PR                  Extpgm('QUSRJOBI')          4
D  rtnData                            LIKE(QUSI0400)
D  rtnLen                     10I 0 Const
D  odFormat                    8A   Const
D  JobName                    26A   Const
D  JobId                      16A   Const
D  API_Error                          Like(ApiError)

D  ApiError       DS                  Inz
D   errDSLen                  10I 0 Inz(%size(apiError))
D   errRtnLen                 10I 0 Inz
D   errMsgID                   7A   Inz(*ALLX'00')
D   errReserved                1A   Inz(X'00')
D   errMsgData                64A   Inz(*ALLX'00')

/Copy qsysinc/qrpglesrc,qusrjobi

Dv1_ds            ds                   dim(1000) qualified        5
Dtransid                       9b 0
Dtranstime                     z
Dtranstype                    30    varying
Dstrtransid                    9b 0
Dsalesitemid                   9b 0
Ditemname                     80    varying
Dbrandname                    30    varying
Dcurrency                     30    varying
Damount                        9p 2

Dv1_null_ds       ds                  qualified dim(1000)
D  v1_null_value               5i 0 dim(4)
```

```
D psds          sds                                                6
D  psdsdata                 429
D  psdsjobnam                10    overlay(psdsdata:244)
D  psdsjobusr                10    overlay(psdsdata:254)
D  psdsjobnbr                 6    overlay(psdsdata:264)
```

We begin by coding a prototype (PR) and procedure interface (PI) for the CMD user input data at line 1 in Example 5-2. Since we allowed for variable user input to be supplied to our composition module, this code passes the user input from the CMD as input to the module after validation is completed.

We must provide input for the following parameters:

- INPUT
  - The schema name that contains our tables
  - The sale date to be used to select our rows contained in our tables that we are reporting for and to be inserted into our XML document
  - The store ID to be inserted in our XML document for identification purposes
- OUTPUT
  - The directory name in which the XML document will be stored after generation
  - The name of the generated XML document
- CODEPAGE
  - The code page used for the generated XML document as stored on the integrated file system
- DEBUG
  - A debug flag to control logging of the CGI generation command script

Now that the CMD interface is completed, we copy in the provided prototype and data structure copy books from the CGI toolkit as indicated at line 2. This saves us from coding all the required structures to interface with the toolkit.

Since we are using SQL to create our Aliases over our database tables, we must provide a prototype for our SQL procedure named ExecuteSql (see line 3).

The CGI toolkit requires that the coded character set identifier (CCSID) be set to 37 as a default. To determine the job CCSID at time of change, we use the RTVJOB API to store the job's CCSID as shown at line 4. This code is for the QRTVJOBI prototype, error data structure, and returned data types.

We must provide a data structure that will contain our database table rows when they are retrieved from our input tables (Transactions and SalesItem; see line 5). Because we are performing a block fetch for 1000 rows, we code our data structure to be dimensional for 1000 elements to hold the retrieved rows. Since it is defined as dimensional, it must also be qualified. We define the columns with the same attributes as in our Transactions and SalesItem tables and in the order related to the sequence by which the tables were joined. The accompanying v1_null_ds is defined to hold the null values, if any, that our columns may contain in the event that we want to use null processing.

Our last data structure code, as indicated at line 6, is for the Program Status Data Structure (PSDS), which will contain job information that is used by the CGI toolkit debugger processor.

Calculation specifications coding

We begin our coding by setting up our job and CGI toolkit environment to function properly. Example 5-3 shows the SQLRPGLE source to set up our environment for processing.

Example 5-3 StrSlsGenr SQLRPGLE setup processing

```
callp qrtvjobi(                                                    1
       QUSI0400:
       %Len(QUSI0400):
       'JOBI0400':
       '*':
       ' ':
       apierror);

if QUSCCSID07 <> ccsid;
   eval rc = doCmd('chgjob ' +
                  %trim(qusjnbr05) + '/'  +
                  %trim(qusun05) + '/'  +
                  %trim(qusjn06) + ' CCSID(' +
                  %trim(%editc(ccsid:'Z')) + ')');
 endif;

ClrHtmlBuffer();                                          2

CallP     GetHtml('QXMLSRC':'XMLREDBOOK':'STRSLSGENR');

SqlStm = 'drop alias qtemp/TRANS_alias';            3
eval sqlcod = ExecuteSql(SqlStm);

SqlStm = 'drop alias qtemp/SALES_alias';
eval sqlcod = ExecuteSql(SqlStm);

SqlStm = 'create alias qtemp/TRANS_alias for ' +
            %trim(schema) + '/TRANS00001';
eval sqlcod = ExecuteSql(SqlStm);

SqlStm = 'create alias qtemp/SALES_alias for ' +
            %trim(schema) + '/SALESITEM';
eval sqlcod = ExecuteSql(SqlStm);

C/exec sql                                                         4
C+ declare STORESALES cursor for
C+     select *
C+     from ( TRANS_alias left outer join
C+            SALES_alias
C+            on  TRANS_alias.trans00001 =
C+                SALES_alias.trans00001 )
C+     where :date = char(date(TRANS_alias.TransactionTime), ISO)
C/end-exec

C/exec sql
C+ open STORESALES
C/end-exec
```

```
dow sqlstt = '00000';                          5
 /end-free

C/exec sql
C+ fetch next
C+  from STORESALES
C+  for 1000 rows
c+ into :v1_ds :v1_null_ds
C/end-exec
```

Our job environment CCSID must be set to 37 (EBCDIC) for processing to work correctly as indicated at line 1 in Example 5-3. We first retrieve the current jobs attributes and settings using the QRTVJOBI procedure. If the retrieved CCSID is not 37, we then set the job's CCSID to 37 using the Change Job (CHGJOB) command executed by the **doCmd** procedure (supplied in the CGIDEV2 toolkit). This is required to ensure that our default CCSID for the job is EBCDIC.

We can now begin to code the integration of the CGI toolkit into our RPG module, which we do at line 2. The CGI toolkit performs the housekeeping tasks that are required to generate XML after the interface is established.

The process flow for using the CGI toolkit is as follows:

1. Clear the current state of the memory buffer.
2. Load the XML template to memory.
3. Update the XML template in memory.
4. Write the XML template from memory to a stream file in the integrated file system.

First we must clear the HTML memory buffer to prevent any issues with previously loaded templates, and then we must load our CGI XML template. The clearing of the HTML buffer provides peace of mind that we can start fresh with the buffer flushed. We use the **ClrHtmlBuffer** procedure to clear the buffer and use the **GetHtml** procedure to load our CGI XML template. The **GetHtml** procedure requires three parameters when called:

- The source file that contains the CGI XML template
- The schema that contains the source file
- The template name (source member)

The load process places the template source member in a memory buffer so that we can manipulate the contents of the data tags and write the sections using the CGI procedures supplied in the toolkit.

Now that our CGI toolkit environment is established, we can begin the process of populating our XML template loaded in memory with data from our database tables (Transactions and SalesItem). We provide the capability to run composition for multiple stores, whose sales data potentially reside in different schemas. Therefore, we must override our tables prior to our SQL data retrieval processing so that the correct data is retrieved for the store for which we are generating the XML document as indicated at line 3.

We first remove any existing aliases for our tables that may exist using the SQL Drop Alias for each alias. Then we use the SQL Create Alias to create a path to our Transactions and SalesItem tables using the user supplied schema name that is provided at initiation. The Alias performs a similar function to the Override Database Function (OVRDBF) command to create a path to a database table or to a database table member.

After the tables are set to the correct location, we build a cursor that joins the Transactions and SalesItem tables, by using the Transaction Id key that is contained in both tables. We select only the data that contains a transaction date equal to the sales date specified by the user during program initialization.

We build an SQL cursor to be used for our block fetch retrievals of data from our two tables. Using our previously created alias names for each table, we perform a left outer join between our two tables. The parent table, Transactions, is joined to our child table, SalesItem, by the like key contained in both. This key is the Transaction Id column that is represented by the trans00001 internal column name. It allows us to do one fetch to retrieve both sets of data in a denormalized fashion for our composition module.

We allow for the user-supplied sales date input that instructs us as to which date the sales are to be retrieved for from our databases. Therefore, we use the Where clause that is coded so that the contents of the variable *:date*, which contain the user-specified sales date, are used to select only transactions that were time stamped as having occurred for the specified date. Our next task is to open the cursor to make it available to our program when it is declared.

Example 5-4 shows the SQLRPGLE source that our XML composition processing using data retrieved from our tables.

Example 5-4 StrSlsGenr SQLRPGLE XML processing

```
dow sqlstt = '00000';                                               1

   C/exec sql                                                       2
   C+ fetch next
   C+  from STORESALES
   C+  for 1000 rows
   C+ into :v1_ds :v1_null_ds
   C/end-exec

   if sqlstt <> '00000';
      if sqlstt = '02000';
         leave;
      endif;
   endif;

   for i1 = 1 to sqler3;                                            3

      if  header_done = *off;                                       4
         eval header_done = *on;
         Callp WrtSection('Header');
         CallP UpdHTMLVar('salesdate': date);
         CallP UpdHTMLVar('storeid': store);
         Callp WrtSection('StoreSales');
      endif;

      if  v1_ds(i1).transid <> sv_transid;                          5

         if sv_transid <> *zeros;
            Callp WrtSection('EndTransaction');
         endif;

         eval sv_transid = v1_ds(i1).transid;

         CallP UpdHTMLVar('type' : v1_ds(i1).transtype);
         Callp WrtSection('Transaction');
      endif;

      if %scan('&': v1_ds(i1).brandname) > *zero;                6
         eval i2 = %scan('&': v1_ds(i1).brandname) -1;
         eval wrk_brandname = %subst(v1_ds(i1).brandname:1:i2) +
                                amp +
                                %subst(v1_ds(i1).brandname:i2+2);
         CallP UpdHTMLVar('brand' : wrk_brandname);
      else;
         CallP UpdHTMLVar('brand' : v1_ds(i1).brandname);
      endif;
      CallP UpdHTMLVar('name' : v1_ds(i1).itemname);
      CallP UpdHTMLVar('currency' : v1_ds(i1).currency);
      CallP UpdHTMLVar('amount' : %char(v1_ds(i1).amount));

      Callp WrtSection('SalesItem');
   endfor;
enddo;

Callp WrtSection('EndTransaction');                              7
CallP WrtSection('Trailer');

Callp WrtHtmlToStmF(ifs_document:ifs_code_page);
```

With our cursor being declared and opened, we can now retrieve our data from our database tables. By placing our fetch processing within a *do* loop as we do in line 1 in Example 5-4, we can retrieve all the rows from our tables until the end of data is complete. Using SQL Fetch at line 2, we retrieve up to 1000 rows in a block and populate our defined data structures for our tables and null values as shown in Example 5-4. We provide limited error checking to account only for end of data reached (sqlstt = 0200), so that we may leave our data retrieve loop that we established.

With our XML document being loaded into memory, and our data retrieved from disk and stored in our data structure, we can now populate the XML template and create our XML document. We establish a *for* loop, as indicated at line 3, that will process all of the dimensions (elements) of our populated data structure. SQL provides the actual number of rows that were retrieved from a fetch and stores the value as part of the SQL Communications Area (SQLCA) data structure in the SQLER3 column. Using that value, we can control our repeat processing to account for all the data that is retrieved and end our *for* loop when all data has been processed that was retrieved during the current fetch.

The Header section of our XML document only needs to be populated and written once to our XML document (see line 4). We control this processing based on a user-defined flag (header_done) that indicates that the Header processing has already been executed when it is set within our Header processing section. After we set the user-defined flag, we write our Header section to our XML buffer using the **WrtSection** procedure. We then set our StoreSales attribute and element values, date, and store ID respectively. We do this by calling the **UpdHtmlVar** procedure for each data tag in our CGI XML template and passing the user-supplied sales date and store ID provided by our user through our CMD interface prototype and procedure. With our data tags updated, we can now write our StoreSales section to memory using the **WrtSection** procedure.

With our static Header section completed, we begin the process of writing the repeating element data of our XML document. Each SalesItem that is reported belongs to a Transaction that has a unique Transaction Id (see line 5). Since each Transaction element has a beginning and ending, we must then account for the change of a Transaction Id and save the change in order to write the ending element for each beginning element before another Transaction element is started. We do so by defining a save Transaction Id variable and use for comparison to our current Transaction Id that is being processed. If this is not the first time that it is processed, we write the ending Transaction element using the **`WrtSection`** procedure. Next we update the Transaction type attribute again using the **`UpdHtmlVar`** procedure and write our Transaction section with **`WrtSection`**.

The SalesItem section contains an additional processing concern that we must allow for (see line 6). The BrandName column data can potentially contain an ampersand (&) embedded in the data. The ampersand, which is similar to an apostrophe (’), greater than (>), or less than (<) sign, and a few others are reserved character entities that instruct XML and HTML how to present data. For more information and examples of character entities, see:

`http://www.w3schools.com/tags/ref_entities.asp`

Therefore we must substitute any embedded, reserved character found in our data to the character entity that replaces it. Using the RPG Built In Function (BIF) %SCAN and %SUBST, we search our BrandName data to find if there is an embedded ampersand, and by manipulating the data, rebuild the data to be acceptable to XML and HTML. If our BrandName data contains ‘ITSO & Electronics’ after replacement, the data can contain ‘ITSO &amp; Electronics’.

The remaining SalesItem data tags are updated with the related data and we write our SalesItem section using the **`WrtSection`** procedure.

All the previous processing occurs within our *for* and *do* loops that we established earlier and will repeat for each element stored in our dimensional data structure for all rows contained in our tables.

At this point, we only need to write the ending Transaction element to close our last Transaction block, write our Trailer section that closes our XML document, and generate our XML document using the **`WrtHtmlToStmf`** procedure. The **`WrtHtmlToStmf`** procedure requires the following two parameters when called:

- Integrated file system document name and location
- Integrated file system document code page

Both of these parameters were provided to our composition module by our user through our CMD interface prototype and procedure.

Note: Refer to 5.1.6, “Considerations for RPG composition” on page 74, for known limitations that affect your coding design when using the **`WrtHtmlToStmf`** procedure.

We end our coding by removing the temporary Aliases that we created, resetting up our job environment, and clearing our activation group.

Example 5-5 shows the SQLRPGLE source for cleanup processing.

Example 5-5 StrSlsGenr SQLRPGLE cleanup processing

```
SqlStm = 'drop alias qtemp/TRANS_alias';
eval sqlcod = ExecuteSql(SqlStm);

SqlStm = 'drop alias qtemp/SALES_alias';
eval sqlcod = ExecuteSql(SqlStm);

if QUSCCSID07 <> ccsid;
eval returncode = doCmd('chgjob ' +
%trim(qusjnbr05) + '/'  +
%trim(qusun05) + '/'  +
%trim(qusjn06) + ' CCSID(' +
%trim(%editc(qusccsid07:'Z')) + ')');
endif;

callp ceetrec( );
```

Example 5-6 shows the SQLRPGLE source for our ExecuteSql procedure.

Example 5-6 StrSlsGenr SQLRPGLE ExecuteSql procedure

```
P ExecuteSql      b

D ExecuteSql      pi           10i 0
D  SqlStmI                   5000     value

C/EXEC SQL
C+ Execute Immediate :SqlStmI
C/END-EXEC

C                   return    SqlCod
P ExecuteSql      e
```

5.1.4 Deployment of RPG composition modules

All CMD, CLLE, PNLGRP, and SQLRPGLE modules, as well as our QXMLSRC source file member, can be saved using the Save Object (SAVOBJ) command and restored on a target system using the Restore Object (RSTOBJ) command.

The following example shows how to deploy the StoreSales composition modules from the source system to a target system:

1. On the source system, create a save file to store the saved objects:

   ```
   CRTSAVF SAVF(XMLREDBOOK/STRSLSMODS)
   ```

2. On the source system, save the StoreSales composition modules to the STRSLSMODS save file:

   ```
   SAVOBJ OBJ(STRSLSGEN* QXMLSRC) LIB(XMLREDBOOK) DEV(*SAVF)
   SAVF(XMLREDBOOK/STRSLSMODS) FILEMBR((QXMLSRC (STRSLSGENR)))
   ```

3. On the target system, create a save file to receive the saved objects to be deployed :

   ```
   CRTSAVF SAVF(XMLREDBOOK/STRSLSMODS)
   ```

4. On the target system, restore the StoreSales composition modules to the target schema:

   ```
   RSTOBJ OBJ(*ALL) SAVLIB(XMLREDBOOK) DEV(*SAVF) SAVF(XMLREDBOOK/STRSLSMODS)
   ```

5.1.5 Composition execution

Now that we have coded and deployed our solution, we must execute the modules and compose our XML document. The command must be run for each XML document that requires composition from our Transactions and SalesItem tables that reside in separate store schemas.

The STRSLSGENR CMD is used to initiate the composition process for each StoreSales XML document that we want to generate. From a command line, type the following command and press F4:

```
STRSLSGENR
```

This command uses the following parameters:

- INPUT

  This parameter is composed of three parts:

  – The schema name that contains the tables

  – The sales date to be used to select the rows contained in the tables for which we are reporting

  – The store ID to be inserted into the XML document for identification purposes

- OUTPUT

  This parameter is composed of two parts, a directory and a document. We enter the integrated file system directory that contains the XML document that we want to compose. The name of the document is the actual XML document that we are decomposing.

- CODEPAGE

  This is the code page used for the generated XML document as stored in the integrated file system.

The following example shows how to enter the command on a command line after the variable user input is completed:

```
STRSLSGENR INPUT('Store7' '2006-04-06' '7')
           OUTPUT('/XMLRedbook/StoreXML' 'Store706.xml')
           CODEPAGE(819)
           DEBUG(*NO)
```

After the compose has completed for the document specified, the command line returns. We validate that the XML data was composed correctly. We also validate that the XML document that resides in the integrated file system directory, with the name specified, exists and is populated with the sales data for the specified day, for the appropriate store.

Generated StoreSales XML document

Example 5-7 shows the generated Store706 XML document.

Example 5-7 Store706 XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<StoreSales Date="2006-04-06"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="StoreSales.xsd">
   <StoreId>7</StoreId>
   <Transactions>
      <Transaction type="SALE">
         <SalesItem>
            <Brand name="Pepsi" />
            <Name>Mt. Dew 20 oz.</Name>
            <Currency>USD</Currency>
            <Amount>1.19</Amount>
         </SalesItem>
         <SalesItem>
            <Brand name="General Electric" />
            <Name>60 watt bulbs, 4 pk.</Name>
            <Currency>USD</Currency>
            <Amount>.98</Amount>
         </SalesItem>
      </Transaction>
   </Transactions>
</StoreSales>
```

5.1.6 Considerations for RPG composition

As of the writing of this redbook, take into account the following considerations when using the CGI toolkit procedures to compose an XML document:

- The size of the memory buffer associated with the CGI toolkit is 16 Mb. If your XML document that is being generated can potentially exceed this size, the limitation causes a problem.
- Each call to the **`WrtHtmlToStmf`** procedure recreates the document that is specified by removing and creating the document each time it is called.

Since it is logical to perform the call to the **`WrtHtmlToStmf`** procedure more often to clear the buffer and not exceed the 16 Mb limitation, the second limitation regarding **`WrtHtmlToStmf`** prevents that from happening. There are two solutions to these considerations:

- Provide smaller tables to the composition processor that creates smaller XML documents. You can accomplish this by splitting input tables to smaller sizes and process multiple times. The disadvantage to this is that, if only one XML document can be used, you must put multiple documents together before using it.
- Modify the CGI toolkit procedure to remove the limitations that were noted previously. The good news is that since the CGI toolkit is open source, when you download and install the CGIDEV2 toolkit library on our system, with the source code included. In the CGIDEV2 schema, you can find the QRPGLESRC source file that contains the modules that are used to build the CGISRVPGM2 service program. In addition, a command (COMPILE) is included to recreate the service program.

The module that requires modification is the XXXWRKHTMLmember. The **`WrtHtmlToStmf`** procedure needs modifications to create and open the integrated file system document once, and then allow for multiple calls to it so that data can be appended to the document.

5.2 Decomposing XML using RPG

As of the V5R4 of i5/OS, the ILE RPG compiler has native operation codes and built-in functions that facilitate decomposition of an XML document into data structures that are defined in the program using a non-validating parser. For more information about the XML parser that is used by ILE RPG, refer to Chapter 11 in the *WebSphere Development Studio ILE RPG Programmer's Guide,* SC09-2507. Using this functionality, we can decompose our CorpSales XML document.

The advantage of using the new operation codes and built-in functions added to RPG is that, when the hierarchy of the XML document is replicated using data structures in our RPG program code, the decomposition of the XML data and the population of the data structure fields is done automatically and efficiently without the developer manipulating storage to get the XML data into a workable form. The complexity has been removed from the XML decomposition process, so the developer now can concentrate on the business logic that is required to implement the XML data. There are no pointers, API interfaces, or buffer manipulation; there is only straightforward RPG coding.

This section provides a solution for step 6 from our fictional scenario (see Figure 5-3).

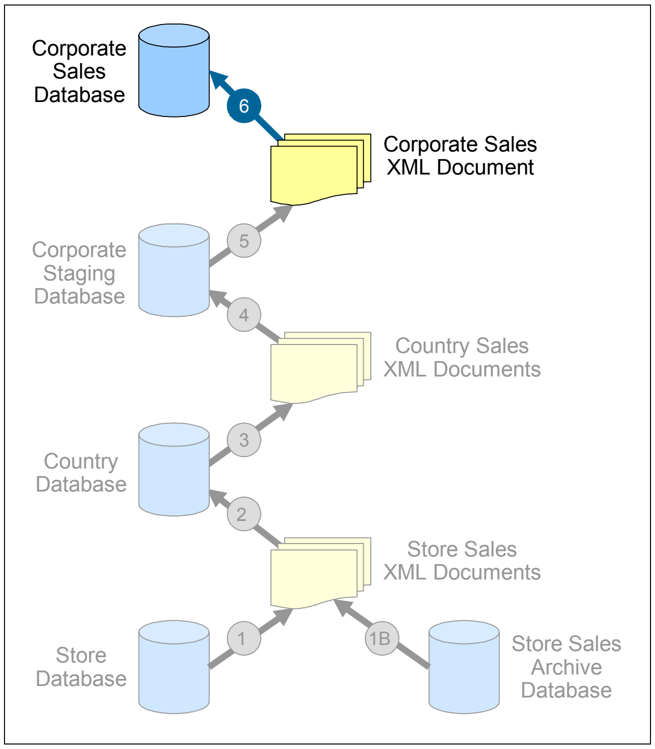

Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 5-3 Scenario step 6

5.2.1 Prerequisites

In the examples in this section, we use ILE CL, CL user commands, and PNLGRPs. To code the examples in this section, you must meet the following requirements:

- Verify that you have installed WebSphere Development Studio - option 21 (5722WDS) on your System i machine.
- Verify that you have installed WebSphere Development Studio - option 34 (5722WDS) on your System i machine that is needed for the example done in RPG to compose XML documents.
- Verify that your default version of Java is 1.4; you can do this by entering the following command in Qshell:

  ```
  java -version
  ```

Note: CLLE, CMD, UIM PNLGRP, and RPG module development are part of the WebSphere Development Studio for iSeries license program.

5.2.2 Decomposition planning and design

The CorpSales XML document is the final XML document that requires decomposition at the corporate level of our A^2BCM company scenario. Since this decomposition is for several countries that are reporting their sales data, each contained in a separate XML document, we require input parameters to be passed to the decomposition processor.

We accomplish this by providing a CMD interface to our RPG module along with an associated help panel (PNLGRP). This flexibility allows the same decomposition processor to be used for any number of inbound CorpSales XML documents that potentially reside in different integrated file system directories or have different document names. In addition, a CLLE program was developed to validate the user input prior to processing to prevent any errors in user input at the point of origination. This approach grants us the flexibility to reuse a one-coded solution for any number of document locations and naming conventions that are provided.

5.2.3 Decomposition coding

In this section, we explain the program flow, the different modules, and the XML document decomposition.

Program flow

Figure 5-4 illustrates the processing flow of the CorpSales XML document decomposition.

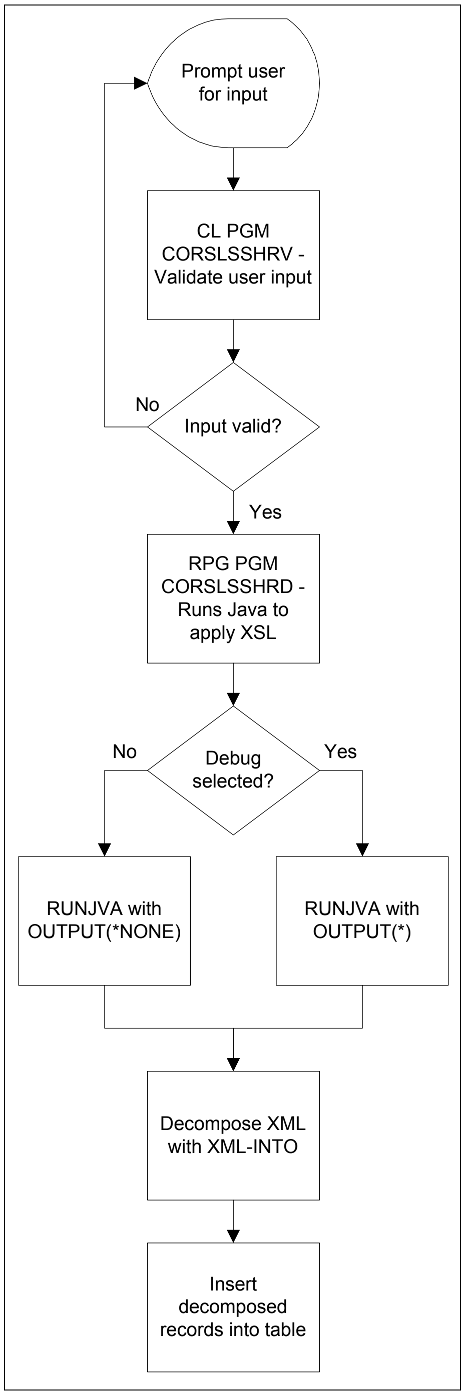

Prompt user for input
CL PGM CORSLSSHRV - Validate user input
Input valid?
No
Yes
RPG PGM CORSLSSHRD - Runs Java to apply XSL
Debug selected?
No
Yes
RUNJVA with OUTPUT(*NONE)
RUNJVA with OUTPUT(*)
Decompose XML with XML-INTO
Insert decomposed records into table

Figure 5-4 CorSlsShrd program flow

Required modules

The following modules were created for decomposition processing via RPG:

- CORSLSSHRV CLLE to perform command parameter validity checking
- CORSLSSHRD SQLRPGLE to decompose an XML document and populate the CorpSales table
- CORSLSSHRD CMD to provide a user interface for input variables
- CORSLSSHRD PNLGRP to provide for help text for each command parameter

The source code for the associated CMD, PNLGRP, and CLLE program, as well as all other items, is available in the additional materials available for download. For information about downloading this code, see Appendix A, “Additional material” on page 207. Module generation instructions are contained in each of the sample source members.

Input documents

Example 5-8 and Example 5-9 show the inbound XML and document type definition (DTD) documents respectively. These two inputs are the basis for the mapping of the XML document to our target CorpSales table using the RPG operation code and built-in functions to decompose our XML document. We begin by reviewing our CorpSales XML document to determine the tree structure of the XML XPATH nodes and note any repeating or wrapper elements.

Example 5-8 shows the CorpSales XML document.

Example 5-8 CorpSales XML document

```
<?xml version="1.0" encoding="UTF-8" ?>
<CorpSales Date="2006-04-06">                          3
   <CountryInfo>
      <Name>USA</Name>                                 4
   </CountryInfo>
   <SalesByBrand>
      <Brand>                                          5
         <Name>Bosch</Name>
         <Sales>
            <Currency></Currency>
            <Amount></Amount>
         </Sales>
         <Returns>
            <Currency>USD</Currency>
            <Amount>39.95</Amount>
         </Returns>
      </Brand>
   </SalesByBrand>
</CorpSales>
```

Example 5-9 shows the CorpSales DTD document.

Example 5-9 CorpSales DTD document

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT CorpSales (CountryInfo,SalesByBrand) >         3
<!ATTLIST CorpSales
  Date CDATA #IMPLIED>
<!ELEMENT CountryInfo (Name) >                           4
<!ELEMENT Name (#PCDATA)>
<!ELEMENT SalesByBrand (Brand+) >                        5
<!ELEMENT Brand (Name,Sales?,Returns?) >
<!ELEMENT Sales (Currency,Amount) >
<!ELEMENT Returns (Currency,Amount) >
<!ELEMENT Currency (#PCDATA)>
<!ELEMENT Amount (#PCDATA)>
```

Target table

Table 5-4 shows the existing CorpSales table in which we will decompose our XML document. This table layout, along with the previous examples of the XML and DTD documents, gives us all the input that is required to begin a mapping in our RPG program to decompose our XML document. Refer to 10.2, “Analyzing the XML document structure and determining the mapping” on page 157, for more information about the XML document structure and mappings.

Table 5-4 CorpSales table

Column name	Data type	Length	Allocated
CountryName	VarChar	50	30
BrandName	VarChar	80	50
SalesDate	Date		
Type	VarChar	30	10
Currency	VarChar	30	10
Amount	Decimal	9.2	

Table 5-5 shows a mapping of the XML element and attribute names to our CorpSales table columns.

Table 5-5 XML names to CorpSales columns

XML name	Column name
CorpSales/Date	salesdate
CorpSales/CountryInfo/Name	countryname
CorpSales/SalesByBrand/Brand/Name	brandname
CorpSales/SalesByBrand/Sales/Currency	currency
CorpSales/SalesByBrand/Sales/Amount	amount
CorpSales/SalesByBrand/Returns/Currency	currency
CorpSales/SalesByBrand/Returns/Amount	amount

Data definition specifications coding

Example 5-10 shows the SQLRPGLE source for the DDS.

Example 5-10 CorSlsShrd SQLRPGLE Data Definition specifications

```
D CORSLSSHRD     PR                   extpgm('CORSLSSHRD')        1
D Input                     163
D Classpath                  80
D Debug                       4

D CORSLSSHRD     PI
D Input                     163
D Classpath                  80
D Debug                       4

CorpSales_ds   e DS                  extname(CorpSales)           2
```

```
d CorpSales       DS                     QUALIFIED               3
d  Date                        10d
d  CountryInfo                           likeds(CountryInfo)     4
d  SalesbyBrand                          likeds(SalesByBrand)

d CountryInfo     DS                     QUALIFIED
d  Name                        50a

d SalesByBrand    DS                     QUALIFIED
d  Brand                                 likeds(Brand)
d                                        dim(10)                 5

d Brand           DS                     QUALIFIED
d  Name                        80a
d  Sales                                 likeds(Sales)
d  Returns                               likeds(Returns)

d Sales           DS                     QUALIFIED
d  Currency                    30a
d  Amount                       9p 2

d Returns         DS                     QUALIFIED
d  Currency                    30a
d  Amount                       9p 2
```

We begin by coding a prototype (PR) and procedure interface (PI) for the CMD user input data at line 1 in Example 5-10. Since we allowed for variable user input to be supplied in our decomposition module, this code will pass the user input from the CMD to the input of module after the validation is completed.

We must provide input for the following parameters:

- The directory name that contains the XML document and XSL that transforms the XML document to remove white space, empty elements, and attributes
- The name of the XML document itself
- The directory used as the Classpath to locate the Transform class
- A debug flag to control the interactive display of the Transform class execution messages

Now that the CMD interface is completed, mapping the XML document hierarchy to data structures begins. Use the following guidelines when mapping the XML document to the data structures:

- Beginning with the Root element of the XML document, code your data structure hierarchy by following a top-down flow.
- Match the XML element and attribute names to the data structure, data structure array, and field names.
- The data structures must be qualified.
- XML elements that repeat are defined as dimensional (DIM) with a maximum anticipated number of elements specified.

The CorpSales_ds data structure is an external data structure that is derived from our target table that will be populated with the decomposed XML data (see line 2 in Example 5-10).

The concept illustrated in Example 5-8 on page 78, Example 5-9 on page 78, and Example 5-10 on page 79, with 3 corresponding to 3 and so on, shows the mapping of XML element and attribute names via the DTD to the SQLRPGLE data structure, data structure array, and field names.

Since the Brand (see line 5) is the only element that can repeat, as indicated in the DTD with a plus sign (+) next to the element name, the Brand data structure is set up as an array with 10 elements allowed.

Following the hierarchy of the XML document, you can see how the SQLRPGLE code replicates the XML tree from top to bottom. Because of the nesting of our data structures that replicate the XML tree and the fact that the Brand data structure is defined as dimensional, each data structure name must be defined as qualified.

CorpSales is the root element of the XML document, so the CorpSales is the first coded data structure. The CorpSales element contains the Date attribute that is defined as the Date field in the data structure. The CountryInfo element and SalesbyBrand element are both defined as data structures that are linked using the LIKEDS keyword that links to the CountryInfo and SalesbyBrand data structures respectively. Each XML element and attribute is provided for either as a data structure or as a field within a data structure as it is found in the XML document. The SalesbyBrand element, and related data structure, is a wrapper (contains no actual data) that contains the multi-occurrence Brand element and data structure but must be accounted for to preserve the XML tree.

Note: Refer to 5.2.6, “Considerations for RPG decomposition” on page 87, for known considerations that can affect your coding design.

Calculation specifications coding

When reviewing our input XML document, we learned that the document would contain empty elements and attributes that were included during the generation process. As noted in 5.2.6, “Considerations for RPG decomposition” on page 87, empty elements and attributes that contain zero-length numeric, date, time or timestamp fields are not allowed. Knowing this, our document must be transformed to remove those conditions. We accomplish this by applying our RemoveEmpty stylesheet (XSL) to transform the XML document to remove any white space, empty elements, and attributes. The Java Transform Class referred to throughout this redbook is used to apply the RemoveEmpty XSL to the XML document and to produce a transformed XML document for decomposing.

Example 5-11 shows the RemoveEmpty XSL document that is used to transform our input XML document to remove any white space, empty elements, and attributes.

Example 5-11 RemoveEmpty XSL document

```
<?xml version="1.0"?>
<!-- ****************************************************************** -->
<!-- *  This XSL is used to remove white space and empty elements and   * -->
<!-- *  attributes in an XML document.                                  * -->
<!-- ****************************************************************** -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
 xmlns:xalan="http://xml.apache.org/xslt">

<!-- Remove white space -->
<xsl:strip-space elements="*" />

<!-- Remove each empty element and attribute -->
<xsl:template match="@*|node()|text()">
```

```
 <xsl:if test="count(descendant-or-self::node()[.!='' or @*[.!=''] ])>0" >
  <xsl:copy>
  <xsl:apply-templates select="@*[.!='']|node()"/>
  </xsl:copy>
 </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Example 5-12 shows the SQLRPGLE source for the XML decomposition processing.

Example 5-12 CorSlsShrd SQLRPGLE decomposition processing

```
select;                                                                1
when Debug = '*NO';
 cmd = 'RUNJVA CLASS(Transform) +
              PARM(' +
                    apos + %trimr(XMLdocumentin) + apos + ' ' +
                    apos + %trimr(XSLdocument) +   apos + ' ' +
                    apos + %trimr(XMLdocumentout) + apos + ' ' +
                    apos + ' ' + apos + ') +
              CLASSPATH(' +
                    %trimr(XMLclasspath) + ') +
              PROP((java.version 1.4)) +
              OUTPUT(*NONE)' ;

when Debug = '*YES';
 cmd = 'RUNJVA CLASS(Transform) +
              PARM(' +
                   apos + %trimr(XMLdocumentin) + apos + ' ' +
                   apos + %trimr(XSLdocument) +   apos + ' ' +
                   apos + %trimr(XMLdocumentout) + apos + ' ' +
                   apos + ' ' + apos + ') +
              CLASSPATH(' +
                   %trimr(XMLclasspath) + ') +
              PROP((java.version 1.4))';
endsl;

cmdlen = %len(%trim(cmd));

callp qcmdexc(cmd:cmdlen);

xml-into CorpSales  %XML(XMLdocumentout :                              2
                         'doc=file +
                          allowextra=yes +
                          allowmissing=yes +
                          case=any');

eval   salesdate =                                                     3
     CorpSales.date;
eval   count00001 =
     CorpSales.CountryInfo.Name;

for i1 = 1 to 10;                                                      4
```

```
   if CorpSales.SalesByBrand.Brand(i1).name <> *blanks;            5
      eval   brandname =
             CorpSales.SalesByBrand.Brand(i1).name;

      if CorpSales.SalesByBrand.Brand(i1).Sales.currency <> *blanks; 6
         eval   type = 'SALE';
         eval   currency =
                  CorpSales.SalesByBrand.Brand(i1).Sales.currency;
         eval   amount =
                  CorpSales.SalesByBrand.Brand(i1).Sales.amount;
      eval returncode = WriteTbl();
      endif;

      if CorpSales.SalesByBrand.Brand(i1).Returns.currency <> *blanks;
         eval   type = 'RETURN';
         eval   currency =
                CorpSales.SalesByBrand.Brand(i1).Returns.currency;
         eval   amount =
                CorpSales.SalesByBrand.Brand(i1).Returns.amount;

         eval returncode = WriteTbl();
      endif;
   endif;
endfor;
```

We condition our execution of the Java transformation based on the DEBUG parameter specified by the user at program initialization (see section 1 in Example 5-12). Either one of the following statements can be true:

- DEBUG = *NO will condition the RUNJVA command to suppress the display of the Java execution script window.
- DEBUG= *YES will condition the RUNJVA command to display the Java execution script window if it is run in the interactive environment.

The RUNJVA command is executed via the QCMDEXC API to apply the XSL to an XML document for transformation. The variables XMLdocumentin, XSLdocument, XMLdocumentout, and XMLclasspath are set prior to the execution:

- *XMLdocumentin* is the XML document that is to be transformed.
- *XSLdocument* is the XSL that performs the transformation.
- *XMLdocumentout* is the transformed XML document.
- *XMLclasspath* is the directory name where the Transform Class is located.

The QCMDEXC command requires only two parameters:

- Command string to execute
- The length of the command string

The transformed XML document can now be decomposed using the XML-INTO operation code that will decompose the XML document into a data structure that is workable within an RPG module (see section 2).

The format of the XML-INTO operation code is as follows:

- CorpSales is the data structure into which the XML data is decomposed.
- XMLdocumentout is the XML file that is being decomposed and is comprised of the directory and document name.
- doc=file instructs that the XML document to be decomposed is from an integrated file system file versus a string variable.
- allowextra=yes instructs decompose to allow for extra XML elements and attributes existing in the XML document that are not represented in the CorpSales data structure tree.
- allowmissing=yes instructs decompose to allow for the XML document that does not contain data for a represented CorpSales data structure field.
- case=any instructs that the XML naming of elements/attributes can be in mixed case.

Since our XML data is now decomposed into our data structures, we can begin populating our table columns and writing the rows to the table. In our data definition specifications, we created an external defined data structure that is derived from our target table (see line **2** in Example 5-10 on page 79). Mapping can now begin to populate the columns of our target table and then, using the **`WriteTbl`** procedure, insert our rows into our table.

Since each of the data structures is qualified, we must code our Eval statements accordingly. The salesdate eval (see line **3**) shows that the qualified CorpSales.date value is used to populate the salesdate column.

Beginning with the Brand data structure processing (see line **4**), we set up a *for* loop to handle each of the 10 data structure array elements that could have been populated. With Brand as a parent element to Sales and Returns, it will then dictate what happens for each child (Sales and Returns) that follows in the tree. Brand.name is checked for data (see line **5**) to determine if the array element is to be processed because it contains data. Each Sales and Returns data structure is optional so each is checked for data (see line **6**) to determine if it should be processed. After we complete the tree for all data, the **`WriteTbl`** procedure is used to insert the row in our table.

All of this processing occurs within our *for* loop that we established earlier. It will repeat for each element that is stored in our dimensional data structure until all data structure array elements have been processed.

We end our coding by removing the temporary transformed XML document and clearing our activation group.

Example 5-13 shows the SQLRPGLE source for cleanup processing.

Example 5-13 CorSlsShrd SQLRPGLE cleanup processing

```
if %subst(XMLdocumentin:varlen:1) <> '/';
   eval XMLdocumentout = %trim(%subst(Input:3:80)) + '/' +
      %trim(%subst(Input:83:80)) + '.xml';
else;
   eval XMLdocumentout = %trim(%subst(Input:3:80)) +
      %trim(%subst(Input:83:80)) + '.xml';
endif;

cmd = 'RMVLNK OBJLNK(' + apos + %trimr(XMLdocumentout) + apos + ')';
```

```
cmdlen = %len(%trim(cmd));

callp qcmdexc(cmd:cmdlen);

callp ceetrec( );
```

Example 5-14 shows the SQLRPGLE source for the WriteTbl procedure. Since our CorpSales table exists in only one schema, we use *LIBL when our SQL insert is performed as indicated by the line in bold.

Example 5-14 CorSlsShred SQLRPGLE WriteTbl procedure

```
P WriteTbl        b                   export
D WriteTbl        pi            10i 0

/exec sql
C+ insert into CorpSales
C+   (COUNTRYNAME,
C+    BRANDNAME ,
C+    SALESDATE ,
C+    TYPE      ,
C+    CURRENCY  ,
C+    AMOUNT)
C+ values(
C+    :COUNT00001,
C+    :BRANDNAME ,
C+    :SALESDATE ,
C+    :TYPE      ,
C+    :CURRENCY  ,
C+    :AMOUNT)
C+ with nc
C/end-exec

C                   return    sqlcode

P WriteTbl        e
```

5.2.4 Deployment of the RPG decomposition modules

All CMD, CLLE, and PNLGRP modules can be saved using the SAVOBJ command and restored on a target system using the RSTOBJ command. The following example shows how to deploy our CorpSales decomposition modules from our source system to a target system:

1. On the source system, create a save file to store the saved objects:

   ```
   CRTSAVF SAVF(XMLREDBOOK/CORSLSMODS)
   ```

2. On the source system, save the CorpSales decomposition modules to the CORSLSMODS save file:

   ```
   SAVOBJ OBJ(CORSLSSHR*) LIB(XMLREDBOOK) DEV(*SAVF) SAVF(XMLREDBOOK/CORSLSMODS)
   ```

3. On the target system, create a save file to receive the saved objects to be deployed:

   ```
   CRTSAVF SAVF(XMLREDBOOK/CORSLSMODS)
   ```

4. On the target system, restore the CorpSales decomposition modules to the target schema:

   ```
   RSTOBJ OBJ(*ALL) SAVLIB(XMLREDBOOK) DEV(*SAVF) SAVF(XMLREDBOOK/CORSLSMODS)
   ```

The RemoveEmpty XSL document and Java Transform class can be saved using the SAV command and restored on a target system using the RST command. The following example shows how to deploy our RemoveEmpty XSL module and Java Transform class from our source system to a target system:

1. On the source system, create a save file to store the saved module:

   ```
   CRTSAVF SAVF(XMLREDBOOK/RMVEMTIFSS)
   ```

2. On the source system, save the RemoveEmtpy XSL module and Java Transform class to the RMVEMTIFSS save file:

   ```
   SAV DEV('/QSYS.LIB/XMLREDBOOK.LIB/RMVEMTIFSS.FILE')
   OBJ(('/XMLREDBOOK/CORPXML/RemoveEmpty.xsl'))
         ('/XMLREDBOOK/CLASSES/Transform.class')) DTACPR(*YES)
   ```

3. On the target system, create a save file to receive the saved module to be deployed:

   ```
   CRTSAVF SAVF(XMLREDBOOK/RMVEMTIFSS)
   ```

4. On the target system, restore the RemoveEmpty XSL module to the target schema:

   ```
   RST DEV('/QSYS.LIB/XMLREDBOOK.LIB/RMVEMTIFSS.FILE')
   OBJ(('/XMLREDBOOK/CORPXML/RemoveEmpty.xsl')) DTACPR(*YES)
   ```

5. On the target system, restore the Java Transform class to the target schema:

   ```
   RST DEV('/QSYS.LIB/XMLREDBOOK.LIB/RMVEMTIFSS.FILE')
   OBJ(('/XMLREDBOOK/CLASSES/Transform.class')) DTACPR(*YES)
   ```

5.2.5 Decomposition execution

Now that we have coded and deployed our solution, we must execute the modules and decompose our XML documents. The command must be run for each XML document that requires decomposition into our CorpSales table.

The CORSLSSHRD CMD is used to initiate the decomposition process for each CorpSales XML document that we want to decompose. From a command line, type the following command and press F4:

```
CORSLSSHRD
```

This command uses the following parameters:

- INPUT

  This parameter is composed of two parts, a directory and a document. We enter the integrated file system directory that contains the XML document that we want to decompose. The document is the actual XML document that we are decomposing.

- CLASSPATH

  This parameter is the directory that contains the Transform Class.

- DEBUG

  This is a debug flag to control the display of the Java execution script window.

The following example shows how the command is entered on a command line after the user has entered the necessary parameters to decompose corporate sales for the USA for day six without debugging:

```
CORSLSSHRD INPUT('/XMLRedbook/CorpXML' 'CorpSalesUSA2006-04-06.xml')
           CLASSPATH('/XMLRedbook/classes')
           DEBUG(*NO)
```

After the decompose has completed for the document specified, the command line returns. Validate that the XML data was decomposed correctly and inserted into the CorpSales table.

5.2.6 Considerations for RPG decomposition

As of the writing of this redbook, keep in mind the following considerations when using RPG to decompose an XML document. For more information about the limitations of the XML support added to RPG, refer to Chapter 23 in the *WebSphere Development Studio ILE RPG Language Reference*, SC08-2508.

- The decomposition module provided in this chapter does not provide examples of error handling processing. When coding your decomposition solution, you must take this into consideration and trap any errors that might occur during decomposition and report them back to the user or log for later retrieval.
- Use of the Java virtual machine (JVM) on a System i machine requires simple tuning to optimize performance. Since we use the JVM to run our Transform class, it is important to note the memory pool that your JVM executes in to set it appropriately to improve performance. This tuning comes into play more when you are processing large or complex XML documents. Refer to the *Tuning Garbage Collection for Java and WebSphere on iSeries* paper at the following Web address for additional information:

  http://www.ibm.com/servers/eserver/iseries/perfmgmt/pdf/tuninggc.pdf
- The decomposition module provided in this chapter does not provide examples of using the %HANDLER built-in function added to RPG to handle XML documents of large or unknown size since our XML document was known to be small in size. If our XML document was larger or unknown in size, we might use the %HANDLER built-in function to initiate the XML-INTO processing. The %HANDLER identifies a procedure that handles overflow of XML data after the defined data structures are populated and more XML data exists to be decomposed.
- As shown in our code example, the XML-INTO operation cannot handle empty elements or attributes that contain zero-length numeric, date, time, or timestamp fields during decomposition. Since our CorpSales XML document contains empty elements and attributes, one of which contains the Amount field that is numeric, we cannot decompose the XML document as is. To correct this, we chose to use a stylesheet (XSL) to transform the inbound XML document to remove all white space, empty elements, and attributes prior to decomposition to prevent errors. Other approaches are to have the document generated with default data or to code our RPG data structure as all character fields and handle any numeric columns programmatically.
- The RPG compiler limits character variables to 65535 characters in length. Data structures that are considered as character fields are subject to this length limitation as well. When replicating the XML tree using data structures, especially data structures that are dimensional, you must be aware that the total size of the data structure tree does not exceed this limit. Since our XML documents were relatively simple and small in size, we did not reach this limitation. A workaround is to break apart the data structures without nesting them in side each other and use multiple XML-INTO operations to decompose, specifying the specific path of the XML document that matches the data structure names.

6

Using SAX and Java to decompose XML

In Chapter 5, “Using RPG for XML processing” on page 59, we discuss how to use the built-in XML support in RPG to implement step 6 of our scenario. With solid, traditional OS/400® programming, we created an application that decomposes CorpSales documents into the DB2 database. In this chapter, we implement the same step 6 (see Figure 6-1) of our scenario by taking advantage of the SAX parser to parse the XML and JDBC to persist the data in the DB2 database.

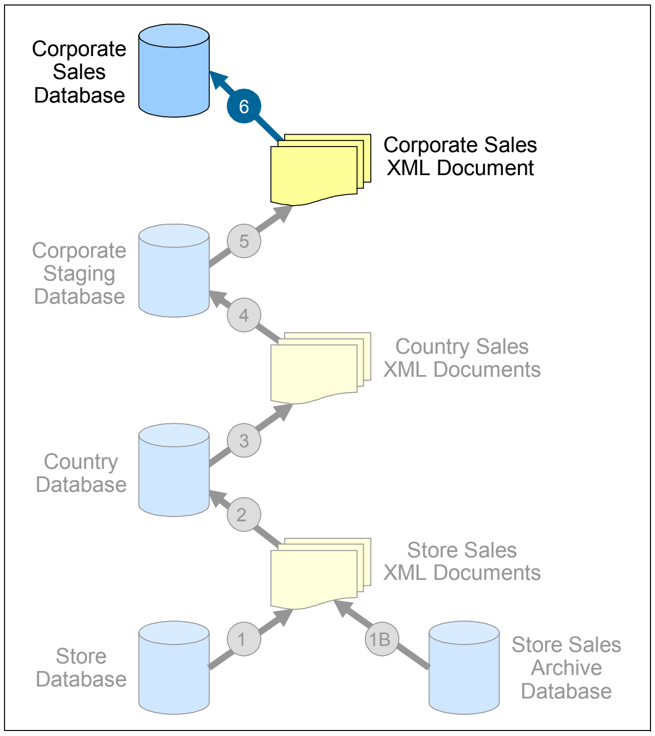

Corporate Sales Database
6
Corporate Sales XML Document
Corporate Staging Database
5
4
Country Sales XML Documents
Country Database
3
2
Store Sales XML Documents
Store Database
1
1B
Store Sales Archive Database

Figure 6-1 Scenario step 6: Alternate method

We use Java as the programming language. This approach should appeal to those in the System i development community who are not familiar with traditional green-screen-based programming.

The solution implementation process is divided roughly into two phases:

- *Design and implementation*: In this phase, we analyze the structure of the inbound XML document, define the XML to relational database (RDB) mapping, and design and code the necessary Java classes. We use WebSphere Development Studio Client to manage the development process. You can, however, use any Java development environment. In fact, you can develop the entire solution on System i machine using a text editor and Qshell.
- *Deployment*: In this phase, we deploy the Java classes to the target System i machine and run the application to shred inbound documents.

6.1 Setting up the development environment

The solution described in this chapter does not depend on a specific development environment. However, we decided to use WebSphere Development Studio Client to manage the project. In this section, we provide a short overview of the setup procedure. We intend to call SAX and JDBC APIs from Java. In WebSphere Development Studio Client, we create a new Java project called *DB2XMLRedbook_ScenarioStep6*.

The development process starts by importing a sample inbound document to be shredded (decomposed) along with its document type definition (DTD). The CorpSales documents were created in step 5 of our scenario:

1. In the Java perspective, create a new folder, called `xml`, in the newly created project.
2. Import the files that reside on the target IBM System i5™ machine in the /XMLRedbook/CorpXML directory. (Refer to “Locating the Web material” on page 207). The integrated file system root is mapped as a network drive on the development workstation.
   a. In the Resource perspective Navigator panel, of WebSphere Development Studio Client, right-click the **CorpXML** folder and select **Import**.
   b. In the Import window, select **File System** and click **Next**.
   c. In the File System window, next to the From Directory text box, click **Browse**. Navigate to the /XMLRedbook/CorpXML integrated file system directory. Select the following files:
      - CorpSalesUSA2006-04-06.xml
      - CorpSales.dtd
3. Create a new Java package to contain the Java classes. We call this package iDB2XML.
4. Since the development is performed on a workstation and we want to access remote DB2 for i5/OS database server, add the IBM Toolbox for Java JDBC driver to the current project’s Java Build Path. We recommend that you download the latest version of the Toolbox JDBC driver from the following Web address:

   `http://www.software.ibm.com/webapp/download/search.jsp?go=y&rs=expastbjm3`

   a. Select the latest open source version of the driver called JTOpen.
   b. Save the JTOpen zip file to a local workstation directory, for example C:\code\JTOpen-Latest.
   c. Extract the jt400.jar file into the current directory.

5. In WebSphere Development Studio Client, right-click the **DB2XMLRedbook_ScenarioStep6** project and select **Properties**.
6. In the left pane of the Properties window (Figure 6-2), click **Java Build Path**. Then in the right pane, click the **Libraries** tab. Click the **Add Variable** button.

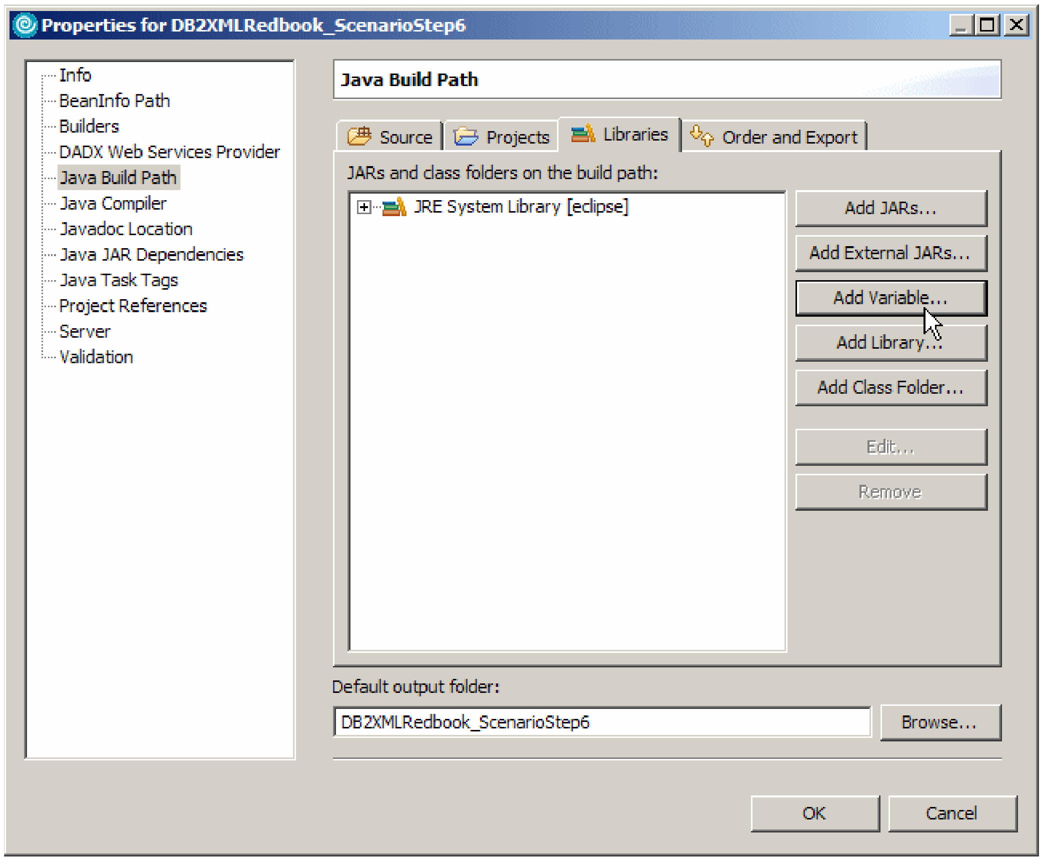
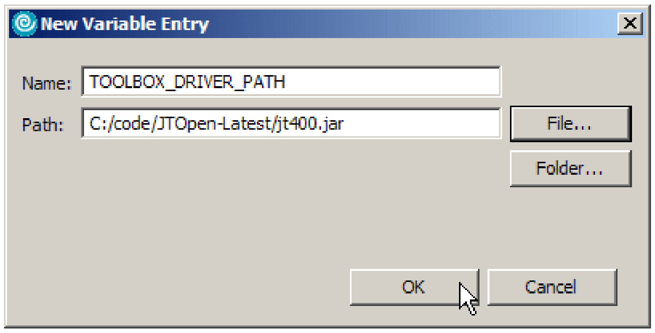


Figure 6-2 Adding variable to Java Build Path

7. In the New Variable Classpath Entry window, click the **Configure Variables** button.
8. In the Preferences window that opens, click **New**.

9. In the New Variable Entry window (Figure 6-3):
   a. In the Name text box, type `TOOLBOX_DRIVER_PATH`.
   b. Click **File**.
   c. Navigate to the directory in which you previously extracted the jt400.jar file. Select this file. Click **OK**.
   d. Click **OK** to add the variable definition to the current project.

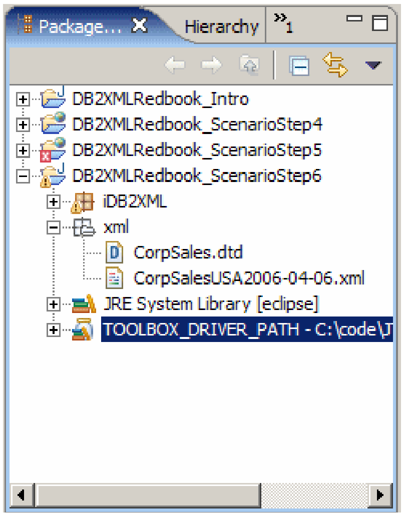

New Variable Entry

Name: TOOLBOX_DRIVER_PATH

Path: C:/code/JTOpen-Latest/jt400.jar

Figure 6-3 New Variable Entry definition window

Figure 6-4 shows the structure and the initial content of the project.



DB2XMLRedbook_Intro
DB2XMLRedbook_ScenarioStep4
DB2XMLRedbook_ScenarioStep5
DB2XMLRedbook_ScenarioStep6
iDB2XML
xml
CorpSales.dtd
CorpSalesUSA2006-04-06.xml
JRE System Library [eclipse]
TOOLBOX_DRIVER_PATH - C:\code\J

Figure 6-4 Initial Java project structure

6.2 Defining the XML to RDB mapping

The purpose of XML to RDB mapping is to associate a given element or attribute value with a specific column of a table in the target database. We start the process by analyzing the hierarchy of a sample inbound document to determine the relationship among elements and attributes. Specifically, we focus on such concepts as element containment, repeating elements, and self-named elements. Refer to 1.3, “XML to relational database mapping” on page 15, for more details.

The XML hierarchy is described by the DTD associated with the inbound documents. Example 6-1 shows the CorpSales.dtd.

Example 6-1 CorpSales.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT CorpSales (CountryInfo,SalesByBrand) >
<!ATTLIST CorpSales
  Date CDATA #IMPLIED>
<!ELEMENT CountryInfo (Name) > 3
<!ELEMENT Name (#PCDATA)>
<!ELEMENT SalesByBrand (Brand+) > 1
<!ELEMENT Brand (Name,Sales?,Returns?) > 2
<!ELEMENT Sales (Currency,Amount) >
<!ELEMENT Returns (Currency,Amount) >
<!ELEMENT Currency (#PCDATA)>
<!ELEMENT Amount (#PCDATA)>
```

The DTD shown in Example 6-1 describes the following information about the XML hierarchy:

- **Repeating elements**: Some elements may repeat within a parent element. In line 1 in Example 6-1, a SalesByBrand element can contain one or many occurrences of the Brand element. Notice the plus (+) sign after *Brand*.
- **Optional elements**: Some elements are optional. In line 2, a Returns element may occur zero or one time within a Brand element. Notice the question mark (?) after *Returns*. Similarly, the Sales element is optional within Brand.
- **Wrapper elements**: These elements have no child attributes or text nodes. However, they contain children elements that have attributes or text nodes. In this case CountryInfo in line 3 is a wrapper element. It is used to logically separate country from brand sales data.

In this case, the target database already exists, and the table layout is predetermined by the corporate headquarters. The target schema contains one table called *CorpSales*. Example 6-2 shows the structure of this table.

Example 6-2 CorpSales table definition

```
CREATE TABLE CORPSALES.CORPSALES (
   COUNTRYNAME VARCHAR(50) ALLOCATE(30) NOT NULL ,
   BRANDNAME VARCHAR(80) ALLOCATE(50) NOT NULL ,
   SALESDATE DATE NOT NULL ,
   TYPE VARCHAR(30) ALLOCATE(10) NOT NULL ,
   CURRENCY VARCHAR(30) ALLOCATE(10),
   AMOUNT DECIMAL(9, 2))
   ;
```

By comparing the DTD to the CorpSales structure, most of the mapping is straightforward. We start the mapping from the root element in the XML hierarchy namely CorpSales. CorpSales has an attribute called *Date*. This attribute occurs in a given document once and is mapped to the SALESDATE column.

SalesCorp has two children: CountryInfo and SalesByBrand. Both of these elements are wrappers so they contain no data to be written to the database. CountryInfo contains the element *Name*. This element occurs only once and is mapped to the COUNTRYNAME column. The SalesByRegion wrapper, however, contains a multi-occurrence element called *Brand*. The multi-occurrence elements must be shredded into separate rows in the target table.

A particular occurrence of a Brand is identified by its child element *Name*. This Name element maps to BRANDNAME column. Brand has two other child elements, Sales and Returns. Both of these elements are optional. Each element contains two children, Currency and Amount.

We need four additional columns in the CORPSALES table to accommodate the data contained in Sales and Returns. The CORPSALES table, however, has two appropriate columns, CURRENCY and AMOUNT. Consequently, Brand elements that have both optional elements require two separate rows to accommodate the data. We also need an indicator if the data in a particular row refers to sales or returns. We use the TYPE column for that purpose. Our shredding logic must generate the value for this column depending on whether the current row contains sales or returns data.

Table 6-1 summarizes the final mapping that we performed.

Table 6-1 CorpSales hierarchy mapped to the CORPSALES table

Location path	Table name	Column name	Comment
/CorpSales/@Date	CORPSALES	SALESDATE	
/CorpSales/CountryInfo/Name	CORPSALES	COUNTRYNAME	
/CorpSales//SalesByBrand/ Brand/Name	CORPSALES	BRANDNAME	Each occurrence of a Brand element results in one or two rows in the table depending on whether one or both optional elements exist.
-	CORPSALES	TYPE	This column is generated by the shredding program logic.
/CorpSales/SalesByBrand/ Brand/Sales/Currency	CORPSALES	CURRENCY	
/CorpSales/SalesByBrand/ Brand/Sales/Amount	CORPSALES	AMOUNT	
/CorpSales/SalesByBrand/ Brand/Returns/Currency	CORPSALES	CURRENCY	This column is mapped to the same column as /CorpSales/ SalesByBrand/Brand/Sales/ Currency.
/CorpSales/SalesByBrand/ Brand/Returns/Amount	CORPSALES	AMOUNT	This column is mapped to the same column as /CorpSales/SalesByBrand/Brand/Sales/ Amount.

To recap the logic behind the mapping, one inbound XML document will generate multiple rows in the CORPSALES table. There will be one or two rows for a particular occurrence of Brand element. A particular row in a table can be uniquely identified by the columns COUNTRYNAME, SALESDATE, BRANDNAME, and TYPE. From the relational database theory point of view, these four columns can be used as the primary key for that table.

6.3 Implementing the mapping in Java

The mapping process outlined in the previous section determined the association between data contained in the inbound XML document and columns in the DB2 table. Now we must implement the mapping logic in a Java application. Here is a list of the programming artifacts in the solution:

- **ShredXML.java**: This class uses the SAX parser APIs to parse the inbound XML documents. It keeps track of the current location path for the context node (currently parsed node). It also interfaces with the RdbAdapter class to write the shredded (decomposed) data into the database.

- **RdbAdapter.java**: This interface defines the methods that must be implemented by the database layer adapter class. These methods are invoked by the ShredXML class to pass a location path and the data contained in an element or attribute pointed to by that location path.
- **CorpSalesAdapter.java**: This class implements the RdbAdapter interface. It is used to perform the mapping by matching a given location path, passed by the ShredXml class, with a target column. It then uses the JDBC API to insert the data into the CORPSALES table.

We start the detailed analysis with the ShredXML class. We decided to use SAX because it is fast and efficient. It allows us to process vary large documents without requiring large memory footprint. Example 6-3 shows the source code.

Example 6-3 ShredXML.java

```
package iDB2XML;
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import java.sql.*;

public class ShredXml extends DefaultHandler {
   private StringBuffer textBuffer;
   private StringBuffer currentLocationPath;
   static RdbAdapter rdbAdapter;

   public static void main(String[] argv) {
if (argv.length != 2) {
         System.err.
         println("Usage: java -jar iDB2XML.jar XMLFileName RdbAdapterClassName");
         System.exit(1);}
      // Use an instance of ourselves as the SAX event handler
      DefaultHandler handler = new ShredXml();
      // Use the non-validating parser
      SAXParserFactory factory = SAXParserFactory.newInstance();1
      try {
         // Use the RdbAdapter to store data in ralational database
         Class c = Class.forName(argv[1]);  //Current implementation of RdbAdapter
         rdbAdapter = (RdbAdapter) c.newInstance();
         // Parse the inbound XML document
         SAXParser saxParser = factory.newSAXParser(); 2
         saxParser.parse(new File(argv[0].trim()), handler); 3
         System.err.println( "File : " + argv[0].trim() + " successfully 
shredded.");
      } catch (Throwable t) {
         t.printStackTrace();
      }
      System.exit(0);
   }
   // SAX DocumentHandler methods implementation
   public void startDocument() throws SAXException {
```

```
   currentLocationPath = new StringBuffer("");
}
public void endDocument() throws SAXException {
   try {
      //Flush batched but unwritten rows to the database
      rdbAdapter.flushRowCache();
   } catch (SQLException ex) {
      throw new SAXException("SQL error", ex);
   }
      currentLocationPath = null;
}
public void startElement(String namespaceURI, String sName, 4 // simple name
      String qName, // qualified name
      Attributes attrs) throws SAXException {
   String s = getText(); // cleanup leading white chars 5
   String eName = sName; // element name
   if ("".equals(eName)) {
      eName = qName; // not namespaceAware
   }
   //keep track of the current location path
   currentLocationPath.append("/" + eName); 6
   if (attrs != null) {
      for (int i = 0; i < attrs.getLength(); i++) {
         String aName = attrs.getLocalName(i); // Attr name
         if ("".equals(aName)) {
            aName = attrs.getQName(i);
         }
         currentLocationPath.append("/@" + aName); 7
         // attribute's location path and its value
         writeTextToDb(currentLocationPath.toString(),
                       attrs.getValue(i).trim()); 8
         // attribute processed - return to containing element
         currentLocationPath.setLength(currentLocationPath.lastIndexOf("/@"));9
      }
   }
}
public void endElement(String namespaceURI, String sName, // simple name 10
      String qName // qualified name
) throws SAXException {
   //accumulated text node for the current element
   String s = getText(); 11
   String eName = sName; // element name
   if ("".equals(eName)) {
      eName = qName; // not namespaceAware
   }
   //element's location path and its text node value
   writeTextToDb(currentLocationPath.toString(), s); 12
   //element processed - return to parent
   currentLocationPath.setLength(currentLocationPath.lastIndexOf("/")); 13
}
public void characters(char[] buf, int offset, int len) throws SAXException {
   String s = new String(buf, offset, len);
   if (textBuffer == null) {
      textBuffer = new StringBuffer(s);
   } else {
         textBuffer.append(s);
      }
   }

   // Helper Methods
   private String getText() throws SAXException {14
      if (textBuffer == null) {
         return null;
      }
      String s = "" + textBuffer;
      textBuffer = null;
      return s;
   }
   private void writeTextToDb(String locPath, String value)15
         throws SAXException {
      try {

         if (value == null) {
            return;
         }
         rdbAdapter.fillDbRows(locPath, value);
      } catch (SQLException ex) {
         throw new SAXException("SQL error", ex);
      }
   }
}
```

The ShredXML class extends the DefaultHandler class, which is contained in the org.xml.sax.helpers package. Inheriting from the DefaultHandler class helps us to implement the ContentHandler interface. This interface, in turn, defines a number of methods that the SAX parser invokes when various parsing events occur. For our purposes, we must implement the following event-handling methods:

- startDocument
- endDocument
- startElement
- endElement

The ShredXML class takes two arguments:

- **XMLFileName**: The name of the inbound XML file to be shredded
- **RdbAdapterClassName**: The name of the specific implementation of the RdbAdapter

In line 1 in Example 6-3, we create a SaxParserFactory instance that is used in line 2 to create a SAXParser. Then in line 3, we use the parser to parse the inbound XML document. We also pass to the parser the reference to the ShredXML class (self-reference) since this class handles the parsing events.

In line 4, the startElement method is invoked each time a new element opening tag is encountered by the parser. In line 5, we clean up the white space characters that were accumulated since the preceding element closing tag has been processed. Remember that adjacent text nodes are concatenated. Typically, at this point the textBuffer contains tabs, new line characters, and so on, so we can safely ignore this content.

In line 6, we update the current location path by appending the name of the current element. Keeping track of the location path is critical for our implementation. As indicated in Table 6-1 on page 94, we use the location path to map an element or attribute to a column in the DB2 table. In line 7, we set the current location path for an attribute. In line 8, the location path for the attribute along with its value are sent to the data persistency layer (RdbAdapter class). In line 9, since the current attribute has been processed, the location path is reset to the location path for the containing element.

Note that all attributes are handled in a loop. In line 10, the endElement method is invoked each time an element closing tag is encountered. In line 11, we retrieve the text node accumulated for the current element. In line 12, we pass the location path for the current element along with its text node value to the database persistency layer. Since the element and all attributes of this element have been processed, we reset the location path so it points to the parent (line 13).

The ShredXML class also contains two helper methods:

- **getText** (line 14): This method is used to return the characters currently accumulated by the parser in the textBuffer object.
- **writeTextToDb** (line 15): This method interfaces with the RdbAdapter class. The empty elements and attributes are handled by this method beforehand so no call to the RdbAdapter is necessary.

Note that the ShredXml class depends neither on the structure of the inbound XML document nor on the particular implementation of the RdbAdapter interface. ShredXml will properly parse any XML document and pass the extracted data to the database persistency layer.

The other class in our implementation is CorpSalesAdapter. As mentioned, it implements the RdbAdapter interface. The interface definition is simple. We require that RdbAdapter implements two methods shown in Example 6-4.

Example 6-4 RdbAdapter interface

```
package iDB2XML;
import java.sql.SQLException;

public interface RdbAdapter {
   public void fillDbRows(String locPath, String value) throws SQLException;
   public void flushRowCache() throws SQLException;
}
```

The CorpSalesAdapter class encapsulates the XML to RDB mapping. It checks if the location paths passed by the ShredXML class are mapped to a table column. If so, it performs any necessary data type conversions and inserts the value associated with a given location path into the database table row buffer. When the row buffer is filled, the row is batched for insert. Finally, when the cache of rows is filled, the batch of rows is inserted into the database. The batch insert programming technique is used to improve performance of the database insert operations.

The class uses the properties file to set a number of properties at run time. Example 6-5 shows the content of the properties file.

Example 6-5 RdbAdapter.properties

```
#Toolbox driver 1
dbDriver=com.ibm.as400.access.AS400JDBCDriver 2
dbUrl=jdbc:as400://rchas10;libraries=CORPSALES; 3
dbUser=jarek 4
dbPassword=******* 5
```

```
#Native Type 2 driver 6
#dbDriver=com.ibm.db2.jdbc.app.DB2Driver
#dbUrl=jdbc:db2://*LOCAL;use block insert=true; 7
#dbUser=jarek
#dbPassword=*******

#Common properties 8
tableName=CORPSALES.CORPSALES
batchInsertSize=100 9
```

The section that starts at line 1 in Example 6-5 contains the parameters required to use the Toolbox JDBC driver. This driver is used when connecting to DB2 for i5/OS from the development workstation. In line 2, we set the name of the driver class. In line 3 the database URL is specified. In lines 4 and 5, we provide the credentials for the user that connects to the database.

The section starting with 6 is commented out. It contains the connectivity information pertaining to the native (Type 2) DB2 for i5/OS JDBC driver. We use this driver when the solution is deployed to the target System i machine. The Type 2 driver is recommended for JDBC applications that run on the same system (in the same partition) where the database resides. The Type 2 driver does not require communication with the database over the network so it typically performs better than a Type 4 (Toolbox) driver.

The Type 2 driver requires that the block insert is explicitly enabled (line 7). In line 8, the section common for both drivers starts. The size of the row cache size is determined in line 9. For large XML documents that generate lots of database rows, we recommend that you set this value to a much larger number such as 10,000. This can dramatically improve the performance of the inserts.

We now discuss the implementation details of the CorpSalesAdapeter class. Example 6-6 shows the source code.

Example 6-6 CorpSalesAdapter.java

```
package iDB2XML;
import java.sql.*;
import java.util.Properties;
import java.text.*;
import java.math.*;
import java.io.*;
public class CorpSalesAdapter implements RdbAdapter{
   private PreparedStatement ps;
   private Connection con;
   private int batchInsertSize = 1;
   private int batchedRows = 0;
   public CorpSalesAdapter() throws Exception {
      Properties properties = new Properties();
      //Retrieve the runtime environment
      properties.load(new BufferedInputStream(new
            FileInputStream("RdbAdapter.properties"))); 1
      String dbDriver = properties.getProperty("dbDriver");
      String dbUrl = properties.getProperty("dbUrl").trim();
      String dbUser = properties.getProperty("dbUser").trim();
      String dbPassword = properties.getProperty("dbPassword").trim();
      String tableName = properties.getProperty("tableName").trim();
```

```
         batchInsertSize =
    Integer.parseInt(properties.getProperty("batchInsertSize").trim());

         Class.forName(dbDriver);
         con = DriverManager.getConnection(dbUrl, dbUser, dbPassword);
         ps = con
            .prepareStatement("INSERT INTO "
            +tableName+"(countryname, salesdate, brandname, type, currency, amount)"
                  + " VALUES( ?, ?, ?, ?, ?, cast(? as decimal(9,2)))"); 2
      }

      public void fillDbRows(String locPath, String value) throws SQLException { 3
         try {
            if (locPath.equals("/CorpSales/@Date")) { 4
               try {
                  java.util.Date tmpDt = new SimpleDateFormat("yyyy-MM-dd")
                        .parse(value);
                  java.sql.Date dt = new java.sql.Date(tmpDt.getTime());
                  ps.setDate(2, dt);
               } catch (ParseException pe) {
                  System.err.println(pe);
               }

            } else if (locPath.equals("/CorpSales/CountryInfo/Name")) {
               ps.setString(1, value);
            } else if (locPath.equals("/CorpSales/SalesByBrand/Brand/Name")) {
               ps.setString(3, value);
            } else if (locPath
                  .equals("/CorpSales/SalesByBrand/Brand/Sales/Currency")) {
               ps.setString(4,"SALE");
               ps.setString(5, value);
            } else if (locPath
                  .equals("/CorpSales/SalesByBrand/Brand/Sales/Amount")) {
               BigDecimal bd = new BigDecimal(value);
               ps.setBigDecimal(6, bd);
               batchRow(); 5
            } else if (locPath
                  .equals("/CorpSales/SalesByBrand/Brand/Returns/Currency")) {
               ps.setString(4,"RETURN");
               ps.setString(5, value);
            } else if (locPath
                  .equals("/CorpSales/SalesByBrand/Brand/Returns/Amount")) {
               BigDecimal bd = new BigDecimal(value);
               ps.setString(6, value);
               batchRow(); 6
            }
         } catch (SQLException ex) {
            System.err.println(ex);
            throw ex;
         }
      }

      public void batchRow() throws SQLException { 7
         ps.addBatch();
         batchedRows++;
         if (batchedRows >= batchInsertSize){
            ps.executeBatch();
            batchedRows = 0;
         }
      }
      public void flushRowCache() throws SQLException { 8
         ps.executeBatch();
      }
   }
```

Starting with line 1 in Example 6-6, the properties file is loaded so that the runtime parameters can be set. In this scenario, there is one target table, so we need one insert statement to populate the rows. At line 2, we use a PreparedStatement object to prepare the necessary INSERT statement.

The implementation of the fillDbRows method starts at 3. The method takes two parameters, the location path and the value. These two parameters are passed by the ShredXml class. A series of if-checks is used to test whether the location path is mapped to a column in the CORPSALES table. If it is, the String passed as value parameter is converted to the target column data type and the appropriate parameter of the PreparedStatement is set. An example is shown in line 4. When the entire row is filled with data, the row is batched for insert.

In line 5, the batchRow method is called when the current Brand element being processed by the parser has the Sales element. Similarly, the batchRow method is called at line 6 to insert the row for the Brand element that contains Returns. The implementation of the batchRow method starts with line 7. Notice how the executeBatch method is invoked on the PreparedStatement object to write a batch of rows into the database. In line 8, the flushRowCache method allows us to insert any unwritten rows when the parser reaches the end of XML document (see the endDocument method in Example 6-3 on page 95).

6.4 Deploying the solution to the System i machine

The finished solution consists of the following objects:

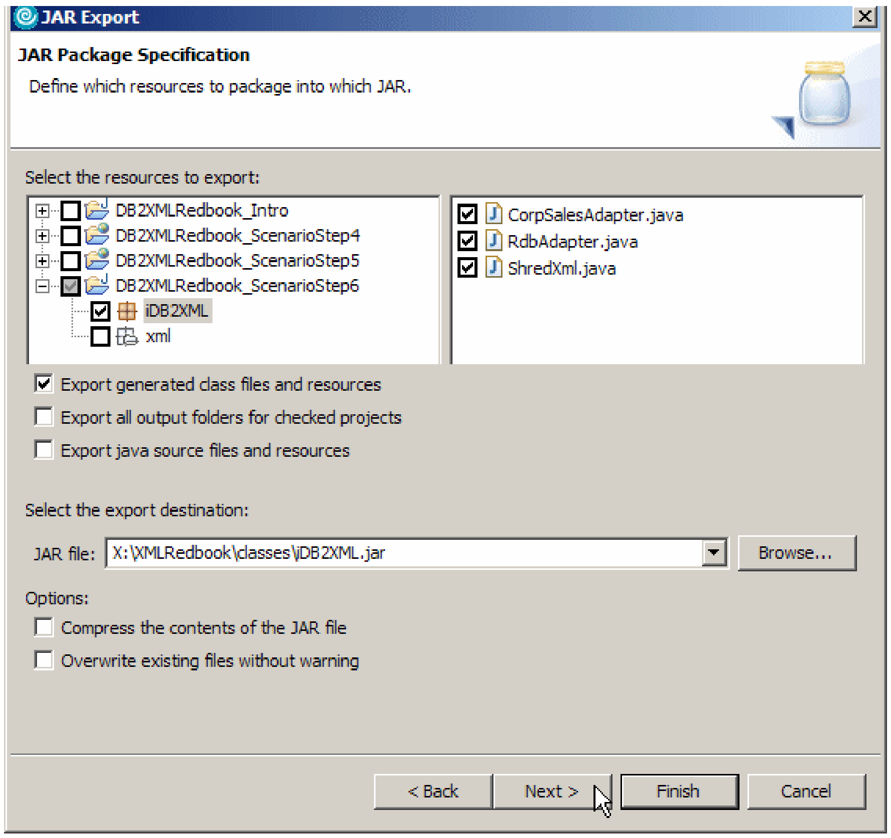
- ShredXML.class
- RdbAdapter.class
- CorpSalesAdapter.class
- RdbAdapter.properties

We decided to package the Java classes as a runnable Java archive (JAR) file. The JAR file is called iDB2XML and is deployed to the /XMLRedbook/classes directory on the target System i machine.

To deploy the solution using WebSphere Development Studio Client:

1. Assume that the local drive X: on the development workstation is mapped to the integrated file system Root directory on the System i machine. In WebSphere Development Studio Client, switch to the Java perspective.
2. In the Navigator window, under the DB2XMLRedbook_ScenarioStep6 project, right-click the **iDB2XML** package and select **Export**.
3. In the Export window, select **Jar file** and then click **Next**.

4. In the JAR Package Specification window, select the three Java objects and set the export destination as shown in Figure 6-5. Click **Next**.


JAR Export
JAR Package Specification
Define which resources to package into which JAR.
Select the resources to export:
DB2XMLRedbook_Intro
DB2XMLRedbook_ScenarioStep4
DB2XMLRedbook_ScenarioStep5
DB2XMLRedbook_ScenarioStep6
iDB2XML
xml
CorpSalesAdapter.java
RdbAdapter.java
ShredXml.java
< Back
Next >
Finish
Cancel
Browse...

[✓] Export generated class files and resources
[ ] Export all output folders for checked projects
[ ] Export java source files and resources

Select the export destination:

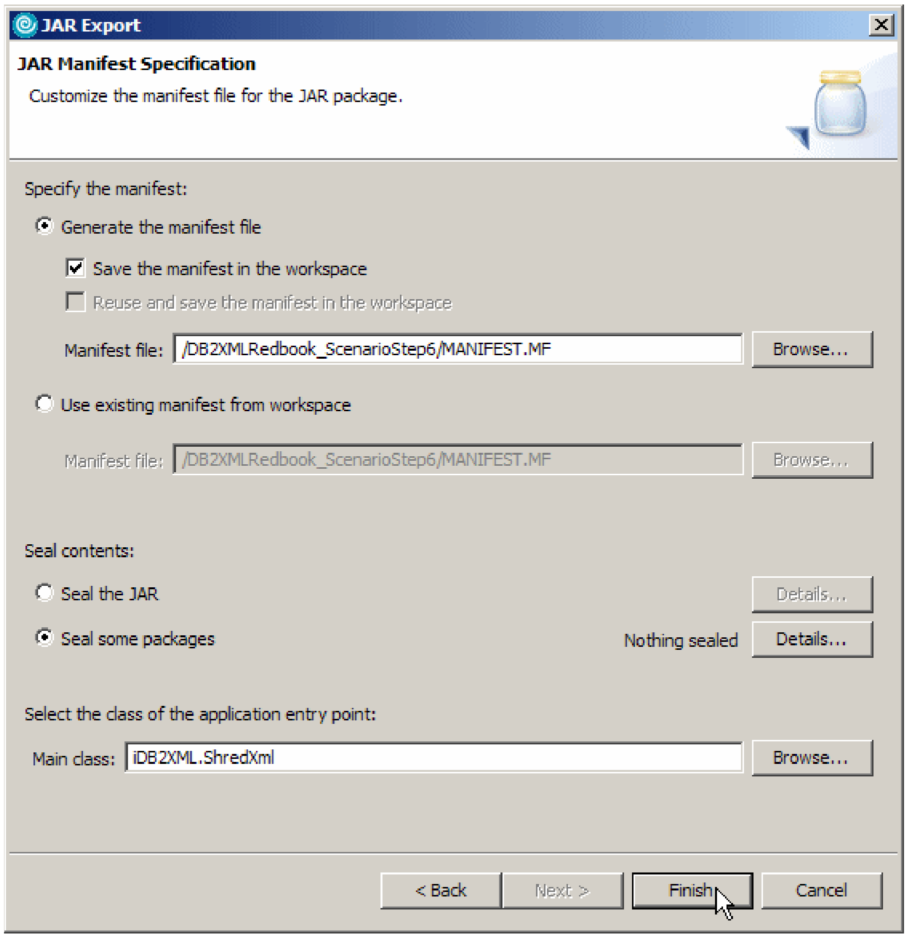
JAR file: X:\XMLRedbook\classes\iDB2XML.jar

Options:
[ ] Compress the contents of the JAR file
[ ] Overwrite existing files without warning

Figure 6-5 Exporting the solution to a JAR file

5. In the JAR Packaging Options window, click **Next**.

6. In the JAR Manifest Specification window, specify the main class file as shown in Figure 6-6. Click **Finish**.


JAR Export
JAR Manifest Specification
Customize the manifest file for the JAR package.
Specify the manifest:
Generate the manifest file
Save the manifest in the workspace
Reuse and save the manifest in the workspace
Manifest file:
Browse...
Use existing manifest from workspace
Seal contents:
Seal the JAR
Seal some packages
Nothing sealed
Details...
Select the class of the application entry point:
Main class:
< Back
Next >
Finish
Cancel

Manifest file: /DB2XMLRedbook_ScenarioStep6/MANIFEST.MF

Main class: iDB2XML.ShredXml

Figure 6-6 Setting the main class in the JAR Manifest Specification window

7. Back in the main WebSphere Development Studio Client window, edit the RdbAdapter.properties file. Comment out all entries in the Toolbox driver section and uncomment the entries in the Native (Type 2) driver section.
8. Use the Export generated class files and resources as shown in Figure 6-4 on page 92 to export the RdbAdapter.properties to the /XMLRedbook/classes directory in the integrated file system on the target System i machine.

6.5 Running the deployed application

After the application is deployed on the System i machine, we can run the JAR file to shred any number of CorpSales documents. We decided to use the J2RE 1.5.0 IBM J9 Java Runtime Environment (JRE™). This is a Power PC® optimized 32-bit JRE that was first introduced in V5R4. To take advantage of this JRE, you must have Option 8 of the 5722-JV1 license product installed on your system.

Note: We also successfully tested the application on Classic 64-bit Java virtual machines (JVMs). Both, JDK™ 1.4.2 and JDK 1.5.0 worked fine.

We created a simple CL program to call the ShredXML program from the i5/OS prompt. Example 6-7 shows the source code.

Example 6-7 STRJVASHRD CL program

```
PGM        PARM(&XMLFILE) 1
   DCL        VAR(&XMLFILE)   TYPE(*CHAR) LEN(256)
   MONMSG     MSGID(CPF0000)
   ADDENVVAR  ENVVAR(JAVA_HOME) +
   VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk50+
   /32bit')                                   2
   CD         DIR('/XMLRedbook/classes')
   RUNJVA     CLASS(iDB2XML.jar) PARM(&XMLFILE +
   iDB2XML.CorpSalesAdapter)                  3
   RMVENVVAR  ENVVAR(JAVA_HOME)
ENDPGM
```

The inbound XML document name is passed as a parameter in line 1 in Example 6-7. In line 2, the environment variable is added for the current job so that the J9 JDK is used. In line 3, the RUNJVA command is used to execute the JAR file.

To call the CL program to shred one of the CorpSales documents stored in the /XMLRedbook/CorpXML directory, enter the following command:

```
CALL PGM(STRJVASHRD) PARM('/XMLRedbook/CorpXML/CorpSalesUSA2006-04-07.xml')
```

The same syntax is used to shred the five other CorpSales documents that are contained in the CorpXML directory.

7

Advantages and disadvantages of the programmatic approach

In Part 2, “Programmatic approach” on page 31, we discuss the programmatic approach to integrate XML into DB2 for i5/OS. We illustrate the use of traditional programming methods in languages, such as RPG, CL, and SQL, to do the integration. The standard parsers Simple API for XML (SAX) and Document Object Model (DOM) are available to assist.

In this chapter, we discuss the advantages and disadvantages of the programmatic approach.

7.1 Advantages

The programmatic approach to integrating XML and DB2 has several advantages, including these:

- It involves use of technology with which developers are most familiar with and can leverage their skill set. For example, many clients have personnel who have skills in such languages as RPG, COBOL, and CGI.
- There is more flexibility in the development since the developer can choose the preferred language. The developer has the choice of language in the process of the mapping.
- There is more flexibility in implementing the mapping, such as:
  - Mapping different sets of elements to the same table structure
  - Generating virtual values from elements to manipulate the data
- The existing languages have been enhanced to work better with XML. Such is the case of ILE RPG in V5R4.
- COBOL and CGI also support XML.
- It allows the leverage of emerging technologies such as SAX, DOM, Java, and Java Database Connectivity (JDBC).
- It allows implementation of best programming practices such as blocking rows for inserts.

7.2 Disadvantages

The programmatic approach to integrating XML and DB2 also has several disadvantages such as these:

- There is lack of tooling to assist in creation and maintenance.
- It is customized to each particular situation, which results in the following disadvantages:
  - Since it tends to use hard coded mapping, it requires high maintenance.
  - It may require you to update multiple programs and structures.
  - There is little code reuse.
  - There is a lack of standards to understand the coding methodologies.
- It requires a detailed knowledge of APIs for low-level programming for XML parsing and database access.

Part 3

Middleware approach

The second approach to integrating XML into DB2 for i5/OS is the middleware approach. In this methodology, we use the middleware products that can help us to store and retrieve XML data into DB2 for i5/OS. One of the products that we use is the DB2 UDB XML Extender for iSeries. The advantage of the middleware approach is that the development can be reused and it is flexible.

In this part, we explain the middleware approach by using examples from our fictional scenario that we introduced in Chapter 2, “Scenario overview” on page 21.

This part includes the following chapters:

- Chapter 8, “Overview of DB2 XML Extender” on page 109
- Chapter 9, “Performance-related information regarding XML Extender” on page 147
- Chapter 10, “Shredding methodology” on page 155
- Chapter 11, “Composing methodology” on page 181
- Chapter 12, “Advantages and disadvantages of the middleware approach” on page 197

8

Overview of DB2 XML Extender

In this chapter, we discuss the DB2 UDB XML Extender for iSeries product and how you can use it in your business applications. We describe the product's capabilities, explain how to get started using XML Extender, and provide examples on how to use the product. We also showcase the SQL mapping document access definition (DAD) solution in our scenario (step 3) and use of the XML Column method to archive StoreSales XML documents (step 1B) from our scenario that we introduce in Chapter 2, “Scenario overview” on page 21.

8.1 Product overview

DB2 UDB XML Extender for iSeries is middleware that helps you store and retrieve XML Data into DB2 for i5/OS. XML Extender is functional and flexible whether you have relational data that needs to be transformed into XML or XML documents that need to be stored in DB2 Universal Database™ tables. The product contains a rich set of user-defined types (UDTs), user-defined functions (UDFs), and stored procedures to manage XML data.

Your application can use the UDTs, UDFs, and stored procedures provided by DB2 UDB XML Extender for iSeries to process XML documents. XML Extender uses document type definitions (DTDs) and XML Schema Definitions (XSDs) to validate XML documents. It also uses DAD files to define how the data in your XML documents should be mapped to the tables in your i5/OS database. See Figure 8-1.

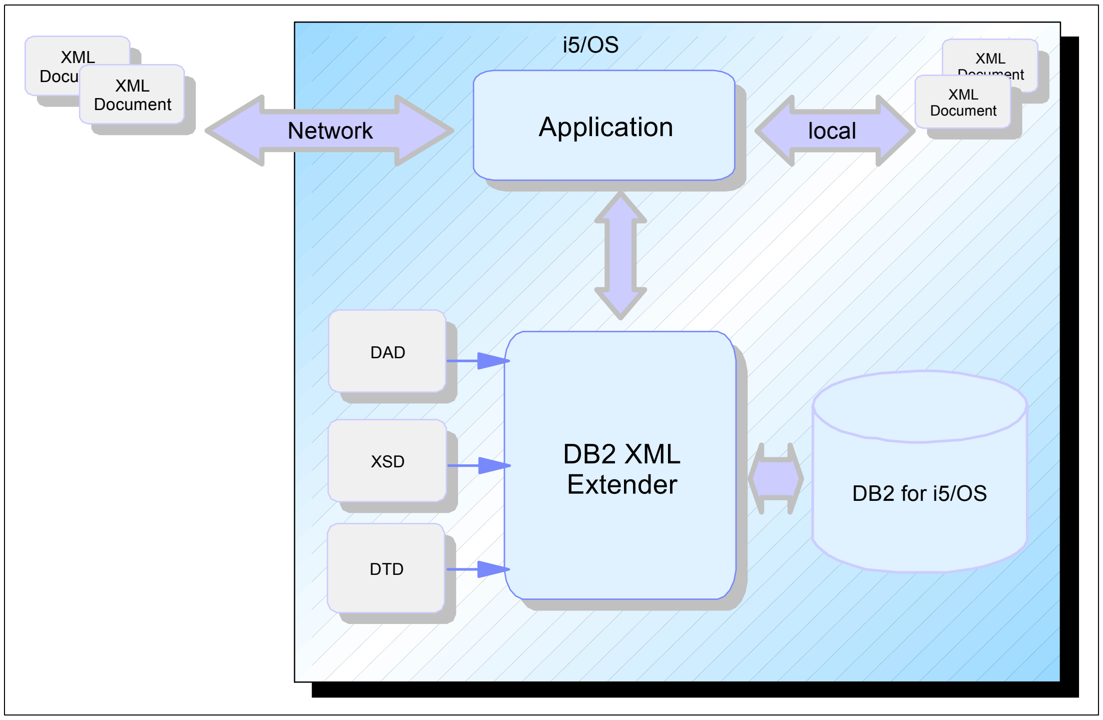


Figure 8-1 DB2 UDB XML Extender architecture for managing XML data

8.2 Setup and common administration tasks

DB2 UDB XML Extender for iSeries is not part of the DB2 for i5/OS runtime. It is shipped as a separate licensed program. XML Extender is option 2 of the licensed program 5722DE1-DB2 Extenders. Options 1 and 3 make up a different and separate product, the DB2 Text Extender for iSeries. The program product, 5722DE1, is included in the Enterprise Edition package or it can be ordered from IBM as a separate chargeable feature.

Before you use XML Extender, you must install some prerequisite software as explained in the following section. Then the XML Extender program product is installed, and additional steps are necessary to create XML Extender objects and install the sample program files.

8.2.1 Prerequisites

XML Extender depends on several other OS/400 components and program products. Install the following licensed programs before you install XML Extender:

- 5722SS1 Option 39: International Components for Unicode
- 5722XT1 *Base: XML Toolkit for iSeries
- 5733XT1 Option 8: XML Parser 5.3.1 Development

You must also install the following products for the XML Extender tutorial lessons and sample programs:

- 5722SS1 Option 13: System Openness Includes
- 5722ST1 *BASE: DB2 Query Mgr and SQL DevKit
- 5722WDS *BASE: WebSphere Development ToolSet for iSeries
- 5722WDS Option 51: ILE C
- 5733XT1 Option 8: XML Parser 5.3.1 Development

You may also require the following programs:

- 5722JV1: IBM developer Kit for Java, if you plan to develop applications that use Java or are Web based
- 5722WDS Option 52: ILE C++, if you plan to develop applications using C++
- 5722SS1 Option 30: Qshell, if you want to use the Qshell interface

XML Extender accepts a wide range of coded character set identifier (CCSID) languages. However, XML Extender does not recognize job CCSID 65535 as valid.

To change the job CCSID for the system:

```
CHGSYSVAL SYSVAL(QCCSID) VALUE(37)
```

To change the job CCSID for your user profile:

```
CHGUSRPRF USRPRF(YOURUSERID) CCSID(37)
```

To change the job CCSID for the current job:

```
CHGJOB CCSID(37)
```

Here you replace *37* with the CCSID of your system.

8.2.2 Preparing your database for XML Extender

To use the XML Extender functions in your application program, you must install the program product, restore the sample programs and sample files, and enable your i5/OS database for XML Extender as explained in the following sections.

Installing the program product

Use the Restore Licensed Program (RSTLICPGM) command to install the *BASE and option 2 of 5722DE1. If the XML Extender installation is successful, the following objects are created:

- QDBXM *LIB

  Product files and executables

- QDB2XML *LIB

  DB2 schema with the enable_db and disable_db stored procedures

- DB2XML *USRPRF

  User profile that owns the XML Extender created objects
- /qibm/ProdData/DB2Extenders/XML/MRI2924/dxx.cat

  XML Extender message files
- /qibm/ProdData/DB2Extenders/XML/include/dxx.h

  XML Extender defined constants and data types
- /qibm/ProdData/DB2Extenders/XML/include/dxxrc.h

  XML Extender return codes

If this is a scratch installation, it fails if either the QDB2XML or DB2XML schema or the DB2XML user profile already exists in the system. Prior to installing XML Extender for the first time, ensure that there are no objects by these names.

After a successful installation, apply the latest DB2 of i5/OS PTF Group and XML Extender PTFs. To find the latest PTF numbers for XML Extender, refer to the following Web page:

`http://www.ibm.com/software/data/db2/extenders/xmlext/support.html`

At the time this book was published, the latest XML Extender PTFs were:

- V5R4M0 5722DE1 SI24837
- V5R3M0 5722DE1 SI24827
- V5R2M0 5722DE1 SI24817

Installing, compiling, and creating the sample programs (optional)

The XML Extender sample files and getting started files are shipped as two savefile objects in the product directory. The savefile objects are:

- QDBXM/QZXMSAMP1

  The QZXMSAMP1 save file contains the DXXSAMPLES library. The library contains sample C source code, C header files, and SQL statements for application development.
- QDBXM/QZXMSAMP2

  The QZXMSAMP2 save file contains the integrated file system directory tree that will contain the sample XML, DTD, and DAD files that are used by the XML Extender Tutorial Lessons. The save file also contains a self-extracting GetStart.exe file to be used with the iSeries Navigator interface.

To install, compile, and create the sample programs:

1. Install the sample programs by entering the following commands on an i5/OS command line:

   ```
   RSTLIB SAVLIB(DXXSAMPLES) DEV(*SAVF) SAVF(QDBXM/QZXMSAMP1)

   RST DEV(‘/qsys.lib/qdbxm.lib/qzxmsamp2.file’)
   OBJ((‘/QIBM/UserData/DB2Extenders/XML/Samples’))
   ```
2. You must have a default schema that matches your user ID if you are going to use the XML Extender sample programs through the iSeries Navigator interface or if you will run the XML Extender Tutorial Lessons:

   a. From the i5/OS command line, enter the following command to open an SQL session:

      ```
      STRSQL
      ```

b. From the SQL session, enter:

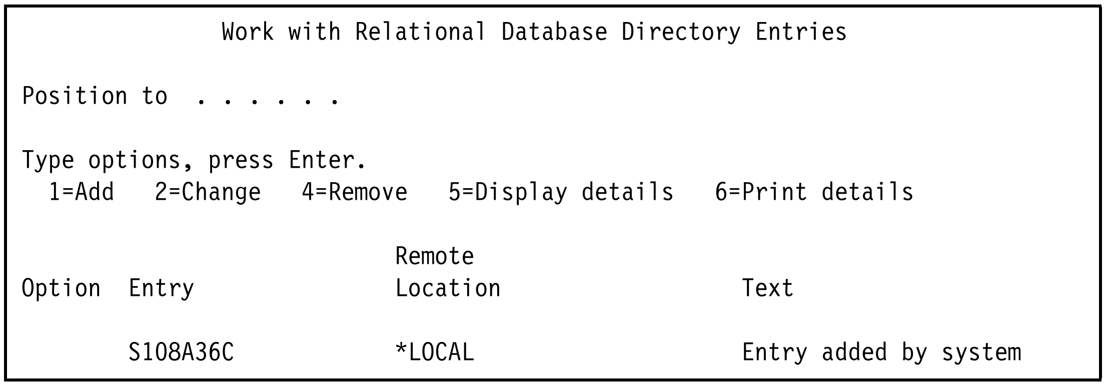
   ```
   CREATE SCHEMA userid
   ```

   Here *userid* is the user ID that you use with XML Extender.

c. You must know the RDB database name for the system where you use XML Extender. To find this name:

   i. Enter the Work with Relational Database Directory Entries (WRKRDBDIRE) command on an i5/OS command line.

   ii. From the list of registered databases, make note of the name with the remote address of *LOCAL, as shown in Figure 8-2.



```
                Work with Relational Database Directory Entries

Position to  . . . . . .

Type options, press Enter.
  1=Add   2=Change   4=Remove   5=Display details   6=Print details

                            Remote
Option  Entry               Location                  Text

        S108A36C            *LOCAL                    Entry added by system
```

Figure 8-2 The RDB database name, S108A35C, for the system

3. The XML Extender sample programs must be compiled and created. There are two ways to accomplish this:

   – If you are going to use a command line, enter the following command from the i5/OS command line:

   ```
   CALL DXXSAMPLES/SETUP
   ```

   – If you are going to use the iSeries Navigator interface:

   i. Download and unpack the /QIBM/UserData/DB2Extenders/XML/Samples/GetStart.exe file using FTP.

   ii. Start the iSeries Navigator.

   iii. Expand the tree for your i5/OS system, right-click your database, and select **Run SQL Scripts** as shown in Figure 8-3.

   iv. Open the setup.sql script file that was unpacked by GetStart.exe.

   v. Change all occurrences of *&SCHEMA* to the schema name that matches the user ID that you use with XML Extender.

   vi. Change all occurrences of *&DBNAME* to the RDB database name for your system.

   vii. Save the setup.sql file.

   viii. If you will run the XML Extender Tutorial Lessons, repeat steps iv through vii for each SQL file.

   ix. Reopen the setup.sql script file and click **Run all**.

   After the sample programs are compiled and created, the /dxxsamples symbolic link is created. The link points to the /QIBM/UserData/DB2Extenders/XML/Samples integrated file system directory.

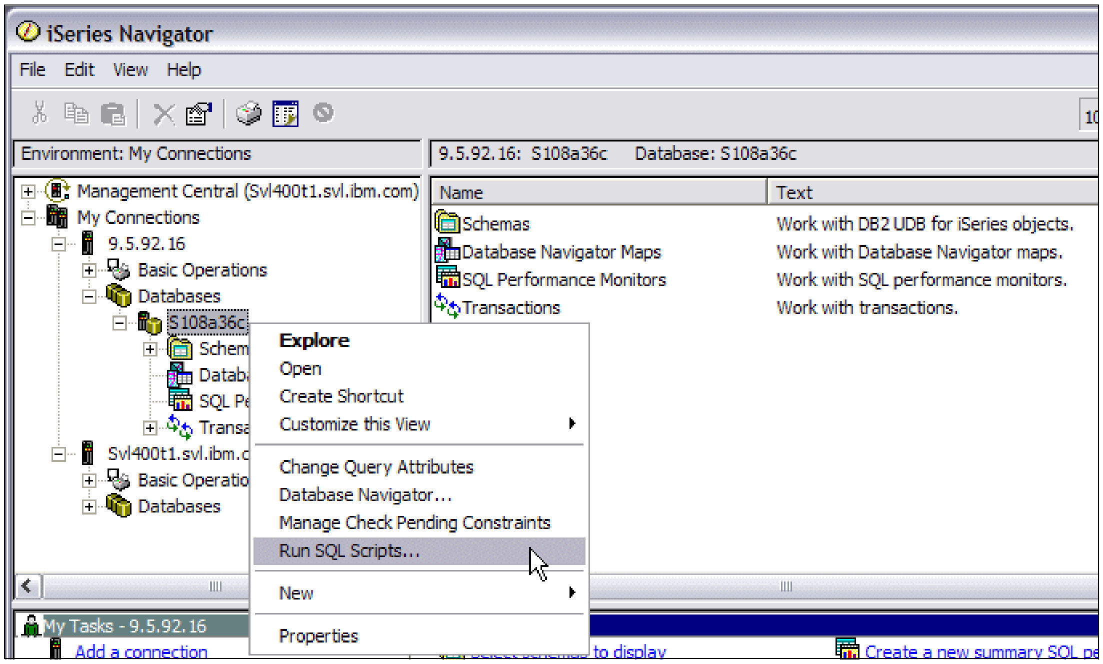


Figure 8-3 Compiling and creating the Sample programs using iSeries Navigator

Enabling the database

When you enable your system's database for XML Extender, XML Extender creates the following objects:

- The DB2XML schema with necessary privileges
- XML Extender UDT, UDFs, and stored procedures
- The XML Extender DTD repository table, DTD_REF, which stores DTDs and information about each DTD
- The XML Extender usage table, XML_USAGE, which stores common information for each column and collection that is enabled for XML Extender

If you know that you will be working with very large documents, you must precreate the XMLVARCHAR and XMLCLOB UDTs to accommodate larger documents. See 8.5, "Handling very large documents" on page 139, for instructions on how to create the DB2XML schema and XMLVARCHAR or XMLCLOB UDTs. Otherwise, you are ready to enable your database for XML Extender.

In the following example, we enable the database using the i5/OS command line:

```
CALL QDBXM/QZXMADM PARM(enable_db &DBNAME)
```

Here *&DBNAME* is the name of your system database. For details about how to enable the database from the iSeries Navigator, refer to *IBM DB2 Universal Database for i5/OS XML Extender Administration and Programming,* SC18-9179.

8.3 XML Extender storage and access methods

After the product is installed and your database is enabled, XML Extender is ready to be used by your application. When you plan an application that uses XML documents, you must decide whether you will store existing XML documents or compose XML documents from existing data. If you plan to store XML documents, you must also decide if you want them to be stored as intact XML documents or decomposed into relational data to be stored into existing tables.

XML Extender provides two different types of storage and access methods for integrating XML documents with DB2 data structures: XML column and XML collection. These methods have different uses and are illustrated in Figure 8-4.

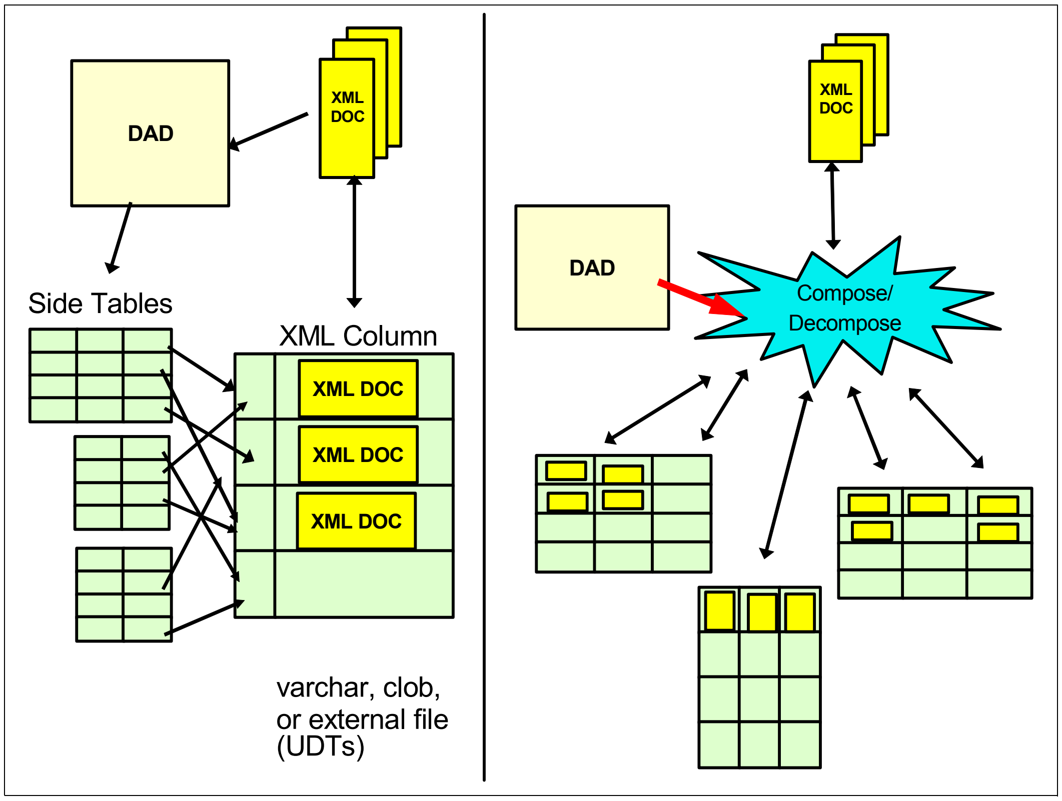


Figure 8-4 XML Extender providing two storage and access methods

The XML Column method helps you store intact XML documents in DB2. The XML Column method works well for archiving documents into a repository. The documents are inserted into columns that are enabled for XML, and then the stored XML documents can be updated, retrieved, and searched. Element and attribute data can be mapped to DB2 Universal Database tables (side tables), which can be indexed for faster searches through your XML repository.

The XML Collection method helps you map XML document structures to DB2 tables so that you can either compose XML documents from existing DB2 data or decompose XML documents, storing the untagged data into DB2 tables.

A key building block that is used by XML Extender is the DAD file. The DAD file defines how structured XML documents are to be processed by XML Extender. Specifically, the DAD file contains the XML to RDB mapping. It contains how a given element or attribute value is associated with a specific column of a table in the DB2. XML Extender uses a DAD file for both the XML Column and Collection methods.

8.3.1 XML Column method

Using the XML Column method allows you to store the entire XML document, as it is, in a column. We recommend that you choose the XML Column method if one or more of the following criteria are met:

- The XML documents already exist or come from an external source and you want to store them in their native XML format. For example, you want to archive documents, such as newspaper articles, orders, and so on, for integrity and auditing purposes.
- The XML documents are read often and updated rarely.
- The performance of updating is not critical.
- You want to keep the intact XML documents external from DB2, in a local or remote file system, and use DB2 for i5/OS for management and search operations.

Before you begin working with XML Extender to store your documents using the XML Column method, you must understand the structure of the XML document:

- Decide in which table you will store the XML documents or pointers to the XML documents. You can create a new table with an XML User-Defined Type Column or alter an existing table to add an XML User-Defined Type Column.
- Identify which elements or attributes might be frequently searched. To perform efficient searches, you can decide whether to create indexes in side tables on the element or attributes that are accessed the most often.
- Decide if you want to automatically validate the XML documents.

A DAD file, which itself is an XML document, specifies how the XML documents that you store in the database are to be handled. In the case of XML columns, the DAD file is only needed if you want to validate your XML documents before you store them, or if you want to store the values of elements or attributes in side tables. *Side tables* are additional tables that are created by DB2 XML Extender that can be indexed to improve performance when searching for the elements or attributes in XML documents that are stored in an XML column. The DAD file for the XML Column method, therefore, defines how a frequently searched element or attribute value is mapped to a specific column of a side table.

The DAD is also used in a different way for the other storage and access method, XML Collection. We discuss this way of using a DAD in detail in 8.3.2, “XML Collection method” on page 127.

If you want to automatically validate and save frequently searched elements and attributes in side tables, you must create a DAD. If you are automatically validating the XML documents with a DTD, you must also store the appropriate DTD document in the DTD repository.

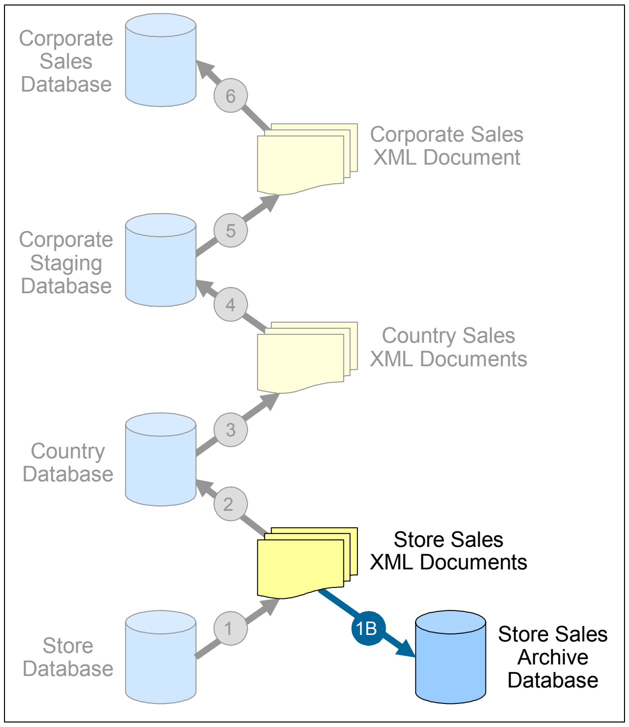
If you created a DAD file for an XML column, you must enable the column to tell the DB2 XML Extender to which XML column this DAD relates. When you enable an XML column, DB2 XML Extender performs the following actions:

- Parses the DAD file
- Creates the side tables with the desired columns corresponding to the elements and attributes in the XML document
- Creates the triggers on the table with the XML user-defined type column (XML column) to synchronize with side tables and perform validation if requested
- Adds a new entry in the DB2XML.XML_USAGE table

  This new entry keeps the relationship between the table, the XML column in this table, the DTD ID, and the DAD file. The DAD file is stored as a CLOB in the XML_USAGE table.

XML column example

In our scenario, the country main offices want to archive the StoreSales XML documents that are generated by each store and sent to the country main offices. This is step 1B of our scenario as shown in Figure 8-5.



Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 8-5 Scenario step 1B

The country main offices want to archive these XML documents before decomposing them into their corporate databases. We use the XML Column method to archive these documents as shown in Figure 8-6.

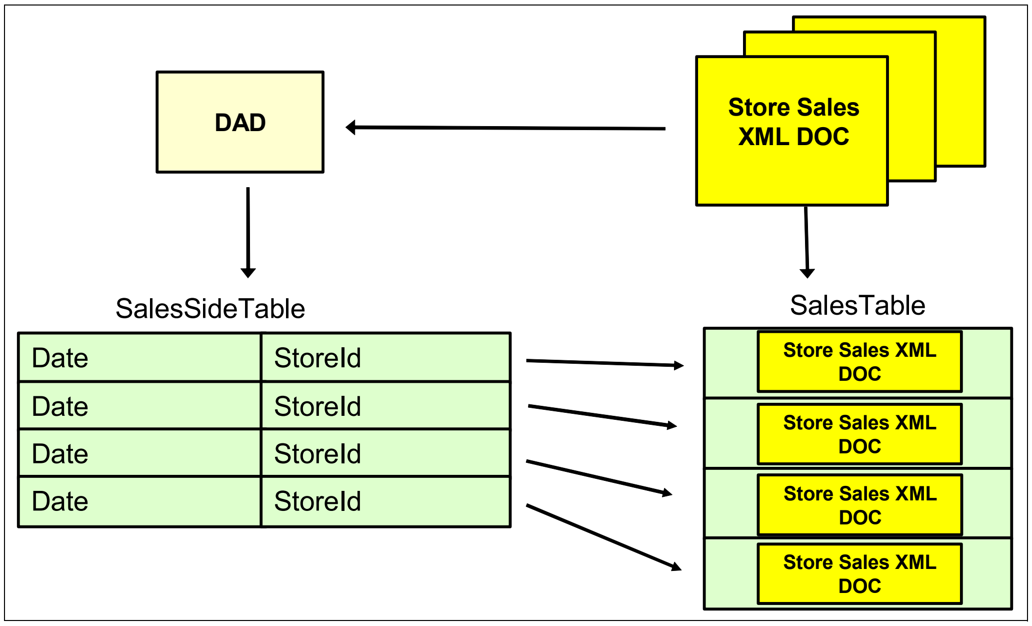


Figure 8-6 Storing StoreSales XML documents using the XML Column method

First, we create a table, called SalesTable. Then, we create a DAD file since we want to validate the documents that we receive from each store. We also want to search for XML documents in this table either by Date or by StoreId. Finally, we enable the column to associate the DAD file to the XML column in SalesTable. The SalesTable table is now ready to store StoreSales XML documents.

Creating the table that contains the XML column

The SalesTable table has only one column that is used to store the StoreSales XML documents received from each store. We want to save the XML documents intact directly into the table. The XML UDT of XMLVARCHAR does not match our needs since we anticipate that the XML documents sent from each store can be up to 1M in size. Instead we must the XMLCLOB UDT since it can accommodate documents as large as 10M. The SQL statement in Example 8-1 creates the table SalesTable for the Colombian main offices.

Example 8-1 SalesTable table definition

```
create table CountryCOL.SalesTable
          (SalesDoc db2xml.XMLCLOB)
```

Inserting the DTD into the DTD_REF table

We want to validate the XML documents submitted by each store to make sure that the XML document matches the XML definition that we expect. We can validate against either a DTD or schema. If validating against a schema, no “registration” is required. However, validating against a DTD requires registration of the DTD with XML Extender. Registration is done by inserting the DTD into the DTD_REF table. The query in Example 8-2 inserts the StoreSales DTD into the DTD_REF table.

Example 8-2 Inserting DTD into the DTD_REF table

```
insert into db2xml.DTD_REF
   values ('project.dtd',                           1
          db2xml.XMLCLOBFromFile('/xmlredbook/StoreXML/StoreSales.dtd'), 2
          0,                                        3
          'author', 'creator', 'updator')           4
```

In Example 8-2, the string in line 1 specifies a DTD ID used in the DAD to identify that DTD. An XML Extender UDF is used in line 2 to retrieve the DTD from the integrated file system and return it as an XMLCLOB UDT. The number of XML columns and XML collections that reference the DTD in their DAD files is initialized at line 3. This number is managed by XML Extender automatically. We initialize it to 0 when we first insert the DTD file into the DTD_REF table. The three values in line 4 are columns in the DTD_REF table that are optional and for our use to record the author of the DTD, the userID of the creator, or the user ID of the last updator of the DTD.

Building the DAD file

The DAD file contains validation information as well as information that describes the date and store ID. The date and store ID are key pieces of information in the XML document that are often searched, so to make the searches faster, we want this information stored in a side table by XML Extender. The DAD in Example 8-3 validates against a DTD, and the DAD file in Example 8-4 validates against a schema. Both identify the date attribute and StoreId element to be extracted and stored in the side table SalesSideTable.

In Example 8-3, the DTD at line 1 identifies the DTD in the DTD_REF table. The value of the <dtdid> element must match the DTD ID used when we “registered” the DTD. If a <dtdid> element is specified, the DTD ID must be registered in the DTD_REF table regardless of the setting of the <validation> element in line 2.

Example 8-3 DAD file that validates against a DTD

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "/dxxsamples/dtd/dad.dtd">
<DAD>
<dtdid>project.dtd</dtdid>          1
<validation>YES</validation>        2
<Xcolumn>
   <table name="SalesSideTable">
      <column name="Salesdate"
      type="DATE"
      path="/StoreSales/@date"
      multi_occurrence="NO"/>
   <column name="StoreId"
      type="varchar(50)"
      path="/StoreSales/StoreId"
      multi_occurrence="NO"/>
   </table>
</Xcolumn>
</DAD>
```

In Example 8-4, the schema file to be used for validation is identified in line 1. Validation is turned on in line 2. In line 3, the use of <Xcolumn> element shows that this DAD is to be used for the XML Column method. One side table, with the name SalesSideTable, is identified in line 4. The side table contains two columns: Salesdate (see line 5) and StoreId (see line 7). The SalesDate column is mapped to the date attribute of the <StoreSales> element in line 6. The StoreId column is mapped to the <StoreId> element in line 8. Both of these values occur only once in the XML document, so `multi_occurrence` is set to `NO` in line 9.

Example 8-4 DAD file that validates against a schema

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "/dxxsamples/dtd/dad.dtd">
<DAD>
<schemabindings>
<nonamespacelocation location="/xmlredbook/StoreXML/StoreSales.xsd"/> 1
</schemabindings>
<validation>YES</validation>           2
<Xcolumn>                              3
   <table name="SalesSideTable">       4
      <column name="Salesdate"         5
      type="DATE"
      path="/StoreSales/@date"         6
      multi_occurrence="NO"/>
   <column name="StoreId"              7
      type="varchar(50)"
      path="/StoreSales/StoreId"       8
      multi_occurrence="NO"/>          9
   </table>
</Xcolumn>
</DAD>
```

Note: Side tables can contain more than one column only if the elements or attributes mapped to the side table occur once in the document. If the element or attribute can occur more than once in the document, then it must be defined in its own side table.

Enabling an XML column

We must now enable the SalesDoc XMLCLOB column in the SalesTable. Enabling this column links the SalesDoc column to the DAD file by creating the side tables and default views as well as triggers. The triggers are activated when the XML documents are stored into the enabled column. Example 8-5 shows how the XML Extender administration command is used from the i5/OS command line to enable the column.

Example 8-5 Enabling an XML column

```
CALL PGM(QDBXM/QZXMADM) PARM(enable_column
                             S108A36C                             1
                             CountryCOL.SalesTable SalesDoc       2
                             ‘/xmlredbook/CountryXML/StoreSales.dad' 3
                             ‘-v' CountryCOL.SALESVIEW)           4
```

In Example 8-5, the name of the system’s database is specified in line 1. The two parameters that specify the name of the table and column where the XML documents will be stored are in line 2. The DAD file is identified in line 3, and the name of the default view that joins the XML table and side tables is defined in line 4. This view is created by XML Extender and is optional.

XML Extender creates the side table, SalesSideTable, and the view, SALESVIEW. XML Extender also adds the column DXXROOT_ID to SalesTable. Since SalesTable has no primary key, DXXROOT_ID is necessary to associate the SalesSideTable together with SalesTable.

Note: DB2 UDB XML Extender for iSeries populates the side tables at insertion or update time with triggers that are created when the XML column is enabled. Therefore, if the column already contains XML documents before it is enabled, the content of these XML documents will not be reflected in the side tables. Do not modify the side table in any way. Updates to the side table should only be made through updates to the original XML document in SalesTable. XML Extender automatically updates the side table when you update the XML document in the SalesTable.

Inserting the XML documents

After the XML column is enabled, we are ready to store the XML documents into the table. The XML documents prepared by each store must be read from the integrated file system and converted to an XMLCLOB type to be inserted into the SalesDoc column of SalesTable. The XML Extender Retrieval UDFs can perform this function. XMLCLOBFromFile is used in Example 8-6, since the SalesDoc column is of type XMLCLOB.

Example 8-6 Inserting an XML document into a table

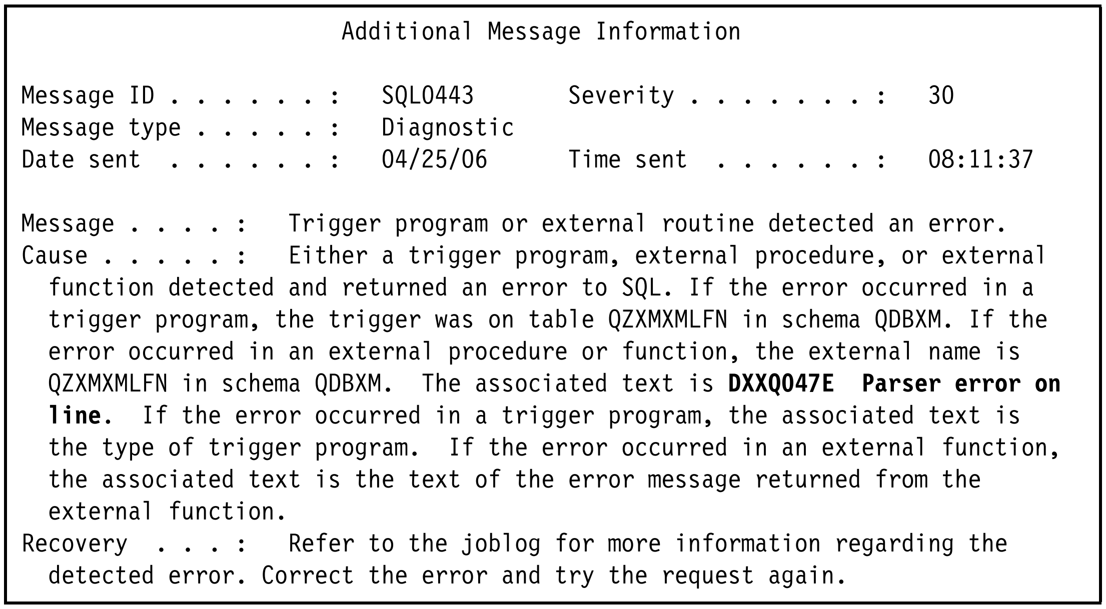
```
INSERT INTO CountryCOL.SalesTable (SalesDoc, DXXROOT_ID)
VALUES(db2xml.XMLCLOBFromFile('/xmlredbook/StoreXML/Store17206.xml'), 1)
```

After successfully inserting the XML document into SalesTable, the side table SalesSideTable is automatically updated with the date and StoreId extracted from the XML document. We can now query the default view using the query in Example 8-7 to find all the Sales XML documents that are sent by a certain store or for a certain date.

Example 8-7 Querying the default view to find XML documents quickly

```
select SalesDoc from CountryCOL.SALESVIEW where StoreId = 172
```

If the XML document that is sent by a store is invalid, the insert fails with an SQL0443 message as shown in Figure 8-7. The reason the insert fails is identified by DXXQ047E. In this example, a parser error occurred because the XML document does not match the schema.



```
                         Additional Message Information

Message ID . . . . . . :   SQL0443       Severity . . . . . . . :   30
Message type . . . . . :   Diagnostic
Date sent  . . . . . . :   04/25/06      Time sent  . . . . . . :   08:11:37

Message . . . . :   Trigger program or external routine detected an error.
Cause . . . . . :   Either a trigger program, external procedure, or external
  function detected and returned an error to SQL. If the error occurred in a
  trigger program, the trigger was on table QZXMXMLFN in schema QDBXM. If the
  error occurred in an external procedure or function, the external name is
  QZXMXMLFN in schema QDBXM.  The associated text is DXXQ047E  Parser error on
  line.  If the error occurred in a trigger program, the associated text is
  the type of trigger program.  If the error occurred in an external function,
  the associated text is the text of the error message returned from the
  external function.
Recovery  . . . :   Refer to the joblog for more information regarding the
  detected error. Correct the error and try the request again.
```

Figure 8-7 SQL0443 received if XML document is invalid

For more detailed information about validation errors, use the XML Extender trace facility. See 8.6.6, “Troubleshooting and error messages” on page 145, for more information about this trace facility.

XML Column method considerations

From our example, the XML Column method is the storage and retrieval method used to archive the StoreSales XML documents. This method is the best for storing entire XML documents in a table in the database. When using this method, consider the following additional points:

- You do not need to create a DAD or enable a column for XML if you do not need side tables. If you only want to validate documents inserted into your XML column, consider using the SVALIDATE or DVALIDATE UDFs.
- When a column is enabled for XML, any changes to the DAD, such as changes to the mapping of side tables or switching to use a different DTD or schema for validation, require that you disable the column and re-enable the column with the new DAD file. Disabling the column results in dropped side tables, which means that any data that is stored in the side tables are lost. The XML documents already stored in the table will remain in your XML table. However, if you re-enable the column, the XML documents that are stored in this table are not reflected in the newly created side tables. You must reinsert these documents into the table to reflect the content in the newly created side tables.

Managing data in XML columns

When you use XML columns to store data, you store an entire XML document in its native format as column data in DB2. This access and storage method allows you to keep the XML document intact, while giving you the ability to search the document, retrieve data from the document, and update the document.

You can manage the data in your XML documents that are stored in XML columns using the UDFs that are provided by XML Extender. The following sections provide a summary of the commonly used XML Extender UDFs. Refer to *IBM DB2 Universal Database for i5/OS XML Extender Administration and Programming,* SC18-9179, for a complete list of XML Extender UDFs.

Storing XML documents in DB2

Storage UDFs import data from a file and return it as an XML UDT to be used by DB2. We have already used the XMLCLOBFromFile UDF to read an XML document stored in the integrated file system and return it as an XMLCLOB type to be inserted into a DB2 table. The XMLVARCHARFromFile UDF is also used often to read an XML document in the integrated file system and return it as an XMLVARCHAR type. Table 8-1 lists the XML Extender Storage UDFs.

Table 8-1 XML Extender Storage functions

Storage UDF	Description
XMLCLOBFromFile	Reads an XML document from a server file and returns the document as an XMLCLOB UDT
XMLFileFromCLOB()	Reads an XML document as a CLOB locator, writes it to an external server file, and returns the file name and path as an XMLFILE UDT
XMLFileFromVarchar()	Reads an XML document from memory as VARCHAR, writes it to an external server file, and returns the file name and path as an XMLFILE UDT
XMLVarcharFromFile()	Reads an XML document from a server file and returns the document as an XMLVARCHAR UDT

Retrieving entire XML documents from DB2

Retrieval UDFs retrieve an entire XML document stored in a DB2 table. To retrieve only the values of an element or attributes from an XML document, use the Extract UDFs. The Content UDF can retrieve an XML document stored in a DB2 table and export it to an external file in the integrated file system. The Content UDF is used in the example in 8.4, “SQL mapping DAD composition example” on page 129, to retrieve the XML document created by the dxxGenXML stored procedure and save it to a file.

The Content UDF shown in Example 8-8 extracts the StoreSales XML Document from SALESVIEW sent in by store 210 on 7 April 2006 and saves it to the integrated file system file ‘/XMLRedbook/CountryXML/StoreSales.xml’.

Example 8-8 Content UDF retrieving the entire XML document

```
select db2xml.Content(
      SalesDoc,                                                      1
      '/XMLRedbook/CountryXML/StoreSales.xml')                       2
from CountryCOL.SALESVIEW
where SalesDate='2006-04-07' and StoreId=210
```

In Example 8-8, the column that contains the XML document is in line 1. The name of the integrated file system file to be created to contain the XML document is specified in line 2. The column type is important. Since SalesDoc is an XMLCLOB type, it is a valid type for this parameter. However, if the column is of type VARCHAR or CLOB, the column needs casting in order to pass this to the Content UDF. For example, to retrieve an XML document stored in a column that is of type VARCHAR, the *columnvarchar* column in line 3 in Example 8-9 must be casted to an XML Extender UDT XMLVARCHAR.

Example 8-9 Casting required when using the Content UDF

```
select db2xml.Content(
      db2xml.XMLVARCHAR(columnvarchar),3
      '/output.xml')
from schemaname.tablewithvarcharcolumn
```

There is only one Retrieval UDF that is an overloaded function. The UDF behaves differently based on where the data is being retrieved. Table 8-2 lists the different input parameters that are accepted by the Content UDF.

Table 8-2 XML Extender Retrieval UDF

Retrieval UDF	Description
Content (XMLFILE)	Retrieves the XML content from a server file and returns it as a CLOB locator
Content (XMLVARCHAR, filename, encoding)	Retrieves the XML content that is stored as an XMLVARCHAR UDT, stores it in an external server file, and returns the file name and path as VARCHAR(512)
Content (XMLCLOB, filename, encoding)	Retrieves the XML content that is stored as an XMLCLOB type, stores it in an external server file, and returns the file name and path as VARCHAR(512)

Extracting element contents and attribute values from DB2.

Extract UDFs retrieve element contents and attribute values from the XML documents that are stored in DB2. To retrieve the names of all the stores who have sent in StoreSales documents, we can use the SQL statement shown in Example 8-10.

Example 8-10 Extracting the content of an element from the XML documents stored in DB2

```
select db2xml.extractVarchar(SalesDoc,
'/StoreSales/StoreId')
from CountryCOL.SalesTable
```

Figure 8-8 shows the result of the query.

```
EXTRACTVARCHAR
172
172
207
207
210
210
42
42
7
7
71
71
87
87
```

Figure 8-8 Results of the SQL query in Example 8-10

The extractCLOB and extractCLOBS UDFs extract a fragment of the XML document, including the XML element tags and subelements as shown in Example 8-11. This function differs from the other extract functions that return only the values of elements and attributes.

Example 8-11 Extracting an XML fragment from XML documents stored in DB2

```
select RETURNEDCLOB
from table (db2xml.ExtractClobs(
                  (Select SalesDoc from CountryCOL.SALESVIEW
                  where SalesDate='2006-04-07' and StoreId=7),
            '/StoreSales/Transactions/Transaction[@type="SALE"]/SalesItem'))
as X
```

The ExtractCLOBS UDF in Example 8-11 returns a table of XML fragments as shown in Figure 8-9. The StoreSales XML document for Store 7 on 7 April 2006 is selected from the SALESVIEW view. The fragments identified by the location path, which is highlighted in bold in Example 8-11, which represents all the SalesItem elements that were sold on that day, are returned by the UDF.

```
RETURNEDCLOB
--------------------------------------------------------------------------------
<SalesItem><Brand name="Pepsi"></Brand><Name>Mt. Dew 20 oz.</Name><Currency>USD
<SalesItem><Brand name="General Electric"></Brand><Name>60 watt bulbs, 4 pk.</N

  2 record(s) selected.
```

Figure 8-9 Results of the SQL query from Example 8-11

Table 8-3 lists the XML Extender Extract UDFs and user-defined table functions (UDTF).

Table 8-3 XML Extender Extract UDFs and UDTFs

Extract UDF	Extract UDTF	Description
extractInteger()	extractIntegers()	Extracts the element content or attribute value from an XML document and returns the data as INTEGER type
extractSmallint()	extractSmallints()	Extracts the element content or attribute value from an XML document and returns the data as SMALLINT type
extractDouble()	extractDoubles()	Extracts the element content or attribute value from an XML document and returns the data as DOUBLE type
extractReal()	extractReals()	Extracts the element content or attribute value from an XML document and returns the data as REAL type
extractChar()	extractChars()	Extracts the element content or attribute value from an XML document and returns the data as CHAR type
extractVarchar()	extractVarchars()	Extracts the element content or attribute value from an XML document and returns the data as VARCHAR(4K) type
extractCLOB()	extractCLOBs()	Extracts a fragment of an XML document and returns the data as CLOB(10K) type; XML Fragments include element and attribute markup and the content of elements and attributes, including subelements
extractDate()	extractDates()	Extracts the element content or attribute value from an XML document and returns the data as DATE type
extractTime()	extractTimes()	Extracts the element content or attribute value from an XML document and returns the data as TIME type
extractTimestamp()	extractTimestamps()	Extracts the element content or attribute value from an XML document and returns the data as TIMESTAMP type

Updating XML documents in DB2

The Update UDF changes the content of specific elements or attributes. You are not required to edit the XML document; XML Extender makes the change for you. When you specify a location path in the Update UDF, the content of every element or attribute with a matching path is updated with the supplied value. If a location path occurs in a document more than once, the Update UDF replaces all of the existing values with the value provided.

ITSO Electronics changed their name to ZYX Electronics. Example 8-12 uses the Update UDF to change all occurrences of "ITSO Electronics" to "ZYX Electronics" in the XML documents stored in the SalesTable table. The double slashes in front of the Brand element (//Brand) in the location path, which is highlighted in bold in Example 8-12, represent any element regardless of ancestry with the name Brand and name attribute of "ITSO Electronics".

Example 8-12 Update UDF changing a value in one or more XML documents

```
update CountryCOL.SalesTable
set SalesDoc =
      db2xml.update(SalesDoc,
      '//Brand[@name="ITSO Electronics"]/@name',
      'ZYX Electronics')
```

Table 8-4 describes the XML Extender Update UDF.

Table 8-4 XML Extender Update UDF

Update UDF	Description
Update()	Updates the XML content stored in an XMLVARCHAR or XMLCLOB column; the element content or attribute value specified by a location path is replaced with the update string specified

Validating XML documents in DB2

XML Extender offers validation UDFs that validate XML documents against either an XML schema or a DTD. The validation functions return "1" if the document is valid or "0" if the document is invalid. More information about an invalid document is available in an XML Extender trace file. See 8.6.6, "Troubleshooting and error messages" on page 145, for more information about the XML Extender trace facility.

The query in Example 8-13 validates all the StoreSales XML documents in the SalesTable table against a schema. The SVALIDATE UDF accepts a file name or CLOB for the XML document parameter. The SalesDoc column is cast from XMLCLOB to CLOB in line 1. The schema file to use during validation is specified in line 2. All the documents are valid based on this query.

Example 8-13 SVALIDATE validating an XML document against a schema

```
select db2xml.svalidate(
        cast(SalesDoc as clob), 1
        '/xmlredbook/StoreXML/StoreSales.xsd') 2
from CountryCOL.SalesTable
```

The query in Example 8-14 uses the DVALIDATE UDF to validate an integrated file system file against a DTD.

Example 8-14 DVALIDATE validating an XML document against a DTD

```
select db2xml.dvalidate('/xmlredbook/StoreXML/Store17207.xml',
'/xmlredbook/StoreXML/StoreSales.dtd') from db2xml.onerow
```

Table 8-5 lists the XML Extender Validate UDFs.

Table 8-5 XML Extender Validate UDFs

Validate UDF	Description
DVALIDATE()	Validates an XML document against a specified DTD and returns a "1' if the document is valid or "0" if not valid
SVALIDATE()	Validates an XML document against a specified schema and returns a "1" if the document is valid or "0' if not valid

Transforming XML documents in DB2

The Extensible Stylesheet Language Transformation (XSLT) consists of a series of markups or instructions that can be used to apply formatting rules to each of the elements inside an XML document. To delete all empty elements, attributes, and white space in the XML documents stored in the SalesTable table, use the Transform UDF and the /xmlredbook/CorpXML/RemoveEmpty.xsl stylesheet as shown in Example 8-15.

Example 8-15 XSLTransformToClob UDF transforming an XML document

```
UPDATE countrycol.SalesTable
SET SalesDoc =
      DB2XML.XSLTransformToClob(
      cast(SalesDoc as CLOB),
'/xmlredbook/CorpXML/RemoveEmpty.xsl',0)
```

Table 8-6 lists the XML Extender Transform UDFs.

Table 8-6 XML Extender Transform UDFs

Transform UDF	Description
XSLTransformToClob()	Transforms an XML document and returns the document as CLOB
XSLTransformToFile()	Transforms an XML document, writes the results into a file, and returns the file name and path as VARCHAR

8.3.2 XML Collection method

The XML Collection method is the second type of storage and access method. It helps you map XML document structures to DB2 tables so that you can either compose XML documents from existing DB2 data or decompose XML documents, storing the untagged data in DB2 tables.

We recommend that you use the XML Collection method in the following cases:

- You have data in existing DB2 tables, and you want to compose XML documents from this data.
- You have XML documents that map well to an existing relational model, and the XML documents contain information that must be stored with existing data in relational tables.
- You have XML documents that must be stored, but you only care about the data, not the XML tags. You want to store pure data in existing or new DB2 tables.

When planning an application that uses the XML Collection method, you must decide whether you want to compose XML documents, decompose XML documents, or do both. Before you begin, you must understand the structure of the XML document that you plan to compose or decompose:

- For decomposition, decide in which DB2 table or tables you will store the data that is extracted from XML documents. For composition, decide on which table or tables you need to query to retrieve the data that you need to create an XML document.
- Analyze the XML document structure to design how the XML document's elements and attributes map to the columns in your relational DB2 tables.
- Decide if you want to automatically validate the XML documents before you decompose them into DB2 or after they are created.

An application that uses the XML Column method uses XML Extender UDTs, XML Extender administration commands, and XML Extender UDFs. For an application that uses the XML Collection method, the application uses XML Extender stored procedures. XML Extender provides stored procedures that enable you to compose or publish one or more XML documents or decompose an XML document and store the untagged data into your DB2 tables.

All the stored procedures require you to pass a DAD because it contains the mapping instructions for manipulating the data. You pass your DAD directly as a file, or you can "register" your DAD by enabling a collection. When you enable a collection, the DAD file is stored in the Db2XML.XML_USAGE table. When the name of an enabled collection is passed, XML Extender retrieves the DAD from the XML_USAGE table.

Table 8-7 lists the XML Extender Composition stored procedures. Composition stored procedures construct one or more XML documents using the data stored in the DB2 tables specified in the DAD file.

Table 8-7 Composition stored procedures

Composition stored procedure	Description
dxxGenXML	► Accepts a DAD file as input ► Inserts one or more generated XML documents into a table
dxxGenXMLCLOB	► Accepts a DAD file as input ► Returns one generated XML document as a CLOB
dxxRetrieveXML	► Accepts an enabled collection name ► Inserts one or more generated XML documents into a table
dxxRetrieveXMLCLOB	► Accepts an enabled collection name ► Returns one generated XML document as a CLOB

Table 8-8 lists the XML Extender Decomposition stored procedures. Decomposition stored procedures break down or shred XML documents and store the untagged data in DB2 tables specified in the DAD file.

Table 8-8 Decomposition stored procedures

Decomposition stored procedure	Description
dxxShredXML	Accepts a DAD file as input
dxxInsertXML	Accepts an enabled collection name

Again, the DAD file is the key in the XML Collection method to describe how to map the hierarchical structure of the XML documents to the actual relational structure of the tables in the database. In the *XML Column method*, the *DAD file is optional* and required only if you want to create side tables to store frequently searched elements in the XML documents that you save. For the *XML Collection method*, the *DAD file is required*. The structure of an XML collection DAD is different from the XML column DAD. In fact, an XML collection DAD can be one of two types, the SQL mapping scheme or the RDB node mapping scheme.

The SQL mapping scheme DAD is based on an SQL select statement. However, because it is based on a select statement, SQL mapping DADs can only be used in composing XML documents. The key to an SQL mapping scheme DAD is to map the SQL select statement to match the structure of the XML document. See 8.4, "SQL mapping DAD composition example" on page 129, for an example of how to work with an SQL mapping DAD to compose an XML document.

The second style of DAD is a relational database (RDB) node mapping DAD. The power of the RDB node mapping DAD is that the same RDB node mapping DAD can be used to compose and decompose XML documents. If a database has a need to decompose incoming XML documents into the database and then compose XML documents out, this means that an RDB node mapping DAD may allow for both of these processes with only a single DAD. The key to an RDB node mapping DAD is to map the underlying physical files to the structure of the XML document. By linking these two together, it allows the data to flow in both directions. For examples of how to use the RDB node mapping DAD, see Chapter 10, “Shredding methodology” on page 155, and Chapter 11, “Composing methodology” on page 181.

The XML Collection method allows you to compose XML documents from existing DB2 data or decompose XML documents and store untagged element or attribute values into DB2 tables. This method is useful for business-to-business (B2B) or electronic data interchange (EDI) applications.

Consider the following additional points when writing an application that uses the XML Collection method:

- You can use WebSphere Studio to help you build DADs to define documents that you want to publish.
- More than one DAD can be defined to describe the same DB2 RDB data to produce customized XML documents based on particular needs.
- RDB node mapping DAD to compose or decompose XML documents is limited to 30 tables.
- Null data in your columns may result in XML documents generated with empty elements or attributes. If empty elements or attributes are not desired, they can be stripped using the Transform UDFs.

For more information and complete descriptions about this method, see *IBM DB2 Universal Database for i5/OS XML Extender Administration and Programming,* SC18-9179.

8.4 SQL mapping DAD composition example

With the two different types of DADs, the SQL mapping DAD is quicker to create. However, an SQL mapping DAD can only by used when composing XML documents, and not when decomposing them. The SQL mapping DAD is based on an SQL statement. After the SQL statement is written that provides the data to be composed into an XML document, the rest of the DAD maps the fields from the select statement to the XML document.

8.4.1 SQL mapping DAD methodology

To illustrate how we can compose using SQL mapping DAD, we show how to go from the CountryXXX database to the County Sales XML document, which is step 3 (Figure 8-10) of the scenario that we introduced in Chapter 2, “Scenario overview” on page 21. The goal of the Country Sales XML document is to list a summary of sales by brand per region for a particular country and date.

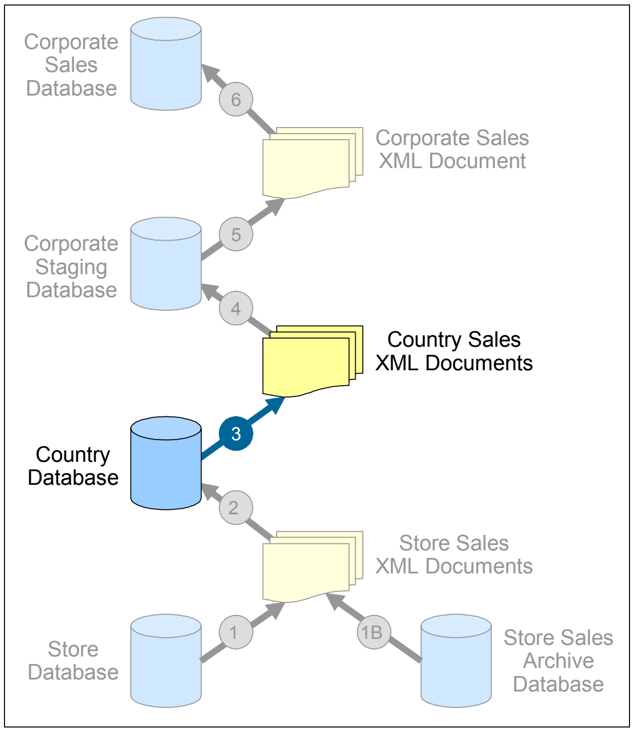

Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 8-10 Scenario step 3

The database model, in Figure 8-11, shows that each country's data is kept in a separate schema. A Sales table lists all of the sales with a foreign key of StoreId. StoreId allows us to link from the Sales table to the Stores table. However, the goal is to summarize by region. The Stores table contains a RegionID for each store that will allow us to link to the Region table and group the sales by regions.

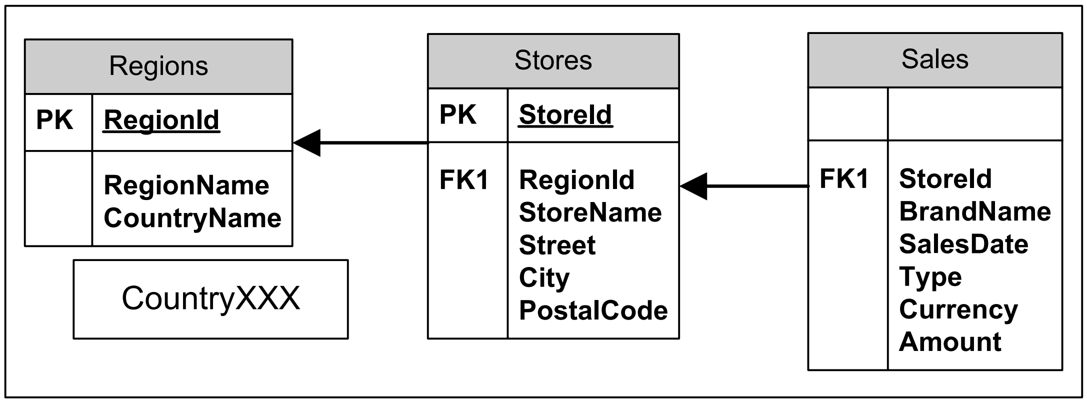


Figure 8-11 CountryXXX schema layout

Next we look at the XML document (shown in Example 8-16) that we are generating. For each brand per region, we need to write the total of the sales per a certain currency. This means that for each region and brand, we will have a repeating set of Sales and Returns elements for each currency with the total values.

Example 8-16 CountrySalesByRegion XML layout

```
<?xml version="1.0" encoding="UTF-8" ?>
<CountrySalesByRegion Date="%YYYY-MM-DD%"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="CountrySalesByRegion.xsd">
   <CountryInfo>
      <Name>%Country%</Name>
   </CountryInfo>
   <Regions>
      <Region>
         <Name>%Region%</Name>
         <Brand>
            <Name>%Brand%</Name>
            <Sales>
               <Currency>%Cur%</Currency>
               <Amount>%XXXXX.XX%</Amount>
            </Sales>
            <Returns>
               <Currency>%Cur%</Currency>
               <Amount>%XXXXX.XX%</Amount>
            </Returns>
         </Brand>
      </Region>
   </Regions>
</CountrySalesByRegion>
```

However, when we look at our tables, we have both sales and returns combined in one table. We must separate the sales from the returns somehow and then do grouping and calculations to combine the individual transactions into one total for that region. To accomplish this, we use an SQL view.

8.4.2 Building the SQL view

In this example, we must pull data from three different tables, separate out the transaction types, group the items together, and run summary functions on the data. We want to do this without changing the underlying data. One method is to keep a second copy of this data someplace that has already processes all of these actions. However, that adds an overhead of maintaining the second copy of data. Instead, we can create an SQL view.

An *SQL view* is a way to mask the underlying data. It does not change the data but processes the data based on the view definition and returns the results back to the calling application. In our sample, there are three different countries each with two dates of data. At first glance, it might appear that we need a separate SQL view and SQL mapping DAD for each one.

Since the data is kept in separate schemas, we create a view per country. This view creates the data as needed but contains the data for all dates. To limit the information to a particular date, we can put the date as part of the where clause in the SQL statement.

It is *not* necessary for us to change the SQL mapping DAD each time to have the country and date of interest used. We can use the override feature to allow us to make one generic SQL view and SQL mapping DAD and then override the SQL statement inside the SQL mapping DAD at each use to include the country and date of interest. This way the SQL view and SQL mapping DAD becomes universal.

For illustrative purposes, we show only one country's SQL view. The complete set of files is included in the additional materials. For information about how to download these materials, see Appendix A, "Additional material" on page 207.

Example 8-17 shows the source of the view for CountryUSA.XMLView.

Example 8-17 Source of CountryUSA.XMLView

```
CREATE VIEW CountryUSA.XMLView as                          1
with salestemp as (                                        2
   SELECT                                                  3
      CHAR(SalesDate, ISO) as salesdate,
      'http://www.w3.org/2001/XMLSchema-instance' as xsi, 4
      'CountrySalesByRegion.xsd' as noname,
      Regions.CountryName as country,
      Regions.RegionName as region,
      Sales.BrandName as brand,
      Sales.Currency as SalesCurrency,
      SUM(Sales.Amount) as SalesAmount                     5
   FROM
      CountryUSA.Regions AS Regions,                       6
      CountryUSA.Stores AS Stores,
      CountryUSA.Sales AS Sales
   WHERE
      Sales.StoreID = Stores.StoreId                       7
      and Stores.RegionID = Regions.RegionID               8
      and Sales.Type = 'SALE'                              9
   GROUP BY
      CHAR(SalesDate, ISO),                                10
      'http://www.w3.org/2001/XMLSchema-instance',
      'CountrySalesByRegion.xsd',
      Regions.CountryName,
      Regions.RegionName,
      Sales.BrandName,
      Sales.Currency),
returntemp as(                                             11
   SELECT
      CHAR(SalesDate, ISO) as salesdate,
      'http://www.w3.org/2001/XMLSchema-instance' as xsi,
      'CountrySalesByRegion.xsd' as noname,
      Regions.CountryName as country,
      Regions.RegionName as region,
      Sales.BrandName as brand,
      Sales.Currency as ReturnCurrency,
      SUM(Sales.Amount) as ReturnAmount
   FROM
      CountryUSA.Regions AS Regions,
      CountryUSA.Stores AS Stores,
      CountryUSA.Sales AS Sales
   WHERE
      Sales.StoreID = Stores.StoreId
      and Stores.RegionID = Regions.RegionID
      and Sales.Type = 'RETURN'                            12
   GROUP BY
      CHAR(SalesDate, ISO),
      'http://www.w3.org/2001/XMLSchema-instance',
      'CountrySalesByRegion.xsd',
      Regions.CountryName,
      Regions.RegionName,
      Sales.BrandName,
      Sales.Currency)
SELECT                                                 13
   salestemp.salesdate,
   salestemp.xsi,
   salestemp.noname,
   salestemp.country,
   salestemp.region,
   salestemp.brand,
   salestemp.SalesCurrency,
   salestemp.SalesAmount,
   returntemp.ReturnCurrency,
   returntemp.ReturnAmount
FROM
   salestemp salestemp                                 14
   left outer join returntemp returntemp               15
      on salestemp.region = returntemp.region          16
      and salestemp.brand = returntemp.brand
      and salestemp.salescurrency = returntemp.returncurrency
UNION DISTINCT                                         17
SELECT
   returntemp.salesdate,
   returntemp.xsi,
   returntemp.noname,
   returntemp.country,
   returntemp.region,
   returntemp.brand,
   salestemp.SalesCurrency,
   salestemp.SalesAmount,
   returntemp.ReturnCurrency,
   returntemp.ReturnAmount
FROM
   salestemp salestemp
   right outer join returntemp returntemp              18
      on salestemp.region = returntemp.region
      and salestemp.brand = returntemp.brand
      and salestemp.salescurrency = returntemp.returncurrency
```

In Example 8-17, we start with a create statement in line 1. This is changed for each view that we create. To split the sales from the returns, we use a common table expression in line 2. Common table expressions allow us to generate a temporary table of data to work with.

We start building our common table with a select at line 3. At line 4, we build in our constants. Since the XML document that will be built needs to map the fields from the SQL statement to the XML document, anything that needs to be filled in on the XML document must be in the SQL statement. The action that sums up all the transactions happens at line 5.

Starting at line 6, we list the three tables that need to be joined to obtain all the data required. In 7, we link Sales to Stores, and in line 8, we link Stores to Regions. This join allows us to group the items by region even though the Sales table only has a StoreId. At line 9, we limit this common table to just sales.

At line 10, we group the records in the format that we need. Also, once these groups are complete, the sum from line 5 is only a total for that group. This gives us exactly what we need for the XML document. At line 11, we start the same process of creating a common table for returns, which is restricted at line 12.

Now that we have separated, grouped, and totaled the sales and returns data, we combine this data to put the records for matching regions and brands together. It is possible that, on these days, we had sales without returns or returns without sales, which makes joining the two common tables challenging. If we do a left outer join, we get sales and matching returns. However, if a brand had a return without a sale, the left outer join drops that row.

We need the concept of a full outer join (see Figure 8-12 for the various join types). Unfortunately, DB2 for i5/OS does not support a full outer join. However, we can simulate this by taking a left outer join and a right outer join and doing a union distinct. This gives us all sales with matching returns and all returns with matching sales, as well as removes any duplicate rows. The rows that are in duplicate are the result of the inner join.

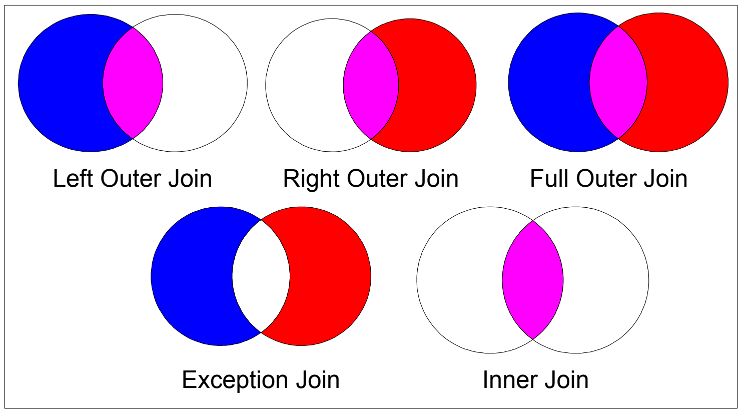


Figure 8-12 Various join types

We start our select at line 13 in Example 8-17 on page 132. At line 14, we start our join by selecting the sales common table. At line 15, we do our left outer join to the returns common table to get all sales and matching returns. We join them at line 16 by the common criteria. At line 17, we do the union distinct to pull the two pieces together and strip out the duplicate values. Finally, at line 18, we do our right outer join to pull in any brands that had returns without sales.

8.4.3 Creating the SQL mapping DAD

Now that we have an SQL view that provides the information in a format that the DAD can transform into the XML document, we must link the SQL statement to the structure of the XML document. The SQL mapping DAD provides this link. Example 8-18 shows the source of the CountrySalesByRegion SQL mapping DAD.

Example 8-18 Source of the CountrySalesByRegion.dad file

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "/dxxsamples/dtd/dad.dtd">
<DAD>
<schemabindings>
<nonamespacelocation location="CountrySalesByRegion.xsd"/>
</schemabindings>
<validation>NO</validation>
<Xcollection>
```

```
<SQL_stmt>                                                              1
SELECT salesdate, xsi, noname, country, region, brand,
       SalesCurrency, SalesAmount, ReturnCurrency, ReturnAmount
FROM countryusa.xmlview
WHERE 0 = 1                                                             2
ORDER BY salesdate, region, brand                                       3
</SQL_stmt>
<prolog>?xml version="1.0" encoding="UTF-8"?</prolog>                   4
<root_node>
<element_node name="CountrySalesByRegion">                              5
   <attribute_node name="Date">
      <column name="salesdate"/>                                        6
   </attribute_node>
   <attribute_node name="xmlns:xsi">
      <column name="xsi"/>
   </attribute_node>
   <attribute_node name="xsi:noNamespaceSchemaLocation">
      <column name="noname"/>
   </attribute_node>
   <element_node name="CountryInfo">                                    7
      <element_node name="Name">
         <text_node><column name="country"/></text_node>                8
      </element_node>
   </element_node>   <!-- end tag CountryInfo -->
   <element_node name="Regions">                                        9
      <element_node name="Region" multi_occurrence="YES">               10
         <element_node name="Name">
            <text_node><column name="region"/></text_node>
         </element_node>
         <element_node name="Brand" multi_occurrence="YES">             11
            <element_node name="Name">
               <text_node><column name="brand"/></text_node>
            </element_node>
            <element_node name="Sales">                                 12
               <element_node name="Currency">
                  <text_node><column name="SalesCurrency"/></text_node>
               </element_node>
               <element_node name="Amount">
                  <text_node><column name="SalesAmount"/></text_node>
               </element_node>
            </element_node>
            <element_node name="Returns">                               13
               <element_node name="Currency">
                  <text_node><column name="ReturnCurrency"/></text_node>
               </element_node>
               <element_node name="Amount">
                  <text_node><column name="ReturnAmount"/></text_node>
               </element_node>
            </element_node>
         </element_node>  <!-- end tag Brand -->
      </element_node>  <!-- end tag Region -->
   </element_node>    <!-- end tag Regions -->
</element_node>      <!-- end tag CountrySalesByRegion-->
```

```
</root_node>
</Xcollection>
</DAD>
```

In Example 8-18 on page 134, inside the SQL mapping DAD, we must first provide the SQL statement, which we do at line 1. We select the fields from our SQL view. At line 2, we add a WHERE clause of 0 = 1. The WHERE condition will never be true so when the SQL mapping DAD runs by itself, it will not produce any rows. If we leave this out and run the generate without an override, the results contain multiple dates worth of data. Since date is not multi-occurring, we add this as a safeguard to prevent error messages.

The ORDER BY at line 3 is the important part of the SQL statement. When looking logically at how the XML document is nested, we must match the ORDER BY to how the XML document that will be built. A similar discussion is in Chapter 3, "Using SQL to compose XML" on page 33. As we build levels in the XML, we start at the top date level. Inside that level, we loop at the region level. Inside that level, we loop at the brand level. Therefore, for the XML Extender product to build the XML file correctly, the data must be ordered in this manner.

We start to build the standard XML document at line 4. At line 5, we build our root node. At line 6, we have our first mapping from the SQL statement. We use the column element to link the field in the SQL select statement. At line 7, we build our next level of the XML document by first creating the CountryInfo element. At line 8, we map the field country to the text node of the CountryInfo element.

At line 9, we build the Regions wrapper element. At line 10, we start the first Region element inside the Regions wrapper. The key point here is that Region is marked as multi-occurring. This is also a flag to us that the ORDER BY statement needs this field as part of the ordering key. At line 11, we have a similar multi-occurring keyword on the Brand element inside of the Region element. This too needs to be part of the ORDER BY statement.

At lines 12 and 13, we build the Sales and Returns elements inside of a brand. We have the column elements inside the text node elements to map the fields from the view into the XML document. Following that, all of the opened elements are closed.

Again, the ORDER BY clause of the SQL statement is important. The results shown in Example 8-19 are from the following SQL query:

```
SELECT salesdate, xsi, noname, country, region, brand,
        SalesCurrency, SalesAmount, ReturnCurrency, ReturnAmount
FROM countryusa.xmlview
ORDER BY salesdate, region, brand
```

Example 8-19 SQL query results where salesdate is the first column

```
SALESDATE   XSI                                        NONAME
2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegion.xsd
```

The first column in the ORDER BY, salesdate, is the key to the number of XML documents that are generated. Since there are two different result dates, 2006-04-06 and 2006-04-07, two XML documents are composed by a DAD file that included this SQL statement. If the intent is to generate one XML document, the first column in the ORDER BY must have the same value. You can accomplish this by adding a column with a constant value.

Including the following SQL statement in a DAD produces one XML document because there is only one unique value in the first column:

```
SELECT 1 as X, salesdate, xsi, noname, country, region, brand,
       SalesCurrency, SalesAmount, ReturnCurrency, ReturnAmount
FROM countryusa.xmlview
ORDER BY X, salesdate, region, brand
```

Example 8-20 shows the result of including the SQL statement.

Example 8-20 SQL query results where a constant of 1 makes up the first column

```
X  SALESDATE   XSI                                        NONAME
 1  2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-06  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
 1  2006-04-07  http://www.w3.org/2001/XMLSchema-instance  CountrySalesByRegio
```

For this scenario, an override SQL statement is used that limits the results to a particular date, ensuring that only an XML document is produced.

8.4.4 Executing the SQL mapping DAD composition

At this point, the view is completed and the SQL mapping DAD is built. Now we must apply the DAD to the database to compose our XML document. To do this, we invoke the XML Extender product with the DAD and a temporary table. This temporary table is populated with the copy of the XML document. We can then call an XML Extender UDF to move that copy of the XML file in the temporary table to the integrated file system.

The SQL view that the SQL mapping DAD is pulling from contains information for all dates from a particular county. Since the goal of the XML document is to be specific to a single country and date, we limit the selection from a particular SQL view to a single date. We can do this by adding a WHERE clause to the select statement in the SQL mapping DAD and changing the FROM clause to point to the correct schema and view. However, it is a hassle to go into the SQL mapping DAD and change these for each run. To work around this, we can dynamically override the SQL statement in the SQL mapping DAD.

Since we have six different XML documents to be generated, we must call this process six times. In the additional materials available that accompany this book, the script for all six calls is in the Run GenCountryXML.sql file in the /XMLRedbook/SQL Source directory. A partial copy of that source is included in Example 8-21.

Example 8-21 Partial source of Run GenCountryXML.sql

```
create schema qxmltemp;                                                    1
create table qxmltemp.result_tab(                                          2
   doc db2xml.XMLVarchar,                                                  3
   dxx_valid integer);                                                     4

CREATE PROCEDURE qxmltemp.GENXML                                           5
   (IN DBName VARCHAR(18),                                                 6A
   IN DADFile VARCHAR(255),                                                7A
   IN ResultTab VARCHAR(160),                                              8A
   IN ResultCol VARCHAR(129),                                              9A
   IN ValidCol VARCHAR(129),                                               10A
   IN Override VARCHAR(1024))                                              11A
LANGUAGE C
NOT DETERMINISTIC
MODIFIES SQL DATA
EXTERNAL NAME 'DXXSAMPLES/GENXMLO'                                         12
PARAMETER STYLE GENERAL;

CALL qxmltemp.GenXML(                                                      13
   'S108A36C',                                                             6B
   '/XMLRedbook/CountryXML/CountrySalesByRegion.dad',                      7B
   'qxmltemp.result_tab',                                                  8B
   'doc',                                                                  9B
   ' ',                                                                    10B
   '-o SQL_OVERRIDE                                                        11B
   SELECT salesdate, xsi, noname, country, region, brand,
      SalesCurrency, SalesAmount, ReturnCurrency, ReturnAmount
   FROM countryusa.xmlview                                                 11C
   WHERE salesdate = ''2006-04-06''                                        11D
   ORDER BY salesdate, region, brand');

select db2xml.Content(                                                     14
      doc,                                                                 15
      '/XMLRedbook/CountryXML/CountrySalesByRegionUSA20060406.xml',        16
      'ibm-819')                                                           17
from qxmltemp.result_tab;                                                  18

commit;                                                                    19
drop schema qxmltemp;                                                      20
```

When looking at the pieces that we need to build together, we start at line 1 in Example 8-21 by creating a temporary schema. In this process, we create a temporary space to work with and then clean it up. If this is something that you will do often, you may want to make this setup permanent and not repeat it at each execution. At line 2, we create a temporary table. This table has two fields. The XMLVarchar field in line 3 holds our document. The field in line 4 has an integer to hold a validation result. In our example, we do not use validation.

In line 5, we create a stored procedure that allows us to access the XML Extender product. We can see in lines 6A through 11A that there are six input parameters. These parameters are the database name, the location of the SQL mapping DAD, the table to put the results in, the column in that table for the XML document, the column for the validity check, and the override parameter. In line 12, we can see the external program in the DXXSAMPLES library that is called to do this work.

Line 13 in Example 8-21 contains the call to the stored procedure. The next lines, labeled 6B through 11B, are the input parameters that match with 6A through 11A, linking 6A to 6B, and so on. The override parameter starting at line 11B allows us to use the same SQL mapping DAD for all of the compositions. At line 11C, we change the FROM clause to point to the SQL view of the country in which we are interested. At line 11D, we add the WHERE clause that limits us to a particular date. Since the value we are comparing to is a literal string, we must pass it in single quotation marks. However, since it is a string parameter itself, as set at line 11A, it is already in single quotation marks. This means that we must put two single quotation marks to get one in the actual SQL statement. This call to the XML Extender product populates the temporary table with the XML document.

To obtain the content from the database file, we use another feature, the *Content UDF*, of the XML Extender product. Since it is a UDF and not a stored procedure, we cannot call it. Instead, we use a select statement in line 14. In line 15, we have the first parameter, which is the column that contains the XML document. In line 16, we give the location that we want the XML document to be written to in the integrated file system. The third parameter in line 17 is the code page in which the integrated file system file should be written. Since this is a select statement, we must provide a from file in line 18. This file should contain the XML document, which is the temporary file in our sample. Our XML document is now complete in the integrated file system.

Now we must clean up what we have done. Depending on whether you are running under commitment control, there may be locks on the temporary file. To release the locks so that we can clean it up, we issue a commit in line 19. Finally, in line 20, we drop the entire temporary schema. Again, if we use this process often, we can modify it to use a permanent schema.

8.5 Handling very large documents

If you work with very large XML documents, the variable sizes that XML Extender uses in its UDTs and stored procedures may need to be reset. In addition, you may need to make changes in your application programs or system environment.

8.5.1 Redefining user-defined types

When XML Extender enables the database, the XMLVARCHAR UDT is created as VARCHAR(3000). The XMLCLOB UDT is created as CLOB(10M). If you know that the XML documents you will be storing or composing will be greater than these sizes, you must change these UDTs. To change the size of one or both of these UDTs, the UDT must be created before you enable the database for XML Extender.

To create a UDT before enabling the database, enter the following commands from an interactive SQL session:

```
CREATE SCHEMA DB2XML
CREATE DISTINCT TYPE DB2XML.XMLVARCHAR AS VARCHAR(3000) WITH COMPARISONS
CREATE DISTINCT TYPE DB2XML.XMLCLOB AS CLOB(10M)
```

Replace the size of *3000* and *10M* with the size desired.

If your database is already enabled for XML Extender, you must disable your database, follow the previous steps to create the UDTs, and then enable your database again. You may have tables that are already using XML UDTs or that are enabled for the XML Column or XML Collection methods. If so, you may need to save your data and drop XML collection tables or XML column tables before you disable your database. After the database is re-enabled, you must recreate your tables and restore your data. This process is described in Chapter 2, "Administration, Saving and restoring for XML columns and XML collections," of *IBM DB2 Universal Database for i5/OS XML Extender Administration and Programming,* SC18-9179.

8.5.2 Redefining stored procedure parameters

The DB2XML.DXXShredXML and DB2XML.DXXInsertXML stored procedures as originally created by XML Extender accept only a 1 MB input buffer. This limits the size of the XML document that can be shredded to 1 MB. If you know that you will be decomposing XML documents greater than 1 MB, you can either switch to use the DB2XML.dxxShredXML100MB or DB2XML.dxxInsertXML100MB stored procedures (V5R3 and later) or redefine these stored procedures to increase the input buffer.

The resultDoc parameter of the DB2XML.dxxGenXMLClob and DB2XML.dxxRetrieveXMLClob stored procedures is defined as 1 MB. If you know that you will generate XML documents greater than 1 MB, you can either switch to use the DB2XML.dxxGenXML or DB2XML.dxxRetrieveXML stored procedures or redefine these stored procedures to increase the parameter size.

You can redefine these stored procedures after the database is enabled for XML Extender. The SQL statements to drop and recreate these stored procedures are shown in the following examples. You must first drop the existing stored procedure and then recreate it, replacing the *1M* in the line marked in bold, in the following examples, with the desired length.

Example 8-22 shows the SQL to redefine the dxxShredXML stored procedure.

Example 8-22 SQL to redefine the dxxShredXML stored procedure

```
DROP SPECIFIC PROCEDURE DB2XML.DXXSHREDXML

CREATE PROCEDURE DB2XML.DXXSHREDXML(
        IN     DAD_BUF     CLOB(100K),
        IN     XMLOBJ      CLOB(1M),
        OUT    ERRCODE     INTEGER,
        OUT    ERRMSG      VARCHAR(1024)
        )
        LANGUAGE C++
        SPECIFIC DB2XML.DXXSHREDXML
        NOT DETERMINISTIC
        MODIFIES SQL DATA
        FENCED
        CALLED ON NULL INPUT
        EXTERNAL NAME 'QDBXM/QZXMOSHDX'
        PARAMETER STYLE DB2SQL
```

Example 8-23 shows the SQL to redefine the dxxInsertXML stored procedure.

Example 8-23 SQL to redefine the dxxInsertXML stored procedure

```
DROP SPECIFIC PROCEDURE DB2XML.DXXINSERTXML

CREATE PROCEDURE DB2XML.DXXINSERTXML(
        IN   COLLECTIONNAME    VARCHAR(128),
        IN   XMLOBJ            CLOB(1M),
        OUT  ERRCODE           INTEGER,
        OUT  ERRMSG            VARCHAR(1024)
        )
        LANGUAGE C++
        SPECIFIC DB2XML.DXXINSERTXML
        NOT DETERMINISTIC
        MODIFIES SQL DATA
        FENCED
        CALLED ON NULL INPUT
        EXTERNAL NAME 'QDBXM/QZXMOINSX'
        PARAMETER STYLE DB2SQL
```

Example 8-24 shows the SQL to redefine the dxxGenXMLCLOB stored procedure.

Example 8-24 SQL to redefine the dxxGenXMLCLOB stored procedure

```
DROP SPECIFIC PROCEDURE DB2XML.DXXGENXMLCLOB

CREATE PROCEDURE DB2XML.DXXGENXMLCLOB(
        IN     DAD_BUF       CLOB(100K),
        IN     OVERRIDE_TYPE INTEGER,
        IN     OVERRIDE      VARCHAR(16336),
        OUT    RESULTDOC     CLOB(1M),
        OUT    VALID         INTEGER,
        OUT    NUM_DOCS      INTEGER,
        OUT    ERRCODE       INTEGER,
        OUT    ERRMSG        VARCHAR(1024)
        )
        LANGUAGE C++
        SPECIFIC DB2XML.DXXGENXMLCLOB
        NOT DETERMINISTIC
        MODIFIES SQL DATA
        FENCED
        CALLED ON NULL INPUT
        EXTERNAL NAME 'QDBXM/QZXMOGENC'
        PARAMETER STYLE DB2SQL
```

Example 8-25 shows the SQL to redefine the dxxRetrieveXMLCLOB stored procedure.

Example 8-25 SQL to redefine the dxxRetrieveXMLCLOB stored procedure

```
DROP SPECIFIC PROCEDURE DB2XML.DXXRETRIEVEXMLCLOB

CREATE PROCEDURE DB2XML.DXXRETRIEVEXMLCLOB(
        IN   COLLECTIONNAME    VARCHAR(128),
        IN   OVERRIDE_TYPE     INTEGER,
        IN   OVERRIDE          VARCHAR(16336),
        OUT  RESULTDOC         CLOB(1M),
        OUT  VALID             INTEGER,
        OUT  NUM_DOCS          INTEGER,
        OUT  ERRCODE           INTEGER,
        OUT  ERRMSG            VARCHAR(1024)
        )
        LANGUAGE C++
        SPECIFIC DB2XML.DXXRETRIEVEXMLCLOB
        NOT DETERMINISTIC
        MODIFIES SQL DATA
        FENCED
        CALLED ON NULL INPUT
        EXTERNAL NAME 'QDBXM/QZXMORTRC'
        PARAMETER STYLE DB2SQL
```

8.5.3 Application program and system changes

Your application programs and other system or DB2 resources may need to be adjusted in order for the stored procedures to successfully decompose and generate large documents. Large documents must be passed as pointers, and programs need to be compiled to allow for teraspace pointers. Example 8-26 shows sample lines of the C code taken from the following Technote to define a CLOB parameter of size >= 64M:

http://www-1.ibm.com/support/docview.wss?uid=swg21177331

Example 8-26 Defining and allocating space for a CLOB parameter

```
SQL TYPE is CLOB(100M) *xmlobjp;

xmlobjp = (_Packed struct xmlobjp_t *) malloc( sizeof (*xmlobjp));
if (xmlobjp ==NULL) {
   fprintf(stderr, "Malloc() error.\n");
   goto exit;
}
```

You must use the following compiler options when compiling and building your application programs that pass a CLOB of size >= 16M:

- TERASPACE(*YES *TSIFC)
- STGMDL(*TERASPACE)

You may also need to use the following PTFs when working with XML Extender and large documents:

- V5R3M0
  - 5722DE1 SI24827: XML Extender Fix Pak 13
  - 5722SS1 SI24353: dxxShredXML100 MB can be called from a Java program
  - 5722SS1 SI23266: Native JDBC Driver with string parms > 8M
  - 5722SS1 SI23500: fread may fail when data exceeds 16M
- V5R4M0
  - 5722DE1 SI24837: XML Extender Fix Pak 13
  - 5722SS1 SI22725: dxxShredXML100 MB can be called from a Java program
  - 5722SS1 SI23499: Native JDBC Driver with string parms > 8M
  - 5722SS1 SI24195: fread may fail when data exceeds 16M

Splitting large documents

XML Extender decomposition limits the number of rows that can be inserted into each table from one document to 500,736 rows. Therefore, it may be necessary to split your large XML documents and perform multiple calls to the XML Extender stored procedure to decompose.

8.6 Additional considerations

In the following sections, we describe additional considerations that are important and useful when developing your XML Extender application.

8.6.1 SQL naming convention

When using SQL, XML Extender requires that you use the *SQL Naming Convention (schema-name.table-name).

8.6.2 Migration

Whenever an XML Extender PTF is applied and your database is already enabled for XML Extender, run migration to update the DB2XML schema or update triggers for existing enabled XML columns. Migration may create new UDFs or a stored procedure, or it may update existing ones. The XML Extender Migration program can also change the XML_USAGE and DTD_REF tables.

From the i5/OS command line, enter:

```
CALL QDBXM/QZXMMIGV
```

If an XML Extender PTF requires migration (the PTF cover letter indicates this), the PTF must not be removed after migration is complete. To ensure that the PTF is not removed, the PTF should be permanently applied.

8.6.3 Commitment control

Follow these guidelines for commitment control when running XML Extender UDFs and stored procedures:

- XML Extender requires COMMIT(*CS) and assumes that the application handles COMMIT and ROLLBACK. XML Extender never performs these actions.
- We recommend that, within one commitment control definition, the application design place any call to an XML Extender UDF or stored procedures, and any INSERT, UPDATE, or DELETE that uses XML Extender triggers. This commit definition should be completed by your application with the appropriate COMMIT or ROLLBACK.
- If another SQL action is performed that affects the same row that is already updated by an XML Extender UDF or another SQL query, then the application is required to explicitly perform a COMMIT after the first action.

See the following Technote for more information:

http://www-1.ibm.com/support/docview.wss?uid=swg21023858

8.6.4 Journal management

Because XML Extender requires commitment control, tables that XML Extender writes to must have journaling started. For example, the tables that XML Extender insert into during decomposition must allow for journaling. A result table into which XML Extender stores a composed document must also allow for journaling.

See the following Technote for more information:

http://www-1.ibm.com/support/docview.wss?uid=swg21205958

8.6.5 Save and restore

If you are doing a backup and restore of your entire system for disaster recovery, there are no special considerations for XML Extender objects.

If you are moving only one application or many applications that use XML Extender UDT, UDFs, or stored procedures from one system to another (not a complete system restore), you must consider the following points:

- Do not save, restore, or delete the DB2XML schema (library). Enable the database for XML Extender on the other system to create the DB2XML schema.
- You can restore user-created schemas that contain database files used by XML Extender under the following conditions:
  - Schemas that contain XML collections, but do not contain XML-enabled columns, *can* be restored at the library level, as long as the database on the new system is enabled for XML Extender. If the XML collection was enabled on the old system, you must re-enable the XML collection on the new system.
  - Schemas that contain columns of XML user-defined types (XMLCLOB, XMLVarchar, and so on) *can* be restored at the library level, as long as the column is *not* enabled for XML and the database on the new system is enabled for XML Extender.
  - Schemas that contain columns that are enabled for XML *cannot* be restored at the library level. The base table and the side tables (database files) can be restored using the RSTOBJ command.

The following Technote provides steps for restoring schemas with database files that are used with XML collections and XML columns:

http://www-1.ibm.com/support/docview.wss?uid=swg21039538

8.6.6 Troubleshooting and error messages

All the embedded SQL statements and DB2 command line interface (CLI) calls that are made by XML Extender, as well as those made by your application program, generate codes that are stored in the job log and indicate whether the embedded SQL or DB2 CLI calls ran successfully. Your application program can retrieve SQLSTATE and error messages and use this diagnostic information to isolate and fix problems in your program.

XML Extender also provides return codes to help resolve problems. When you receive a return code from a stored procedure, check the /QIBM/ProdData/DB2Extenders/XML/include/dxxrc.h file, which matches the return code with an XML Extender error message number and the symbolic constant.

All XML Extender error messages are listed in Chapter 13, “Troubleshooting,” in *IBM DB2 Universal Database for i5/OS XML Extender Administration and Programming,* SC18-9179.

Sometimes the source of a problem is not easily diagnosed. In these cases, try the XML Extender trace facility. This trace facility records XML Extender activity.

To start the trace from an i5/OS command line, enter:

```
call QDBXM/QZXMTRC PARM(on user_profile ‘trace directory’)
```

Here *user_profile* is the user profile of the person who is doing the troubleshooting. *trace directory* is the name of an existing directory where the trace file will be created.

To end the trace from the i5/OS command line, enter:

```
call QDBXM/QZXMTRC PARM(off user_profile)
```

For example, a call to the SVALDIATE UDF returned a 0, which means that validation failed. The XML Extender trace file shows that a file could not be opened.

The trace file tells us that an exception occurred and that the primary document /xmlredbook/StoreXML/Store17209.xml could not be opened. See Figure 8-13.

Usually the most useful information in an XML Extender trace file is found at the end of the file where the error has ended the processing. Because XML Extender continues to append activity to the bottom of the file as long as trace is on, the trace file can grow quite large and slow down performance of XML Extender. We recommend that you turn off trace after you complete the diagnostic activity for a failing scenario.

```
dxxInit:                    tmp dir NULL, file thresh 131072
initICU:                    entered
initICU:                    db2CodePage  = 37
initICU:                    app code page 37, db code page 37
initICU:                    got icu converters db conv = 80000000, app conv = 0
dxxInitializeParser:        Shared library has been initilized successfully
dxxInitializeParser:        Exit, return rc=0
dxxInit:                    dxxInitializeParser rc = 0, memthresh 131072
dxxSchemaValidateFileExternal:Entered, input
docFile='/xmlredbook/StoreXML/Store17209.xml'
dxxSchemaValidateFileExternal:codepage is 37
dxxSchemaValidate:          Entered, doc='/xmlredbook/StoreXML/Store17209.xml'
schemaFile='/xmlredbook/StoreXML/StoreSales.dtd'
namespacePair='(NULL)'
dxxInitializeParser:        Shared library has been initilized successfully
dxxInitializeParser:        Exit, return rc=0
dxxSchemaValidate:          call dxxInitializeParser(), return rc=0
DXX_getEncoding:            Entered.
DXX_getEncoding:            Exit, 69 62 6d 30 errCode = 0
dxxSchemaValidate:          CodePage found:
dxxGetAbsPath:              Exit
dxx_getFileSize:            full path =/xmlredbook/StoreXML/StoreSales.dtd rc=0
dxx_getFileSize:        size =503
dxxSchemaValidate:      setting NoNamespace API for schema file
dxxSchemaValidate:      SystemId  parse err1: line:0 char:0 An exception occurred!
Type:RuntimeException, Message:The primary document entity could not be opened.
Id=/xmlredbook/StoreXML/Store17209.xml
dxxSchemaValidate:      DXXQ033E  Invalid identifier starting at parse err1: line:0
char:0 An
dxxSchemaValidate:       Exit, return rc=0
dxxSchemaValidateFileExternal:Exit, returns 0.
dxxTerminateParser:   Exit
```

Figure 8-13 XML Extender trace file

9

Performance-related information regarding XML Extender

In this chapter, we briefly look at some performance metrics at various levels of the operating system. Specifically we discuss performance PTFs and test results.

9.1 Operating system levels and enhancements

In an effort to help improve the performance of the XML Extender composition and decomposition, IBM has released several enhancements. Some of these apply to V5R3, some to V5R4, and some to both. Make sure to match the level of your system to the appropriate PTF.

9.1.1 XML parser performance increase: V5R3 and V5R4 PTF

The XML Extender product (5722DE1) uses parts of the XML Toolkit product (5733XT1) to do its processing. Because of this, performance enhancements in the XML Toolkit will also enhance the XML Extender product. The XML Toolkit product is currently at R110 and can be installed on i5/OS (5722SS1) V5R3 or V5R4. PTF SI21062 increases the performance of 5733XT1 and should be installed on both i5/OS V5R3 and V5R4.

9.1.2 XML Extender performance increase: V5R3 PTF

i5/OS has implemented several changes to increase the performance of teraspace memory. These changes can improve XML Extender performance when working with large documents. These enhancements are available for i5/OS V5R3 via PTF SI21015. The enhancements are built into the base of i5/OS V5R4, so no PTF is required.

9.1.3 XML Extender Quick Pool memory management: V5R4 PTF

With i5/OS V5R4, the new quick pool memory feature allows for better memory management. This feature increases performance of the XML Extender product. To implement this new feature, you must install PTF SI21069. With this PTF, the environment variable, named DB2DXX_QUICKPOOL_MALLOC, is used to activate this feature. While the value of the variable is not important, it is important that it exists.

To add this environment variable at the system level, enter:

```
ADDENVVAR ENVVAR(DB2DXX_QUICKPOOL_MALLOC) LEVEL(*SYS)
```

9.1.4 Technote

For details about these and possible future performance enhancements for the XML Extender product, see Technote 1231798, “Improving composition and decomposition performance for DB2 XML Extender on iSeries” on the Web at:

http://www-1.ibm.com/support/docview.wss?uid=swg21231798

9.1.5 PTF quick reference

Table 9-1 provides a quick reference of recommended PTFs.

Table 9-1 PTF quick reference

Enhancement	i5/OS V5R3	i5/OS V5R4
XML Toolkit (XML Parser; 5733XT1)	SI21062	SI21062
Teraspace memory (5722SS1)	SI21015	In base
Quick pool memory (5722DE1)	N/A	SI21069 and environment variable
Current cumulative, HIPERs, and database groups	SF99530	SF99540

9.2 XML Extender performance measurements

In an effort to show how these PTFs can help improve the performance of the XML Extender product, we ran several tests at various code levels on similar hardware. These examples are not meant to be a statement of the level of performance that you will see, but are provided for illustrative purposes only.

Important: The performance information in this redbook is for guidance only. System performance depends on many factors, including system hardware, system and user software, and user application characteristics. Customer applications must be carefully evaluated before estimating performance. IBM does not warrant or represent that a user can or will achieve a similar performance. No warranty on system performance or price/ performance is expressed or implied in this document

For our tests, we used two similar model 520 systems, one with V5R3 and the other with V5R4. Each had the latest cumulative, HIPER, and database PTFs loaded. Each system had two CPUs. We created a subsystem just for testing and allocated a 12 GB private memory pool with a maximum activity level of two jobs.

In addition to showing the differences between the two releases of the operating system, we wanted to investigate the performance gains provided by the fixes listed in 9.1, "Operating system levels and enhancements" on page 148. Our base systems had the latest cumulative, HIPER, and database PTFs loaded. When we refer to V5R3 with PTFs, we are applying PTFs SI21062 and SI21015 in addition to the base setup. When we refer to V5R4 with PTFs, we are applying PTF SI21062 and adding the environmental variable DB2DXX_QUICKPOOL_MALLOC on top of the base setup.

The first test case was the decomposition of a 78 MB simple XML document. While this document was large in size, the structure was relatively simple and shredded into a single table.

The second test case involved a set of smaller 10 MB files. While these files are smaller in size, their structure is much more complex and they shredded into 13 different tables. To look at running in serial versus parallel, we tested running two different 10 MB documents one after the other and then running them both at the same time. For the parallel runs, the job queue was held, the jobs submitted, and the job queue released.

For each operation, the private memory pool was cleared. The system activity was limited to just these tests while they were run. We measured the time from the call to the XML Extender program until the end of job message. The time stamps from the job logs were subtracted to find the total runtime for that operation. For the tests in parallel, the earlier of the two start time stamps was compared with the later of the two finish time stamps.

Values recorded are in seconds and are rounded to the nearest second. When the total was calculated for the total run time of the two 10 MB parts in serial, the raw values were used and that total was then rounded. Because of this, the sum of the rounded values of the individual parts may not equal the rounded value of the sum of the raw times. Each test was performed three times and the total durations averaged to obtain the values in Table 9-2 and Figure 9-1.

Table 9-2 Results from XML Extender test runs (in seconds)

Test/system	78 MB simple	10 MB part 1	10 MB part 2	Both 10 MB serial total	Both 10 MB parallel total
V5R3	1,229	852	962	1,813	1,158
V5R3 with PTFs	427	657	658	1,315	1,048
V5R4	877	696	409	1,105	692
V5R4 with PTFs	449	260	316	577	356

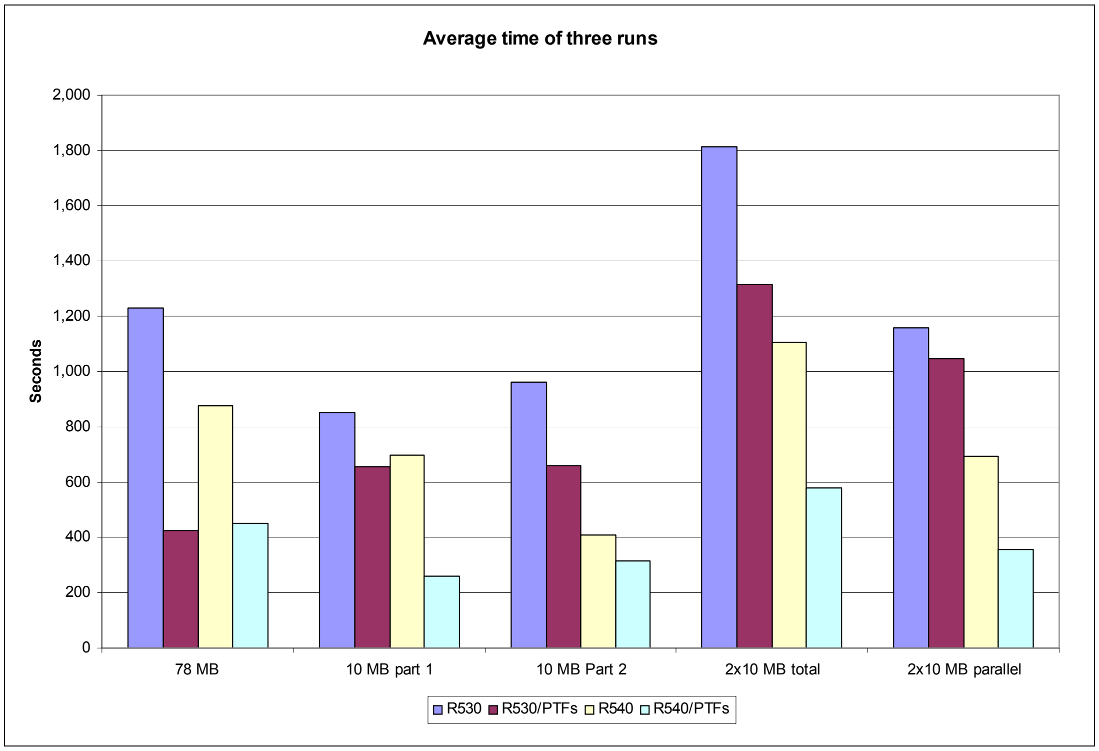


Figure 9-1 Graphical results from XML Extender test runs

From the chart in Figure 9-1, we can see that V5R3 without PTFs is always the slowest. Because of this, we will use this value as a baseline.

Looking at the simple 78 MB XML document, we can see that V5R4 ran in 71% of the time of V5R3. With PTFs at both V5R3 and V5R4, the run time was cut to 35% and 37%, respectively. Between V5R3 with PTFs and V5R4 with PTFs, the systems ran virtually the same.

In the second case, we have several points of investigation. First, we want to look at the total serial run time for the two complex XML documents. This involved running two different 10 MB XML documents back to back, measuring the time for each, and combining the results. The results show that the total serial run time of V5R4 with PTFs was 32% of the time of V5R3 without PTFs. V5R3 with PTFs and V5R4 without PTFs ran basically the same at 73% and 61% respectively.

In the second half of the test, we ran the two jobs in parallel to show the scalability of using multiple processors. We can start by looking at times compared to the same test. Again, we can see that V5R4 with PTFs ran in 31% of the time of V5R3 without PTFs. V5R3 with PTFs ran in 91% of the time of V5R3 without PTFs, while V5R4 without PTFs ran in only 60% the time of V5R3.

The second interesting comparison is to look at the times for the same work in serial versus parallel. In each case, parallel was faster than serial. Looking at V5R3 without PTFs, V5R4 without PTFs, and V5R4 with PTFs between serial and parallel, the run times in parallel were 64%, 63%, and 62% of the serial run time respectively. V5R3 with PTFs was longer with parallel taking 80% of the serial run time, but still providing improvement over running in serial. Considering that true linear scaling results in a 50% run time, seeing only a 12% time increase for running in parallel shows how closely the System i can get to linear scalability.

Overall, V5R4 with PTFs ran three times faster than V5R3 without PTFs and about twice as fast as V5R3 with PTFs or V5R4 without PTFs.

9.3 XSLT and XML Extender compared to XSLT and SQL

In the past, some clients have circumvented the performance problems of XML Extender by using XSLT. Instead of using XML Extender to do insertion of the rows, XSLT was used to transform the XML document into an SQL script. This script is a series of inserts to do the same thing. The advantage of doing this in the past was because the XSLT and the SQL script took about a third of the time of using the XML Extender product. We wanted to test to see if this was still valid and required with the PTFs from 9.1, “Operating system levels and enhancements” on page 148, or if the PTFs increased the performance of the XML Extender product.

For our tests, we used one model 520 system with V5R4. The latest cumulative, HIPER, and database PTFs were loaded. The system had two CPUs. We created a subsystem just for testing and allocated a 12 GB private memory pool with a maximum activity level of two jobs.

In the XML Extender test, first an XSLT was used to add elements that allow us to join the layers together. These artificial keys allow us to join between the multiple tables. This produces a transformed XML document. From there, the transformed XML document is used as input to the XML Extender product to do the decomposition.

The second test accomplished the same tasks but using two different steps. First, the XML document is used with an XSLT. In one pass, the XSLT adds the artificial keys and instead of writing to an XML document, the output is an SQL script. This script contains SQL insert statements to populate the tables. This SQL script was used as input with an SQL interface to populate the tables. In the end, we do the same work via two different methods.

For each operation, the private memory pool was cleared. The system activity was limited to just these tests while they were run. We measured the time from the call to the CL program until the end of job message. The time stamps from the job logs were subtracted to find the total run time for that operation. Values recorded were in seconds and were rounded to the nearest second. Each test was performed three times, and the total durations were averaged to obtain the values shown in Table 9-3 and Figure 9-2.

Table 9-3 Test results comparing XML Extender and XSLT (in seconds)

Test	Run 1	Run 2	Run 3	Average
XSLT + XML Extender	471	580	563	538
XSLT + SQL	520	523	518	520

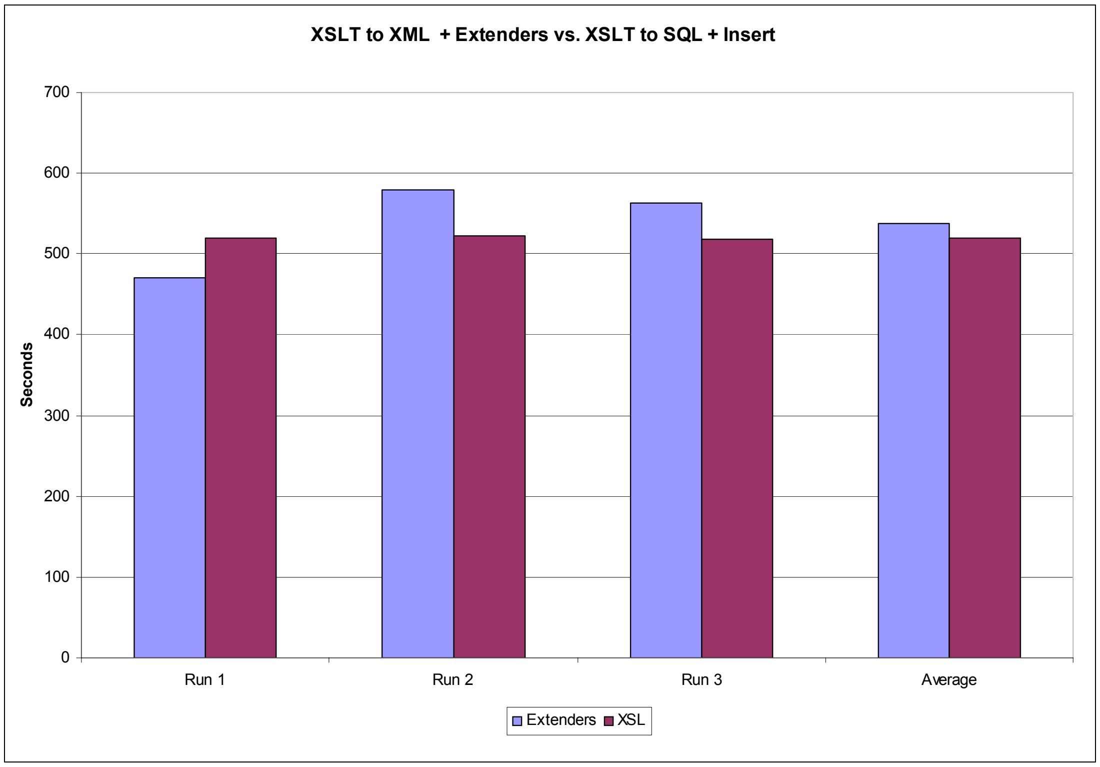


Figure 9-2 Graphics results from the XML Extender and XSLT test runs

Looking at the data, we can see that, with the PTFs, V5R4 is equivalent to the XSLT and SQL method. This means that performance is no longer a consideration when deciding if XML Extender or other alternatives, such as XSLT and SQL, should be used. We can then look directly at the advantages and disadvantages of each approach.

First we must look at how quickly this can be setup for use. In the XML Extender product, we need to create the DAD and the XSL to add in the keys. In the SQL method, we need to create a more complex XSL that adds in keys as well as creating the SQL script. At this point, we are about equal.

Next, we must look at functionality. With the XSLT and SQL, we can only decompose XML documents. There is no way to use a physical file as input to an XSL. XSL can only ready XML documents. However, an RDB node mapping DAD, which is specifically the exact same DAD, can be used for both composition and decomposition.

Which option should you use? While we cannot answer that, the best we can do is provide information about the choices and allow you to make an informed decision. For more details about the benefits of these methods, see Chapter 7, “Advantages and disadvantages of the programmatic approach" on page 105, and Chapter 12, "Advantages and disadvantages of the middleware approach" on page 197.

9.4 Performance summary

There are several key factors to achieve the highest level of performance. First, it is important to stay current on the operating system and database PTFs. Second, PTFs specifically for the XML Toolkit (including the XML Parser) and XML Extender should be kept current. By using a system with multiple processors, the scalability of multiple tasks running at once can dramatically decrease the overall processing time to almost linear scaling. And third, as each release of the operating system is made available, usually performance enhancements are built in. While this is not mandatory, sometimes the performance benefits alone are enough to warrant evaluation of the release even if no new functionality is used.

If you have tried XML Extender in the past and decided not to use it because of performance concerns, we recommend that you take another look. As shown in these tests, we achieved up to three times the performance of the base V5R3 release with V5R4 and all performance PTFs loaded.

If you want to use XML Extender but have worked around the performance issues by using another methodology, we suggest that you try it again. There have been significant performance increases that make it run as well as other programmatic approaches. The additional benefits of the XML Extender product with its increased performance make it a wise choice.

10

Shredding methodology

In Chapter 8, “Overview of DB2 XML Extender” on page 109, we explain the IBM provided middleware product that facilitates composing of XML documents from DB2 data and shredding XML documents into DB2 relational tables. In this chapter, we demonstrate how to take advantage of the DB2 XML Extender and WebSphere Development Studio Client tooling to dramatically speed up the process of implementing step 4 (Figure 10-1) in our scenario. For details about this scenario, see Chapter 2, “Scenario overview” on page 21.

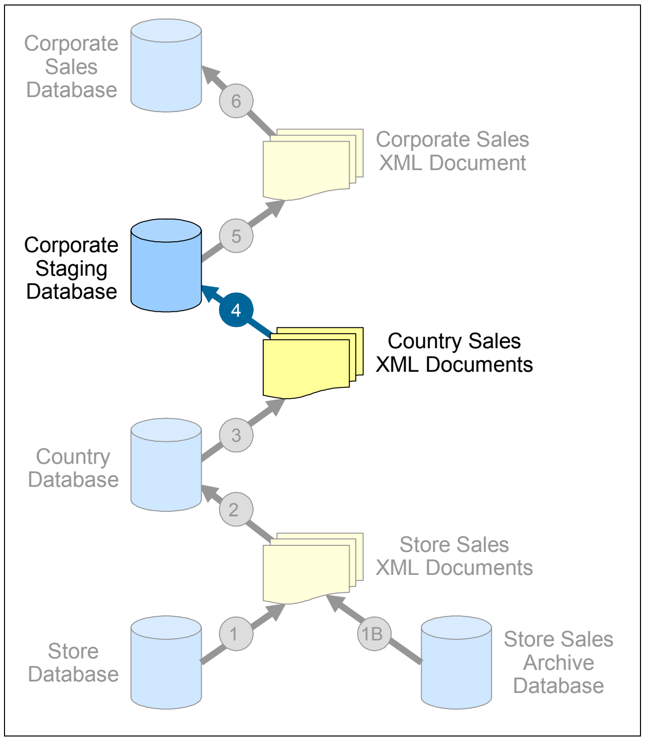

Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 10-1 Scenario step 4

The purpose of shredding is to store untagged elements and attributes in DB2 tables. The presented shredding methodology has been implemented successfully in several large projects.

The proposed methodology can be divided into two major phases:

- *Design and implementation*: In this phase, we analyze the structure of the inbound XML document, define the XML to relational database (RDB) mapping, perform any necessary transformations, build the target tables, and create the document access definition (DAD) file. The entire process is accomplished in WebSphere Development Studio Client.
- *Deployment and validation*: In this phase, we deploy the solution developed in WebSphere Development Studio Client to the target System i machine and run it to shred inbound XML documents.

Note: Although we use WebSphere Development Studio Client to facilitate rapid application development in this chapter, you can use any other Integrated Development Environment (IDE) tool that supports XML and Extensible Stylesheet Language (XSL) Transformation to implement the methodology. An obvious advantage of using WebSphere Development Studio Client is that it provides an XML to RDB mapper wizard that generates the required DAD automatically.

10.1 Setting up the development environment

First we create a new project in WebSphere Development Studio Client. Since we intend to use various technologies, such as XSL Transformations, Java, and XML, we create a new Dynamic Web Project DB2XMLRedbook_ScenarioStep4. This project type allows us to quickly navigate between different parts of the application.

In WebSphere Development Studio Client, we switch to the Resource perspective, which is used most often in the development process. We do not plan to build or deploy a Java 2 Platform, Enterprise Edition (J2EE™) enterprise archive (EAR) file. Therefore, at this point, we remove the DB2XMLRedbook_ScenarioStep4EAR folder that was generated automatically by WebSphere Development Studio Client as part of the Dynamic Web Project. Then we create a new folder called CorpStageXML in the /DB2XMLRedbook_ScenarioStep4/WebContent/WEB-INF folder. The CorpStageXML folder is used to store XML-related files. Figure 10-2 shows the structure of the newly created project.

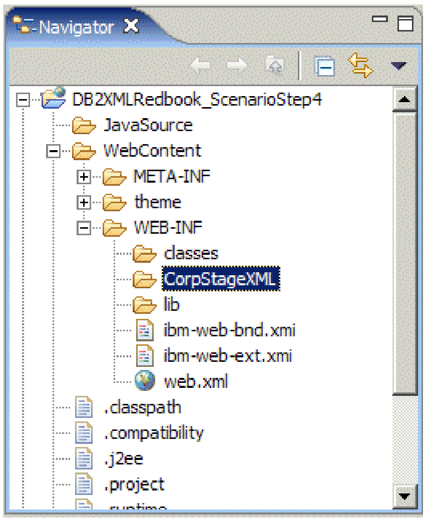
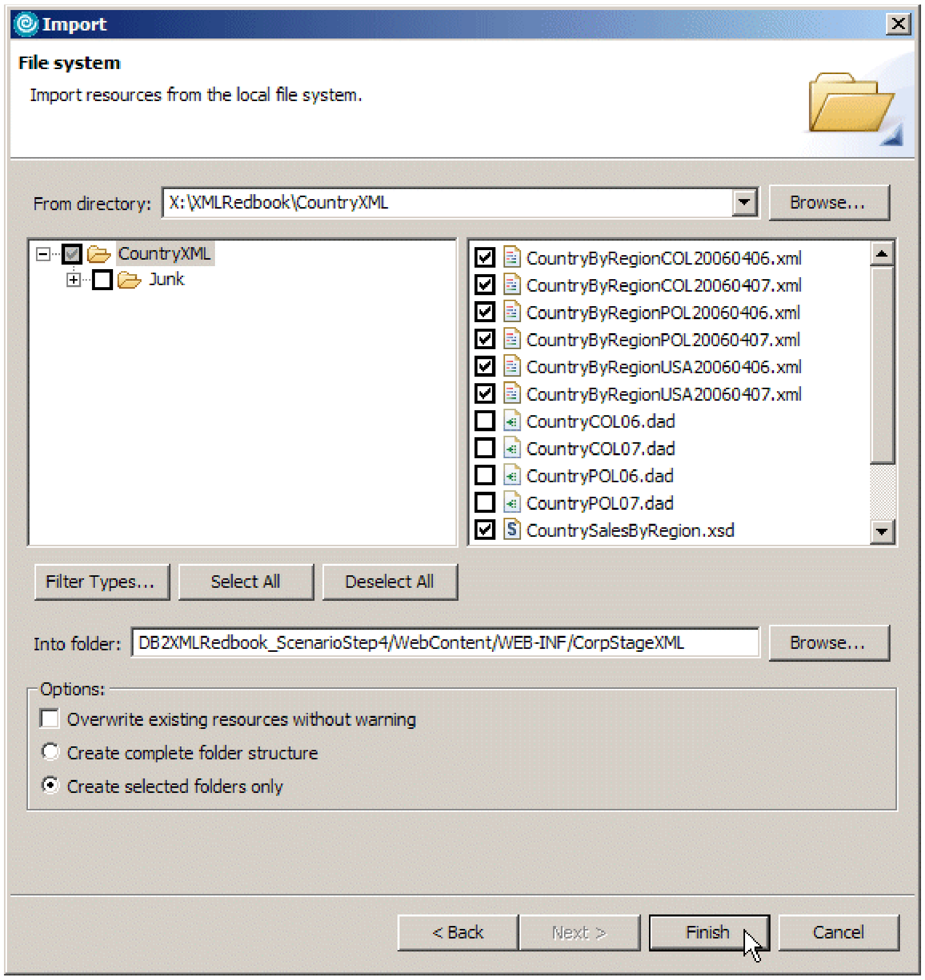


Figure 10-2 WebSphere Development Studio Client project structure

We start the development process by importing the inbound XML documents along with the corresponding schema definition for those documents. The inbound documents were created in step 3 of the scenario. The files reside in the target System i machine in the /XMLRedbook/CountryXML directory. The integrated file system root is mapped as a network drive on the development workstation.

1. In the Resource perspective's Navigator panel of WebSphere Development Studio Client, right-click the **CorpStageXML** folder and select **Import**.
2. In the Import window, select **File System** and click **Next**.
3. In the File System window, next to the From Directory text box, click **Browse**. Navigate to the /XMLRedbook/CountryXML integrated file system directory. Select the necessary inbound files as shown in Figure 10-3. Then click **Finish**.



From directory: X:\XMLRedbook\CountryXML

[■] CountryXML
[ ] Junk

[✓] CountryByRegionCOL20060406.xml
[✓] CountryByRegionCOL20060407.xml
[✓] CountryByRegionPOL20060406.xml
[✓] CountryByRegionPOL20060407.xml
[✓] CountryByRegionUSA20060406.xml
[✓] CountryByRegionUSA20060407.xml
[ ] CountryCOL06.dad
[ ] CountryCOL07.dad
[ ] CountryPOL06.dad
[ ] CountryPOL07.dad
[✓] CountrySalesByRegion.xsd

Into folder: DB2XMLRedbook_ScenarioStep4/WebContent/WEB-INF/CorpStageXML

Options:
[ ] Overwrite existing resources without warning
( ) Create complete folder structure
(●) Create selected folders only

Figure 10-3 Importing Country Sales by Region XML documents

10.2 Analyzing the XML document structure and determining the mapping

The XML documents imported in the previous section comply with the schema definition specified in the CountrySalesByRegion.xsd file. We now analyze the hierarchy of a sample inbound document to determine the relationship among the elements and attributes.

Specifically, we focus on such concepts as element containment, repeating, and self-named elements. Refer to 1.3, “XML to relational database mapping” on page 15, for more details.

A document type definition (DTD) associated with the inbound documents can be used to illustrate the XML hierarchy. We use DTD rather than XML Schema Definition (XSD) because it contains all the necessary hierarchy information. It is also typically more concise than XSD and, therefore, is easier to analyze. In our case, a DTD does not exist so we use WebSphere Development Studio Client to generate it.

To generate a DTD, right-click the imported **CountrySalesByRegion.xsd** document and select **Generate** → **DTD**. Make sure that WebSphere Development Studio Client is switched to the Resources perspective.

The generated DTD lists the elements and attributes alphabetically. Therefore, it is helpful to rearrange them so that the DTD reflects the hierarchy of the XML documents, which is shown in Example 10-1.

Example 10-1 CountrySalesByRegion.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT CountrySalesByRegion (CountryInfo,Regions) >
<!ATTLIST CountrySalesByRegion
  Date CDATA #IMPLIED
>
<!ELEMENT CountryInfo (Name) >
<!ELEMENT Regions (Region+) > 1
<!ELEMENT Region (Name,Brand+) >2
<!ELEMENT Brand (Name,Sales?,Returns?)3 >
<!ELEMENT Sales (Currency,Amount) >
<!ELEMENT Returns (Currency,Amount) >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Currency (#PCDATA)>
<!ELEMENT Amount (#PCDATA)>
```

By carefully analyzing the source in Example 10-1, we learn several important things about the XML hierarchy described by the DTD grammar:

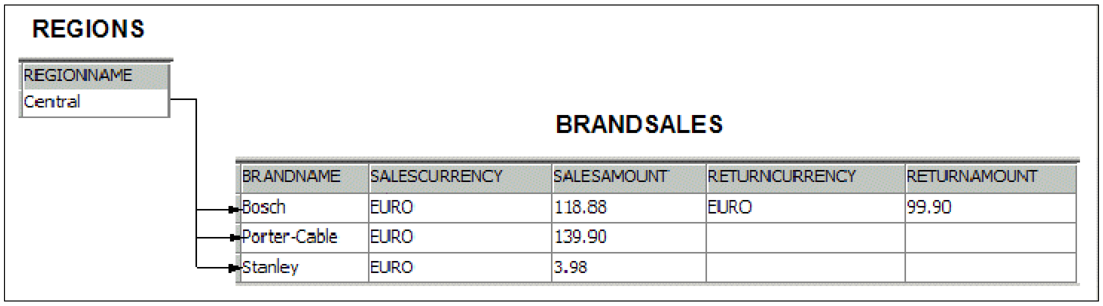
- **Repeating elements**: Some elements may repeat within one parent element. In line 1, a Regions element can contain one or many occurrences of the Region element. Notice the plus sign (+) next to Region. We say that there is a *one-to-many relationship* between Regions and Region. Similarly, there is a one-to-many relationship between Region and Brand as shown in line 2.
- **Optional elements**: Some elements are optional. In line 3, a Returns element may occur zero times or one time within a Brand element. Optional elements are mapped to nullable columns in the database tables. Notice the question mark (?) next to Returns. Similarly, the Sales element is optional within Brand.
- **Wrapper elements**: These elements have no child attributes or text nodes. However, they contain children elements that have attributes or text nodes. In this case, CountryInfo and Regions are wrapper elements. They are used to logically separate country data from regions sales data.

Let us assume that the target tables do not exist and that they must be designed as part of the mapping process. During the mapping, we decide which tables and columns are needed to accommodate the data found in the XML document. Each text node or attribute of an element is mapped into a column in a table.

An element and all its children that cannot repeat *can be mapped* into columns of the same table. This can occur because different columns in one row of a table correctly represent the one-to-one relationship between parent and child for those elements that cannot repeat. For example, each occurrence of a CountryInfo element can have one occurrence of a Name element. Therefore, all data from and CountryInfo can be stored in one row.

Alternatively, a child element that can repeat must be mapped to a separate table. In this case, there is a one-to-many relationship between a parent and its children. The only way to reflect this relationship in a relational database is to store the parent element in a parent table and to store the children elements in multiple rows of a dependent table.

For example, a Region element contains multiple Brand elements. Therefore, the data for a particular occurrence of Region can be stored in one row of a Regions table, while children of this Region element must be stored in multiple rows of the BrandSales table, which is illustrated in Figure 10-4.



REGIONS

REGIONNAME
Central

BRANDSALES

BRANDNAME	SALESCURRENCY	SALESAMOUNT	RETURNCURRENCY	RETURNAMOUNT
Bosch	EURO	118.88	EURO	99.90
Porter-Cable	EURO	139.90		
Stanley	EURO	3.98		

Figure 10-4 One-to-many relationship mapping

Now that we know how to map one-to-one and one-to-many relationships among elements into relational tables, we can revisit the CountrySalesByRegion.dtd to provide the XML to relational database mapping for all the elements and attributes of the inbound document.

The analysis starts at the root element, which is CountrySalesByRegion in this case. It has an attribute, Date, and a child, CountryInfo. An attribute can occur only once. Similarly, according to the DTD, the CountryInfo element can occur only once. CountryInfo is a wrapper that contains an element called Name. Name occurs in CountryInfo only once.

We can map data from all three elements analyzed so far to a single table. For simplicity, we call this table Countries. We need two columns in the Countries table:

- CountryName, which contains data for the /CountrySalesByRegion/CountryInfo/Name element
- SalesDate, which contains data for the /CountrySalesByRegion/@Date attribute

As mentioned, since CountryInfo is a wrapper, it contains no data.

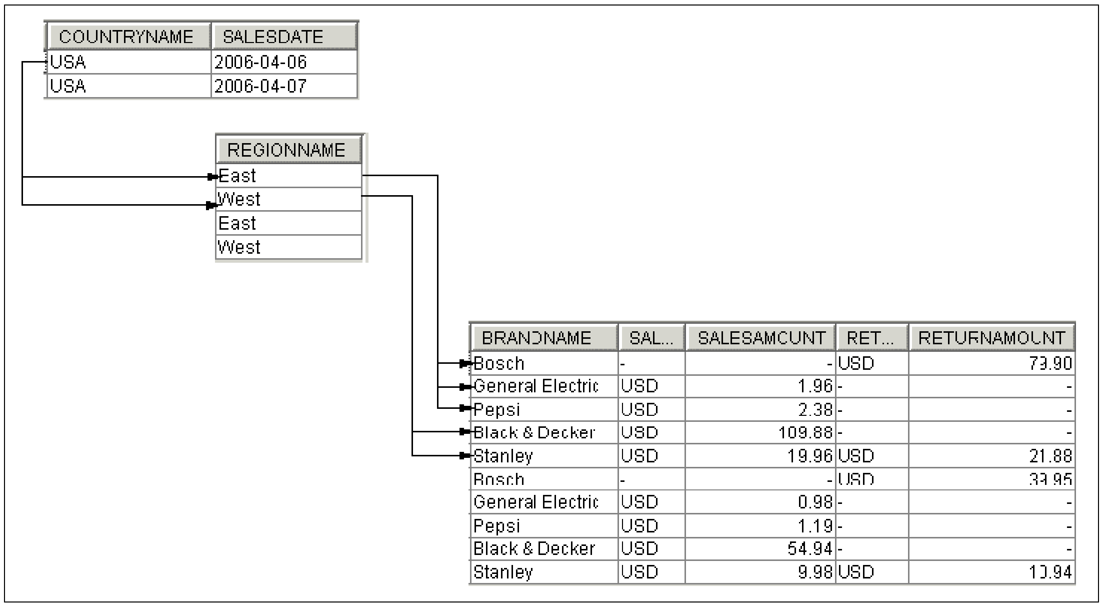
The next element to tackle is Regions. It is a wrapper that contains Region, which, as indicated by the + sign next to it, can occur one or more times. This tells us that there can be multiple regions within a given country. Region needs to be mapped to a separate table. Region contains two elements, Name and Brand. Name can occur once. Brand, however, is a multi-occurrence element so it cannot be mapped to the same table as Region. This leaves us with just one column in the Regions table, RegionName, which contains data for /CountrySalesByRegion/Regions/Region/Name element.

Brand contains three elements, Name, Sales, and Returns. None of these elements can occur more than once so they can be mapped into a single table, BrandSales. Note that Sales, and Returns are marked as optional as indicated by the ? sign next to the elements name in the DTD. Therefore the corresponding columns should be marked as nullable. We document the mapping process in Table 10-1.

Table 10-1 XML hierarchy mapped to relational tables

Location path	Table name	Column name	Nullable
/CountrySalesByRegion/@Date	COUNTRIES	SALESDATE	N
/CountrySalesByRegion/CountryInfo/Name	COUNTRIES	COUNTRYNAME	N
/CountrySalesByRegion/Regions/Region/Name	REGIONS	REGIONNAME	N
/CountrySalesByRegion/Regions/Region/Brand/Name	BRANDSALES	BRANDNAME	N
/CountrySalesByRegion/Regions/Region/Brand/Sales/ Currency	BRANDSALES	SALESCURRENCY	Y
/CountrySalesByRegion/Regions/Region/Brand/Sales/Amount	BRANDSALES	SALESAMOUNT	Y
/CountrySalesByRegion/Regions/Region/Brand/Returns/ Currency	BRANDSALES	RETURNCURRENCY	Y
/CountrySalesByRegion/Regions/Region/Brand/Returns/ Amount	BRANDSALES	RETURNAMOUNT	Y

Figure 10-5 shows the layout of the target tables and their relationships. Sample data is provided to better illustrate the one-to-one and one-to-many relationships between rows.



COUNTRYNAME	SALESDATE
USA	2006-04-06
USA	2006-04-07

REGIONNAME
East
West
East
West

BRANDNAME	SAL...	SALESAMOUNT	RET...	RETURNAMOUNT
Bosch	-	-	USD	73.90
General Electric	USD	1.96	-	-
Pepsi	USD	2.38	-	-
Black & Decker	USD	109.88	-	-
Stanley	USD	19.96	USD	21.88
Bosch	-	-	USD	33.95
General Electric	USD	0.98	-	-
Pepsi	USD	1.19	-	-
Black & Decker	USD	54.94	-	-
Stanley	USD	9.98	USD	10.94

Figure 10-5 The layout of the target database

In Figure 10-5, the arrows indicate the hierarchy of the elements in the XML document mapped into a set of tables. In this case, there is one row per country and sales date in the Country table. One row in the Country table can, however, have multiple dependent rows in the Regions table. Similarly, one row in the Regions table can have multiple dependent rows in the BrandSales table. Recall that columns are mapped to XML nodes. The arrows in Figure 10-5 are necessary to correctly reflect the containment and multi-occurrence of the nodes.

Our discussion has reached the most critical aspect of hierarchical-to-relational database mapping: How do we preserve the relationships represented by the arrows in Figure 10-5 in a relational database model? Referential integrity (RI) is probably the most natural choice, and, in fact, it is the cornerstone of the proposed methodology. To tie the related rows with the unique key and foreign key constraints, we need to insert unique key values into the designated elements. The key value uniquely identifies the given instance of an element. At shredding time, the extender will propagate the key value as a foreign key to the children elements that can repeat. That way, the children will be tied back to the parent element.

Tip: Another approach is to construct a unique key from the data that already exists in the XML document. For example, a composite key “CountryName, RegionName, BrandName” seems to be a good candidate. However, inserting a generated unique key is a more generic solution and works with any XML document. Our experience shows that, in many cases, there are no good candidates for a unique key, or the XML hierarchy is so complex that constructing a composite key by adding a new key segment on each new level of hierarchy becomes difficult if not unmanageable.

The process of selecting the designated elements is pretty straightforward. We must add a unique key to the *parents* of the elements that can repeat, which are CountrySalesByRegion and Region.

Important: When adding unique keys, it is critical to differentiate between the ancestor and preceding nodes. The unique key must be added to the parent node. Adding a key to the preceding node does not work because the key will not be propagated to the dependent element at the time the document is shredded. For example, we add the unique key to the CountrySalesByRegion element that is the ancestor of Region. Conversely, adding the key to CountryInfo will not work since this is the preceding element.

10.3 XSL Transform to insert unique keys for repeating elements

The mapping process determines which elements need to contain the unique keys. We decide to use the XSL Transformation to insert the necessary values into the inbound XML document. We use the javax.xml.transform.Transformer class to apply the simple XSL stylesheet shown in Example 10-2.

Example 10-2 CountrySalesByRegion.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
xmlns:xalan="http://xml.apache.org/xslt">
<xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
<xsl:param name="guuid" /> 1
   <xsl:template match="*">
      <xsl:copy>
         <xsl:copy-of select="@*" />
         <xsl:apply-templates />
      </xsl:copy>
   </xsl:template>
   <xsl:template match="CountrySalesByRegion|Region"> 2
      <xsl:copy>
         <xsl:attribute name="Uid">
            <xsl:value-of select="concat($guuid,generate-id(.))" /> 3
            </xsl:attribute>
            <xsl:copy-of select="@*" />
            <xsl:apply-templates />
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

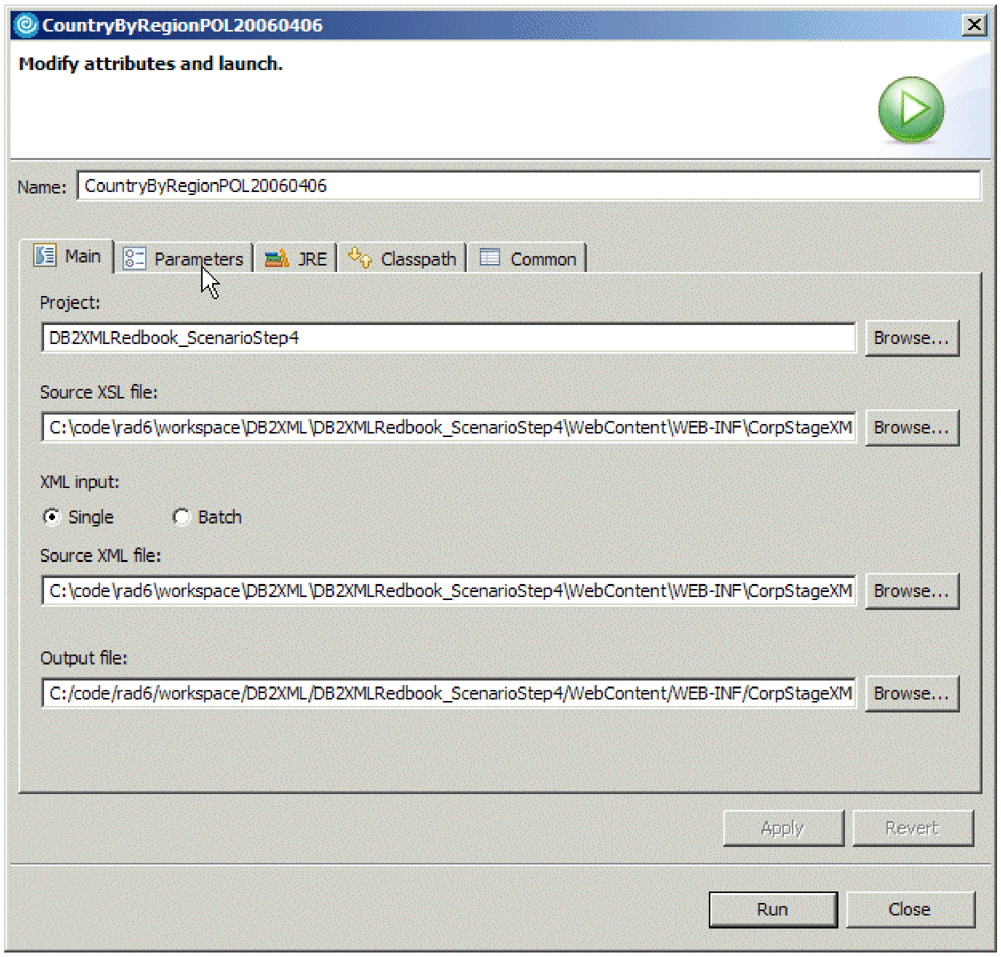
At line 1 in Example 10-2, an input parameter is defined. For simplicity, we use the current time stamp as the input parameter. This guarantees the key uniqueness on consecutive invocations of the transformer. At line 2, the elements that need to contain a unique key are selected. At line 3, a unique key is generated. The input parameter `guuid` is concatenated with the return value of the XSL generate-id() function to produce a unique value. The newly generated unique key value is inserted as an attribute called `Uid`. All other nodes in the inbound document are copied to the transformed XML document.

To create the stylesheet in our project:

1. Right-click the **CorpStageXML** folder and select **New** → **Other...** → **XML** → **XML Schema**.
2. Name the stylesheet `CountrySalesByRegion.xsl`.
3. Copy the contents of Example 10-2 into the newly created document.

You can test the stylesheet in WebSphere Development Studio Client with no additional coding:

1. Select one of the inbound documents imported into the CorpStageXML folder, for example **CountryByRegionPOL20060406.xml**. (See 10.1, “Setting up the development environment” on page 156). Right-click and select **Run** → **XSL Transformation**.
2. In the Modify Attributes and Launch window (Figure 10-6), enter the name of the XSL stylesheet. Change the name of the output file to `CountryByRegionPOL20060406_t.xml`.


CountryByRegionPOL20060406

Modify attributes and launch.

Name: CountryByRegionPOL20060406

Project:
DB2XMLRedbook_ScenarioStep4

Source XSL file:
C:\code\rad6\workspace\DB2XML\DB2XMLRedbook_ScenarioStep4\WebContent\WEB-INF\CorpStageXM

XML input:
[✓ Single] Batch

Source XML file:
C:\code\rad6\workspace\DB2XML\DB2XMLRedbook_ScenarioStep4\WebContent\WEB-INF\CorpStageXM

Output file:
C:/code/rad6/workspace/DB2XML/DB2XMLRedbook_ScenarioStep4/WebContent/WEB-INF/CorpStageXM

Figure 10-6 Modify attributes and launch window

3. Select the **Parameters** tab.
4. On the Parameters page, add an input parameter for the XSL stylesheet. The name of the parameter should be `guuid`. Type the current time stamp as the value, for example `2006-04-18-13:55:00`.
5. Click **Run** to perform the transformation.

Example 10-3 shows an excerpt from the transformed document.

Example 10-3 Excerpt from CountryByRegionPOL20060406_t.xml

```
<CountrySalesByRegion Uid="2006-04-18-13:55:00N10001" Date="2006-04-06">
  <CountryInfo>
    <Name>Poland</Name>
  </CountryInfo>
  <Regions>
    <Region Uid="2006-04-18-13:55:00N1000F">
...
```

Note the presence of the inserted attributes for the CountrySalesByRegion and Region elements.

To perform the necessary transformation in the runtime environment, we provide a simple Java program that invokes the javax.xml.transform.Transformer class. Example 10-4 shows the Java source code.

Example 10-4 Transform.java source code

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class Transform {
   public static void main(String argv[]) throws TransformerException,
         TransformerConfigurationException, FileNotFoundException,
         IOException {
      if ( argv.length != 4) {
         System.err.
         println("Usage: java Transform sourcexml stylesheet targetxml guuid");
         System.exit(1);
      }
      TransformerFactory tFactory = TransformerFactory.newInstance(); 1

      Transformer transformer = tFactory.newTransformer(new StreamSource(
           argv[1])); 2
      // Set the global universal unique identified as input parameter guuid
      transformer.setParameter("guuid", argv[3]); 3
      // Use the Transformer to apply the associated Templates object to an
      // XML document and write the output to a file
      transformer.transform(new StreamSource(argv[0]), new StreamResult(
           new FileOutputStream(argv[2]))); 4
      System.out.println("************* The result is in " + argv[2]
           + " *************");
   }
}
```

The Transform class accepts the following parameters:

- **sourcexml**: The file name of the XML document to be transformed
- **stylesheet**: The file name of the XSL stylesheet that defines the required transformation
- **targetxml**: The file name of the transformed XML document
- **guuid**: The parameter used by XSLT to seed the unique attribute values to be inserted into the target XML document

  Typically, a time stamp is used as the input value since it guarantees that the consecutive invocations of the transformer will produce unique target XML documents.

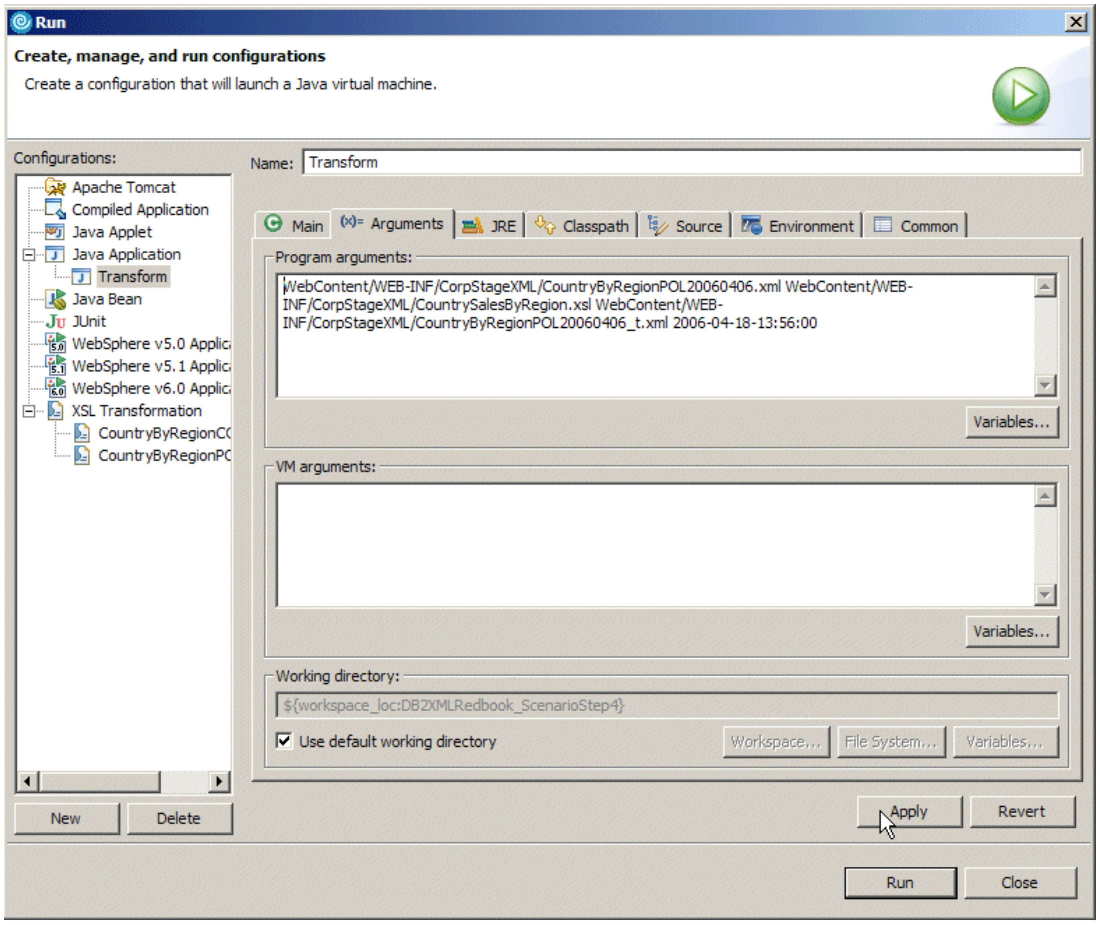
In Example 10-4, at line 1, a TransformerFactory class is instantiated and then is used at line 2 to create a Transformer class instance. At line 3, the setParameter method is called on the Transformer class. This parameter is passed by the Transformer into the XSL stylesheet. At line 4, the transform method is used to perform the transformation and write the resulting XML document into the file system.

To create the program in the project:

1. Switch to the Java Perspective.
2. Expand the **JavaSource** folder, right-click the (default package), and select **New** → **Class**.
3. Name the Java class Transform. Insert the source code listed in Example 10-4.

To test the program:

1. Right-click the newly created **Transform.java** in the (default package) and select **Run** → **Run**.
2. In the Create, manage, and run configurations window, select **Java Application** and then click **New**. Make sure that the class name is Transform. Select the **Arguments** tab and fill the Program arguments text box as shown in Figure 10-7.


Run

Create, manage, and run configurations

Create a configuration that will launch a Java virtual machine.

Name: Transform

Program arguments:

```
WebContent/WEB-INF/CorpStageXML/CountryByRegionPOL20060406.xml WebContent/WEB-
INF/CorpStageXML/CountrySalesByRegion.xsl WebContent/WEB-
INF/CorpStageXML/CountryByRegionPOL20060406_t.xml 2006-04-18-13:56:00
```

Working directory: ${workspace_loc:DB2XMLRedbook_ScenarioStep4}

☑ Use default working directory

Figure 10-7 Transform program arguments

3. Click **Apply** and then click **Run**. The transformed CountryByRegionPOL20060406_t.xml document is created in the CorpStageXML folder.

10.4 Modifying DTD and XSD to reflect the transformed document

The original DTD needs to be modified to reflect the structure of the transformed XML document. The necessary change entails adding a new attribute, called Uid, to the designated elements. The DTD for the transformed XML document is used in the mapping step Refer to 10.7, “Graphically mapping transformed DTD to relational tables (generating DAD)” on page 170.

We added the unique ID attribute to two elements in the original inbound XML document: CountrySalesByRegion and Region. To modify the DTD in WebSphere Development Studio Client:

1. Right-click **CountrySalesByRegion.dtd** and select **Copy**.
2. Right-click the **CorpStageXML** folder and select **Paste**.
3. In the Name Conflict window, change the target file name to `CountrySalesByRegion_t.dtd` and click **OK**.
4. Double-click the newly created DTD to edit it. The source code is displayed in the edit panel. The outline of the document is presented in the Outline panel. If you cannot locate the Outline panel in the IDE, select **Window** → **Show View** → **Outline**.
5. In the Outline panel, right-click the **CountrySalesByRegion** element. Select **Add attribute**. A new attribute is inserted.
6. In the Properties panel, change the name of the newly inserted attribute to Uid. The change is immediately reflected in the source code panel.
7. Using the same method, insert the Uid attribute to the Region element.

Example 10-5 shows the modified DTD source. Notice the inserted attributes at lines 1 and 2.

Example 10-5 CountrySalesByRegion_t.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT CountrySalesByRegion (CountryInfo,Regions) >
<!ATTLIST CountrySalesByRegion
  Date CDATA #IMPLIED
  Uid CDATA #IMPLIED 1
   >
<!ELEMENT CountryInfo (Name) >
<!ELEMENT Regions (Region+) >
<!ELEMENT Region (Name,Brand+) >
<!ATTLIST Region
  Uid CDATA #IMPLIED> 2
<!ELEMENT Brand (Name,Sales?,Returns?) >
<!ELEMENT Sales (Currency,Amount) >
<!ELEMENT Returns (Currency,Amount) >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Currency (#PCDATA)>
<!ELEMENT Amount (#PCDATA)>
```

A modified XSD is required as input to the Generate DDL wizard that is used in the next section. Additionally, the XSD may be required if you decide to validate the inbound documents before shredding them into the database. Again, you can use WebSphere Development Studio Client to insert the two new attributes into the original XSD or directly modify the source.

Example 10-6 shows an excerpt of the updated definition of the CountrySalesByRegion element. Notice the inserted attribute in the line highlighted in bold.

Example 10-6 Excerpt from CountrySalesbByRegion.xsd

```
<xsd:element name="CountrySalesByRegion">
     <xsd:complexType>
        <xsd:sequence>
           <xsd:element ref="CountryInfo" />

           <xsd:element ref="Regions" />
        </xsd:sequence>
        <xsd:attribute name="Date" type="xsd:date" use="required" />
        <xsd:attribute name="Uid" type="xsd:string" use="required" />

     </xsd:complexType>
  </xsd:element>
```

10.5 Building the DDL script and adding RI constraints for the keys

In 10.2, "Analyzing the XML document structure and determining the mapping" on page 157, we defined the layout of the target tables. Now we create a Data Definition Language (DDL) script that can be run on the database server to create the necessary tables. Again, WebSphere Development Studio Client provides a wizard that can facilitate the process. We use the Generate DDL wizard to produce a "draft" DDL script from the modified XML schema (CountrySalesByRegion_t.xsd).

In the Navigator window, right-click the modified XSD file and select **Generate** → **DDL**. Then in the Generate DDL window, click **Finish**.

As indicated, the wizard provides only a guess as to which tables are needed to accommodate data that is contained in an XML document that is described by a given XSD. Despite the wizard's ability, we still prefer to use this estimate rather than type the entire DDL script from scratch.

Typically, in the generated script, we must remove unnecessary and redundant table and column definitions. Then we add the unique key constraints to accommodate the inserted Uid attributes. Next, starting from the Countries and Regions tables, we recursively express the one-to-many relationship between rows in the two tables by specifying a foreign key in the dependent table. This is done by adding a new column to the dependent table with the same attributes as the unique key column in the parent table and defining it as a foreign key. This definition ties the two tables together and is illustrated in Figure 10-8.

In Figure 10-8, the CountryUid column in Regions is a foreign key that references the same-named column in the parent table. Notice that, in the foreign key definition, we specified the ON DELETE CASCADE rule. This rule allows us, if necessary, to remove all shredded data for a given XML document by deleting one row in the Countries table. DB2 automatically removes all other related rows.

Tip: Currently, XML Extender does not provide the functionality to update the data after the document is shredded. To perform the update, you can delete the data from the target tables and then shred again using the modified XML document. The ON DELETE CASCADE rule can be used to facilitate the delete.

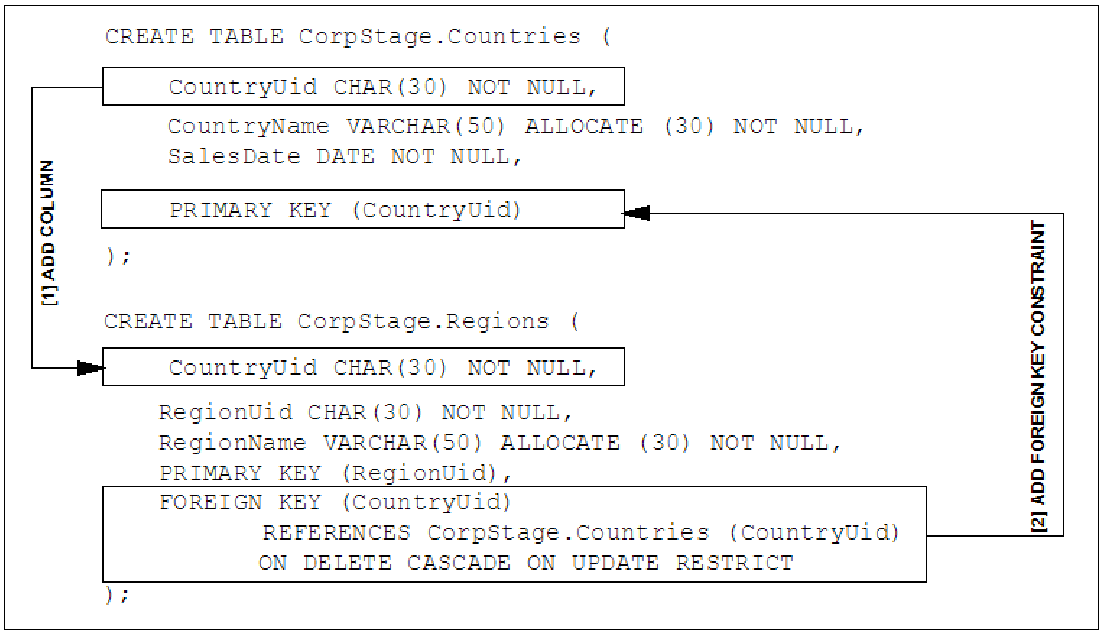

```
CREATE TABLE CorpStage.Countries (
    CountryUid CHAR(30) NOT NULL,
    CountryName VARCHAR(50) ALLOCATE (30) NOT NULL,
    SalesDate DATE NOT NULL,
    PRIMARY KEY (CountryUid)
);

CREATE TABLE CorpStage.Regions (
    CountryUid CHAR(30) NOT NULL,
   RegionUid CHAR(30) NOT NULL,
   RegionName VARCHAR(50) ALLOCATE (30) NOT NULL,
   PRIMARY KEY (RegionUid),
   FOREIGN KEY (CountryUid)
          REFERENCES CorpStage.Countries (CountryUid)
          ON DELETE CASCADE ON UPDATE RESTRICT
);
```

Figure 10-8 Building a referential integrity constraint

Example 10-7 shows the cleaned up DDL script.

Example 10-7 CreateCorpStage.sql

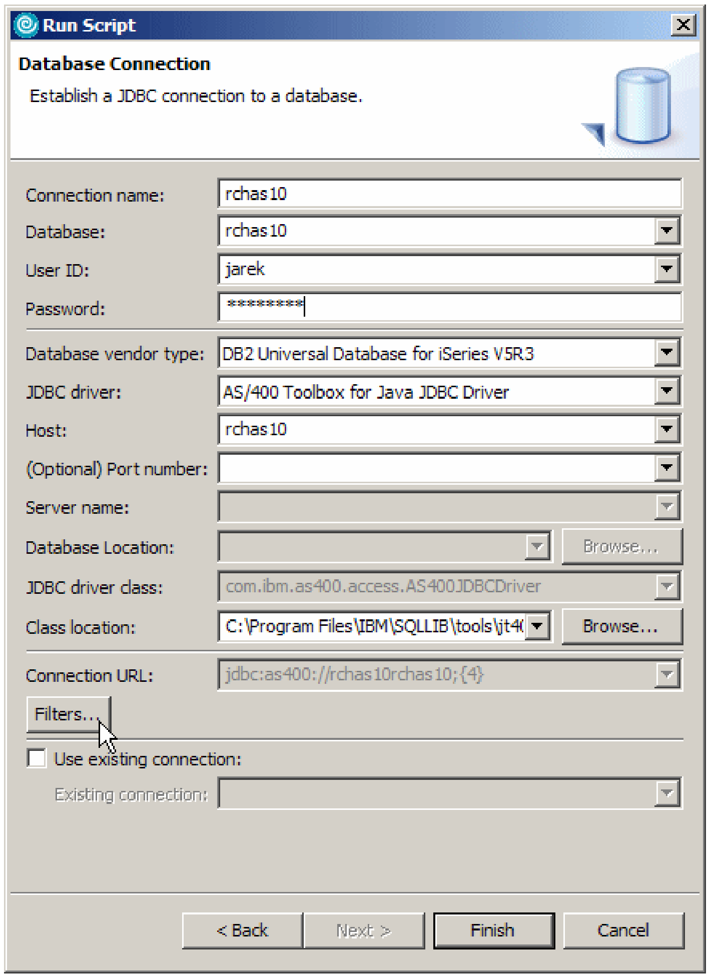
```
CREATE TABLE CorpStage.Countries (
   CountryUid CHAR(30) NOT NULL,
   CountryName VARCHAR(50) ALLOCATE (30) NOT NULL,
   SalesDate DATE NOT NULL,
   PRIMARY KEY (CountryUid)
);

CREATE TABLE CorpStage.Regions (
   CountryUid CHAR(30) NOT NULL,
   RegionUid CHAR(30) NOT NULL,
   RegionName VARCHAR(50) ALLOCATE (30) NOT NULL,
   PRIMARY KEY (RegionUid),
   FOREIGN KEY (CountryUid) REFERENCES CorpStage.Countries (CountryUid)
   ON DELETE CASCADE ON UPDATE RESTRICT
);

CREATE TABLE CorpStage.BrandSales (
   RegionUid CHAR(30) NOT NULL,
   BrandName VARCHAR (80) ALLOCATE (50) NOT NULL,
   SalesCurrency VARCHAR (30)ALLOCATE (10),
   SalesAmount DECIMAL(9,2),
   ReturnCurrency VARCHAR (30)ALLOCATE (10),
   ReturnAmount DECIMAL(9,2),
   FOREIGN KEY (RegionUid) REFERENCES CorpStage.Regions (RegionUid)
   ON DELETE CASCADE ON UPDATE RESTRICT);
```

Notice the use of the DB2 for i5/OS proprietary ALLOCATE keyword. The use of ALLOCATE improves the performance of the VARCHAR data type by eliminating the otherwise necessary additional I/O operation.

10.6 Creating tables on the database server

We create the target tables by deploying the CreateCorpStage.sql script to the target DB2 for i5/OS server as explained in the following steps:

1. In WebSphere Development Studio Client, switch to the Data perspective. Right-click the SQL script and select **Deploy**.
2. In the Run Script window, click **Next**.
3. In the next window, select **Automatically commit changes** and click **Next**.
4. Since this is the first time that an SQL script is deployed, specify the JDBC connection settings for a DB2 for i5/OS as shown in Figure 10-9.



Connection name:	rchas10
Database:	rchas10
User ID:	jarek
Password:	*********
Database vendor type:	DB2 Universal Database for iSeries V5R3
JDBC driver:	AS/400 Toolbox for Java JDBC Driver
Host:	rchas10
(Optional) Port number:	
Server name:	
Database Location:	
JDBC driver class:	com.ibm.as400.access.AS400JDBCDriver
Class location:	C:\Program Files\IBM\SQLLIB\tools\jt4(
Connection URL:	jdbc:as400://rchas10rchas10;{4}
☐ Use existing connection:	
Existing connection:	

Figure 10-9 Creating the JDBC connection

The System i machine can contain a large number of schemas. To limit the amount of metadata that is retrieved from the server, click the **Filter** button.

In the Filter window, add a filter condition, for example Schema Like CorpStage.

Click **Finish**.

The schema and the tables are created on the System i machine. Note that the new database connection appears in the Database Explorer panel.

10.7 Graphically mapping transformed DTD to relational tables (generating DAD)

WebSphere Development Studio Client provides a wizard that enables us to create RDB-to-XML mappings using graphical drag-and-drop editing. The wizard uses the target database metadata and the DTD as input. Then it produces the necessary DAD file that can then be used by XML Extender to shred the inbound documents. First we import the metadata for the target CorpStage schema into the current project:

1. Switch to the Data perspective. In the Navigator panel, right-click the **WEB-INF** folder. Select **New** → **Folder**.
2. In the New Folder window, name the folder database and click **Finish**. The new folder is created.
3. In the Database Explorer panel, right-click the database connection created in the previous section. Select **Refresh**. Right-click the connection again and select **Copy to Project**.
4. In the Copy to Project window, navigate to the newly created folder called **database**. Click **OK** and then click **Finish**. The metadata is imported into the database folder as you can see in Figure 10-10.

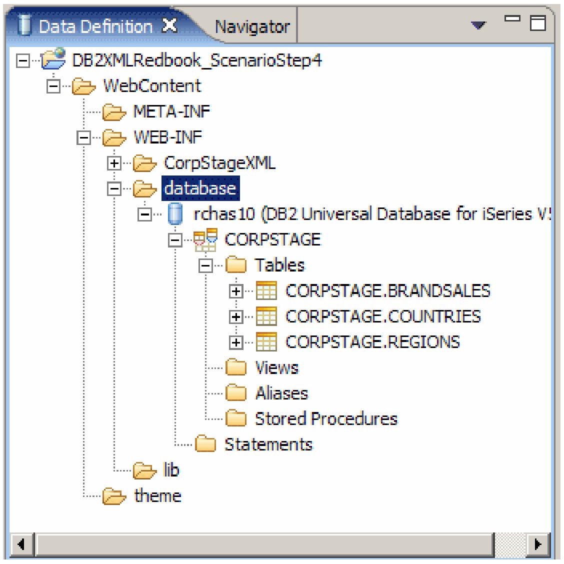
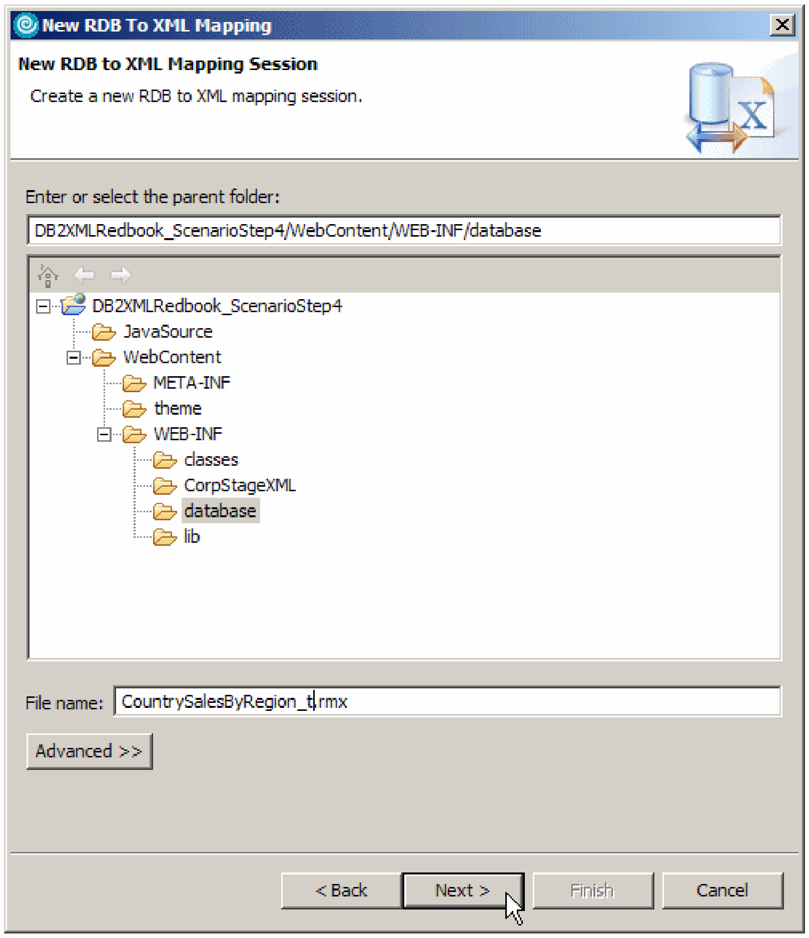


Figure 10-10 Database folder with imported CorpStage metadata

To map using the RDB-to-XML wizard:

1. Switch to the Resource perspective. Select **File** → **New** → **Other** → **RDB-to-XML mapping**. Then click **Next**.
2. In the New RDB to XML Mapping Session panel (Figure 10-11), select database as the target folder and change the mapping file name to `CountrySalesByRegion_t.rmx`. Click **Next**.



New RDB To XML Mapping

New RDB to XML Mapping Session

Create a new RDB to XML mapping session.

Enter or select the parent folder:

DB2XMLRedbook_ScenarioStep4/WebContent/WEB-INF/database

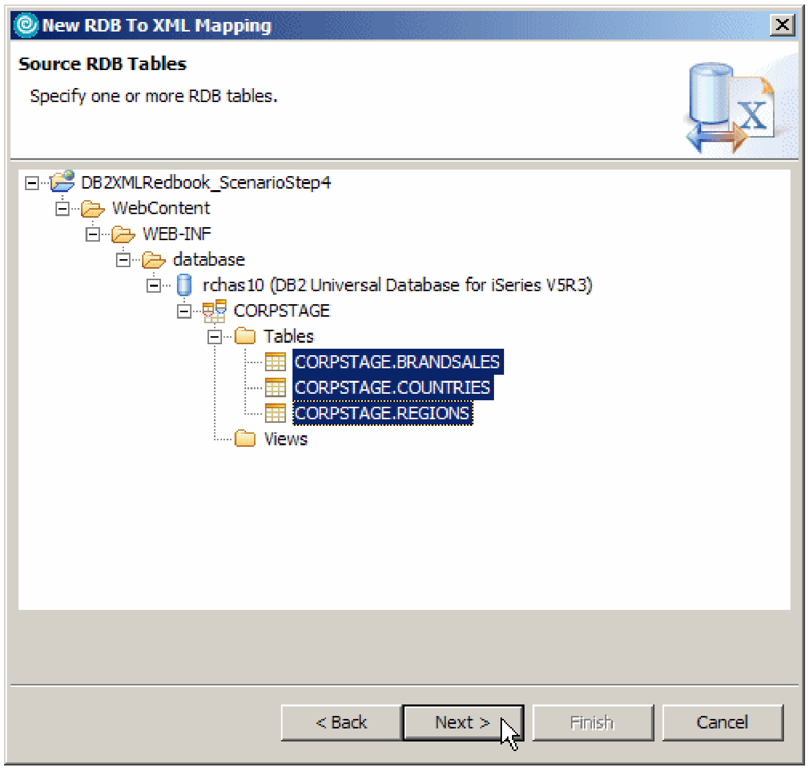
```
DB2XMLRedbook_ScenarioStep4
    JavaSource
    WebContent
        META-INF
        theme
        WEB-INF
            classes
            CorpStageXML
            database
            lib
```

File name: CountrySalesByRegion_t.rmx

Figure 10-11 New RDB to XML mapping

3. In the Choose Mapping window, leave the **RDB table to XML mapping** radio button selected. Click **Next**.

4. In the Source RDB Tables panel, navigate to the database folder and select all three tables as shown in Figure 10-12. Click **Next**.


New RDB To XML Mapping

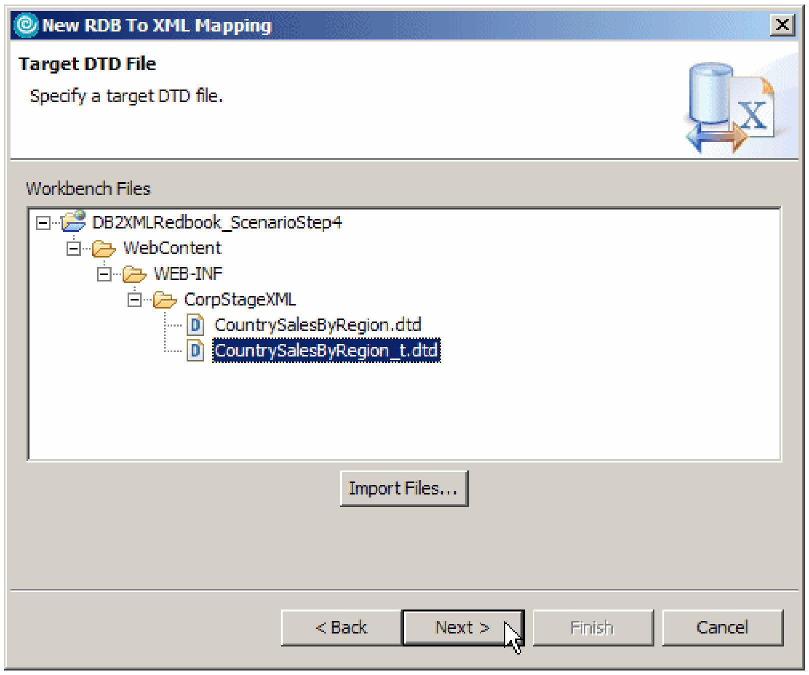
Source RDB Tables

Specify one or more RDB tables.

DB2XMLRedbook_ScenarioStep4
WebContent
WEB-INF
database
rchas10 (DB2 Universal Database for iSeries V5R3)
CORPSTAGE
Tables
CORPSTAGE.BRANDSALES
CORPSTAGE.COUNTRIES
CORPSTAGE.REGIONS
Views

Figure 10-12 Specifying the RDB tables for mapping

5. In the Target DTD File panel, navigate to the **CorpStageXML** folder and select the **CountrySalesByRegion_t.dtd** file as shown in Figure 10-13. Click **Next**.


New RDB To XML Mapping

Target DTD File

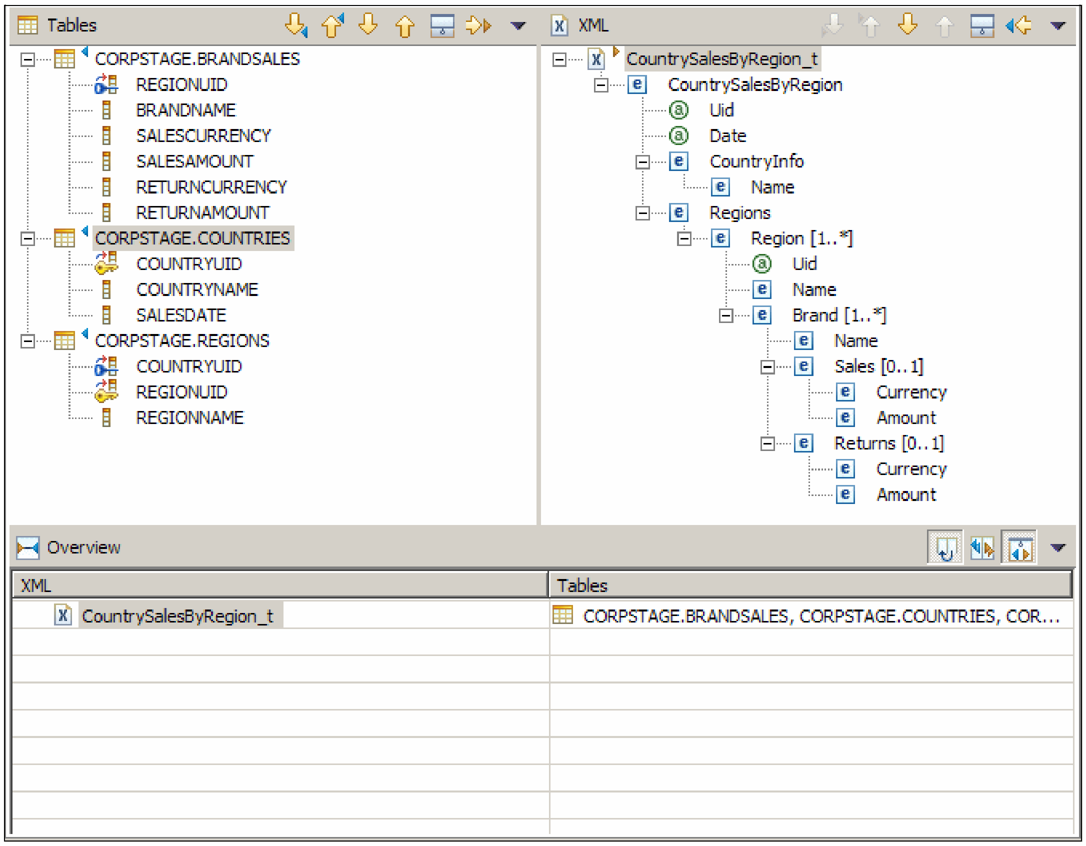
Specify a target DTD file.

Workbench Files

DB2XMLRedbook_ScenarioStep4
WebContent
WEB-INF
CorpStageXML
CountrySalesByRegion.dtd
CountrySalesByRegion_t.dtd

Figure 10-13 Specifying the DTD for mapping

6. In the Root Element panel, leave the Root Element unchanged. Click **Finish**.
7. The graphical RDB to XML mapping editor opens in the main panel. The mapping editor allows you to drag a column from a target table and drop it on an element or attribute in the XML structure. That way, you provide the information about the required mapping. Figure 10-14 shows the mapping editor session.



```
CORPSTAGE.BRANDSALES              CountrySalesByRegion_t
    REGIONUID                         CountrySalesByRegion
    BRANDNAME                             Uid
    SALESCURRENCY                         Date
    SALESAMOUNT                           CountryInfo
    RETURNCURRENCY                            Name
    RETURNAMOUNT                          Regions
CORPSTAGE.COUNTRIES                           Region [1..*]
    COUNTRYUID                                    Uid
    COUNTRYNAME                                   Name
    SALESDATE                                     Brand [1..*]
CORPSTAGE.REGIONS                                     Name
    COUNTRYUID                                        Sales [0..1]
    REGIONUID                                             Currency
    REGIONNAME                                            Amount
                                                      Returns [0..1]
                                                          Currency
                                                          Amount
```

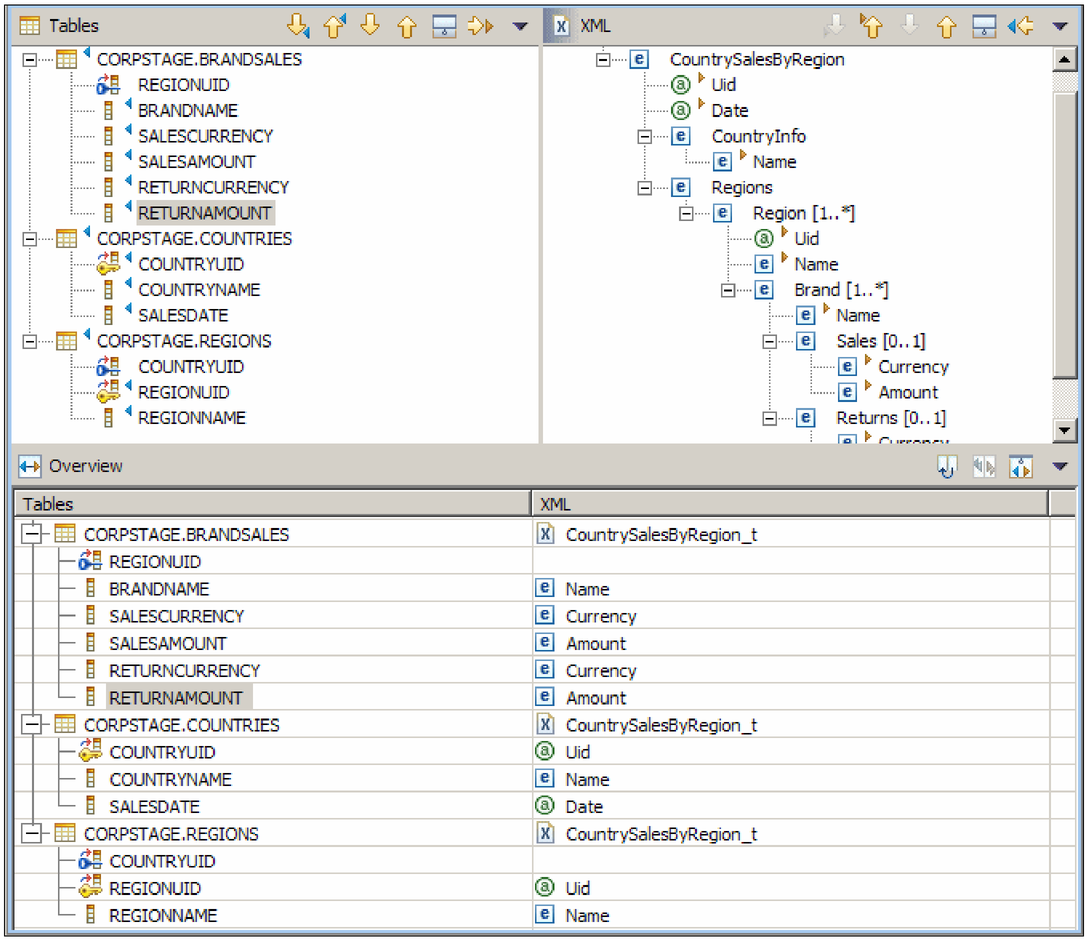
XML	Tables
CountrySalesByRegion_t	CORPSTAGE.BRANDSALES, CORPSTAGE.COUNTRIES, COR...

Figure 10-14 New RDB to XML mapping session

Using Table 10-1 on page 160 as a guide, map the columns from the tables on the left to the elements and attributes on the right. Start with the Countries table and work your way down the hierarchy described in the DTD. Leave REGIONS.COUNTRY and BRANDSALES.REGIONUID unmapped.

Notice that the columns are automatically populated by XML Extender at the time the XML documents are shredded. For example, XML Extender uses the current value of the COUNTRIES.COUNTRYUID unique key to populate the REGIONS.COUNTRYUID foreign key column.

Figure 10-15 shows the final mapping.



Tables	XML
CORPSTAGE.BRANDSALES	CountrySalesByRegion_t
REGIONUID	
BRANDNAME	Name
SALESCURRENCY	Currency
SALESAMOUNT	Amount
RETURNCURRENCY	Currency
RETURNAMOUNT	Amount
CORPSTAGE.COUNTRIES	CountrySalesByRegion_t
COUNTRYUID	Uid
COUNTRYNAME	Name
SALESDATE	Date
CORPSTAGE.REGIONS	CountrySalesByRegion_t
COUNTRYUID	
REGIONUID	Uid
REGIONNAME	Name

Figure 10-15 Final mapping

8. In addition to column-to-element/attribute mapping, you must also specify the join condition for the target tables. Use the unique key/foreign key pairs for that purpose.
   a. Select **Mapping** → **Edit Join Condition**.
   b. In the Edit Join Conditions panel, specify the following condition:

   ```
   CORPSTAGE.COUNTRIES.COUNTRYUID=CORPSTAGE.REGIONS.COUNTRYUID AND
   CORPSTAGE.REGIONS.REGIONUID=CORPSTAGE.BRANDSALES.REGIONUID
   ```

   c. Click **Finish**.
9. Select **Mapping** → **Generate DAD**.
10. In the window that opens, navigate to the **CorpStageXML** folder. Leave the DAD name unchanged. Click **Finish**. CountrySalesByRegion_t.dad is generated.
11. Save the mapping session into the rmx file.

Important: The current mapping session configuration is saved in the rmx file. The mapping wizard recognizes changes in the underlying database metadata or DTD. The process of maintaining DADs can be straightforward. You can open or re-open the existing rmx file and provide mapping for new or changed elements. Then you generate a new version of the DAD file. This is probably the most important advantage of the proposed methodology. As opposed to the programmatic approach changes in the relational database model or in the XML hierarchy can be accommodated with basically no programming effort.

Example 10-8 shows an excerpt from the DAD file that is generated by the RDB-to-XML Mapping wizard.

Example 10-8 CountrySalesByRegion_t.dad

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD PUBLIC "dadId" "dad.dtd">
<DAD>
  <dtdid>CountrySalesByRegion_t.dtd</dtdid>
  <validation>NO</validation>
  <Xcollection> 1
    <prolog>?xml version="1.0"?</prolog>
      <root_node>
      <element_node name="CountrySalesByRegion">
        <RDB_node> 2
          <table name="CORPSTAGE.BRANDSALES"/>
          <table name="CORPSTAGE.COUNTRIES" key="COUNTRYUID"/>
          <table name="CORPSTAGE.REGIONS" key="REGIONUID"/>
          <condition>
            CORPSTAGE.COUNTRIES.COUNTRYUID=CORPSTAGE.REGIONS.COUNTRYUID AND
            CORPSTAGE.REGIONS.REGIONUID=CORPSTAGE.BRANDSALES.REGIONUID
          </condition>
        </RDB_node>
        <attribute_node name="Uid">
          <RDB_node>
            <table name="CORPSTAGE.COUNTRIES"/>
            <column name="COUNTRYUID" type="Character(30)"/>
          </RDB_node>
        </attribute_node>
        <attribute_node name="Date">
          <RDB_node>
            <table name="CORPSTAGE.COUNTRIES"/>
            <column name="SALESDATE" type="Date"/>
          </RDB_node>
        </attribute_node>
        <element_node name="CountryInfo">
          <element_node name="Name">
            <text_node>
              <RDB_node>
                <table name="CORPSTAGE.COUNTRIES"/>
                <column name="COUNTRYNAME" type="VarChar(50)"/>
              </RDB_node>
            </text_node>
          </element_node>
        </element_node>
        <element_node name="Regions">
```

```
          <element_node name="Region" multi_occurrence="YES"> 3
            <attribute_node name="Uid">
              <RDB_node>
                <table name="CORPSTAGE.REGIONS"/>
                <column name="REGIONUID" type="Character(30)"/>
              </RDB_node>
            </attribute_node>
...
```

At line 1 in Example 10-8, we indicate that the DAD is used for the XCollection access method. At line 2, the root element RDB_node is used to define all relational tables used in the mapping. Note that the unique keys are specified for those table that have them. The condition element, in turn, defines the join condition. At line 3, the Region element that can repeat has the multi-occurrence attribute set to “YES”.

10.8 Creating the wrapper stored procedure

There are many methods to invoke XML Extender functions and procedures. For the sake of simplicity, we decided to provide a wrapper SQL stored procedure that, in turn, invokes the dxxShredXML stored procedure provided by XML Extender. The wrapper will enable us to run the solution from the iSeries Navigator Run SQL Scripts utility and monitor the output parameters that indicate if the shredding was successful.

We discuss briefly the content of the wrapper stored procedure. Example 10-9 shows the source code.

Example 10-9 Source code for the wrapper stored procedure

```
CREATE PROCEDURE CORPSTAGE.SHREDXMLWrapper (
   IN DAD_FILE VARCHAR(512) ,
   IN XML_FILE VARCHAR(512) ,
   OUT ERRCODE INTEGER ,
   OUT ERRMSG VARCHAR(1024) )
   LANGUAGE SQL
P1: BEGIN
DECLARE DAD_BUF CLOB(102400) ;
DECLARE XMLOBJ CLOB(1048576);

SET DAD_BUF = DB2XML.XMLCLOBFROMFILE ( DAD_FILE );1
SET XMLOBJ = DB2XML.XMLCLOBFROMFILE( XML_FILE );  2
CALL DB2XML . DXXSHREDXML ( DAD_BUF , XMLOBJ, ERRCODE , ERRMSG ); 3
 IF ERRCODE < 0 THEN
    SET ERRMSG = 'ErrCode=' || TRIM ( CHAR ( ERRCODE ) ) || ' - ' || ERRMSG ;
    SIGNAL SQLSTATE '70001' SET MESSAGE_TEXT = ERRMSG ;
 END IF ;
END P1  ;
```

In line 1 in Example 10-9, the DAD file is read into a CLOB variable. In line 2, the XML file to be shredded is read into another CLOB variable. In line 3, the XML Extender stored procedure is called to perform the shredding. The purpose of the wrapper is to read the DAD and XML from a file system and pass them to XML Extender as CLOBs.

10.9 Deploying the XSL, DAD, Java, and SQL to the integrated file system on the System i machine

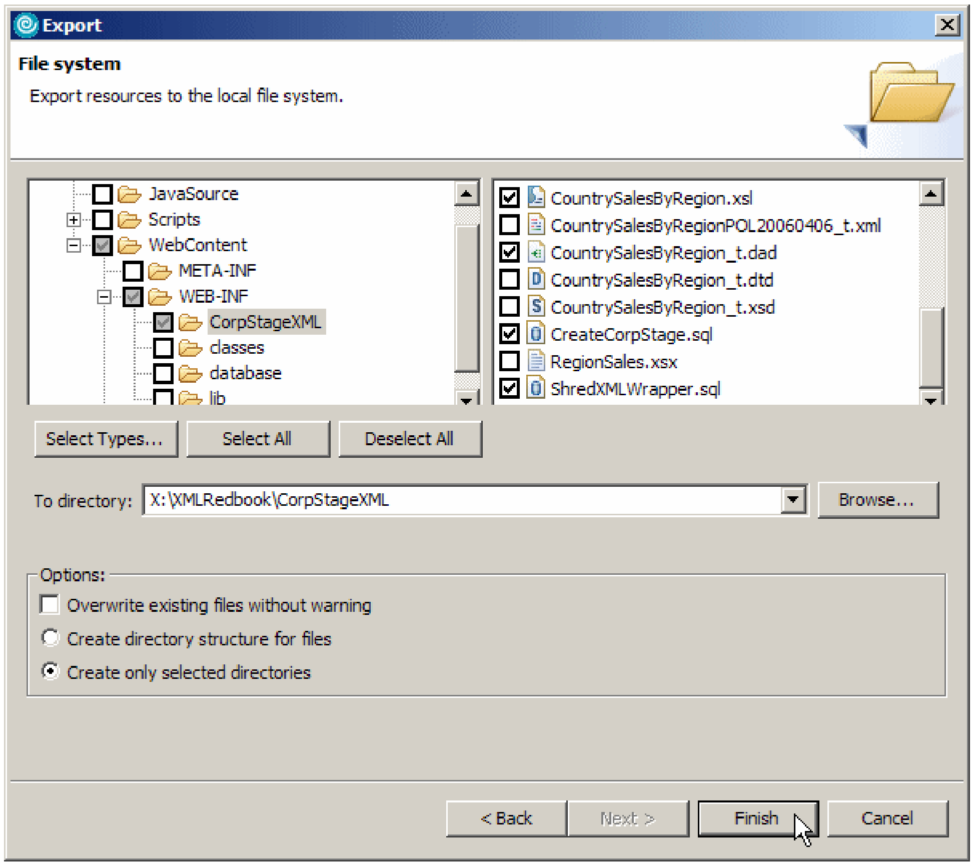
We start with a quick inventory of all the programming artifacts, which we need to deploy a workable solution:

- **Inbound XML documents**: In our scenario, these documents are created by countries and sent to the corporate headquarters for processing. These XML files reside in the /XMLRedbook/CountryXML directory on the target System i machine.
- **CountrySalesByRegion.xsl**: This stylesheet is needed to insert the unique ID attributes into the inbound XML document.
- **Transform.class**: This is the Java program that performs the transformation.
- **CountrySalesByRegion_t.dad**: This DAD is used by XML Extender to shred the transformed inbound XML documents into the target database.
- **CreateCorpStage.sql**: This SQL DDL script is executed to create the target schema called CORPSTAGE and the necessary tables.
- **ShredXMLWrapper.sql**: This SQL script is used to create the wrapper stored procedure.

The solution is deployed into the /XMLRedbook/CorpStageXML directory in the integrated file system. The compiled Java code resides in /XMLRedbook/classes. The following steps outline the deployment process:

1. We assume that the integrated file system Root directory on the target System i machine is mapped to a local X: drive on the development workstation. In WebSphere Development Studio Client, right-click the **CorpStageXML** folder and select **Export**.
2. In the window that opens, select **File System** and click **Next**.

3. In the Export window, select the four files as shown in Figure 10-16. Click **Browse** to navigate to the CorpStageXML folder in the integrated file system. Click **Finish**.



☑ CountrySalesByRegion.xsl
☐ CountrySalesByRegionPOL20060406_t.xml
☑ CountrySalesByRegion_t.dad
☐ CountrySalesByRegion_t.dtd
☐ CountrySalesByRegion_t.xsd
☑ CreateCorpStage.sql
☐ RegionSales.xsx
☑ ShredXMLWrapper.sql

To directory: X:\XMLRedbook\CorpStageXML

Options:
☐ Overwrite existing files without warning
○ Create directory structure for files
● Create only selected directories

Figure 10-16 Exporting the application files

4. Export Transform.class into /XMLRedbook/classes.
5. In iSeries Navigator, start the Run SQL Scripts utility. Navigate to the CreateCorpStage.sql script in the X:\XMLRedbook\CorpStageXML directory. Run the script.
6. Still in the Run SQL Scripts utility, open the ShredXMLWrapper.sql script and execute it.

The application is now ready to shred inbound documents.

10.10 Running the deployed application

When the application is deployed, a few easy steps are required to shred any number of inbound XML documents. We use the iSeries Navigator Run SQL Scripts utility to execute the script shown in Example 10-10.

Example 10-10 Script to transform and shred inbound XML documents

```
set path=corpstage;
set schema=corpstage;
-- cascade delete to purge the data from corpstage
delete from countries;
commit;
CL: cd '/XMLRedbook/classes';
-- Country Sales by region; one sales date, one country
CL: RUNJVA CLASS(Transform)
PARM('/XMLRedbook/CountryXML/CountryByRegionPOL20060406.xml'
'/XMLRedbook/CorpStageXML/CountrySalesByRegion.xsl'
'/XMLRedbook/CorpStageXML/CountryByRegionPOL20060406_t.xml'
'2006-04-20-13:08:00'); 1
CALL SHREDXMLWRAPPER('/XMLRedbook/CorpStageXML/CountrySalesByRegion_t.dad',
'/XMLRedbook/CorpStageXML/CountryByRegionPOL20060406_t.xml',0,''); 2
commit;
```

If everything works fine, the stored procedure returns the following message:

```
Output Parameter #3 = 0
Output Parameter #4 = DXXQ025I  XML decomposed successfully.
```

At line **1** in Example 10-10, the RUNJVA CL command is used to load and execute the Transform class. This produces the transformed XML document that is then used at line **2** as an input to the SHREDXMLWRAPPER stored procedure. You need to repeat the steps from lines **1** and **2** to shred the remaining five XML documents generated by step 4 of our scenario.

The data from the several XML documents is now stored in DB2.

11

Composing methodology

In Chapter 10, “Shredding methodology” on page 155, we present a practical approach that can be used to shred (decompose) XML documents of any level of complexity into a relational database. In this chapter, we present a production system hardened methodology that was devised to generate complex XML documents from existing databases.

Additionally, we try to further increase your familiarity with DB2 XML Extender and XML tooling available in WebSphere Development Studio Client. We use these products to implement step 5 (see Figure 11-1) in our scenario.

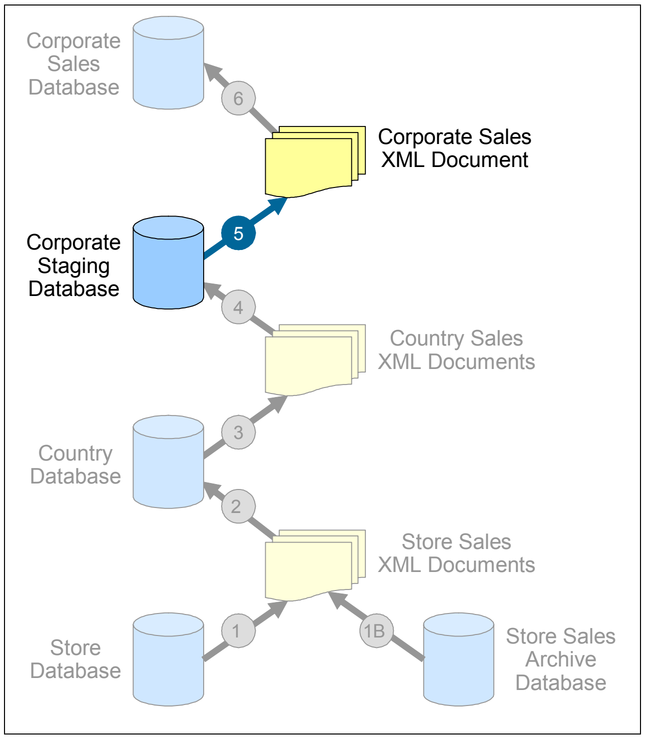

Corporate Sales Database
6
Corporate Sales XML Document
5
Corporate Staging Database
4
Country Sales XML Documents
3
Country Database
2
Store Sales XML Documents
1
1B
Store Database
Store Sales Archive Database

Figure 11-1 Scenario step 5

In this step, we generate outbound XML documents that contain sales data for a given date by country and by brand. The region data is rolled up (summarized). The generated XML documents are used to feed the CorpSales database. Refer to Chapter 2, “Scenario overview” on page 21, for details.

Similar to the shredding methodology, the composing methodology can be divided into two phases:

- *Design and implementation*: In this phase, we analyze the hierarchy of the outbound XML document and the structure of the existing database. We create the necessary view that projects the summarized data, and we determine how to map the resulting relational structure into the required XML hierarchy. Then we create the document access definition (DAD).
- *Deployment*: In this phase, we deploy the solution to the target System i machine and run it to compose (generate) multiple outbound XML documents.

11.1 Setting up the development environment

As mentioned, we use WebSphere Development Studio Client to manage the development project. First we create a new Dynamic Web Project called DB2XMLRedbook_ScenarioStep5. Then we remove the additional DB2XMLRedbook_ScenarioStep5EAR project that was automatically generated by WebSphere Development Studio Client as part of the Dynamic Web Project creation process. We do not create an enterprise archive (EAR) file so we do not need this additional project.

Next we create a new folder called CorpXML in the /DB2XMLRedbook_ScenarioStep5/WebContent/WEB-INF folder. The new folder will contain XML-related files such as document type definition (DTD) and DAD files.

We begin the development process by importing the DTD for the outbound XML documents. The DTD is based on the XML Schema Definition (XSD) that is provided by the company headquarters. Our task is to retrieve the sales data from the CorpStage database and report it by sales date and country to the headquarters. The DTD resides on the System i machine in the integrated file system /XMLRedbook/CorpXML directory. We assume that the integrated file system root directory is mapped on the development workstation as network drive X:.

1. In the Resource perspective’s Navigator panel in WebSphere Development Studio Client, right-click the **CorpXML** folder and select **Import**.
2. In the Import panel, select **File System** and click **Next**.
3. In the File System panel, click **Browse** next to the From Directory text box. Navigate to the /XMLRedbook/CorpXML integrated file system directory. Select **CorpSales.dtd**. Then click **Finish**. The DTD is imported into the project.

As mentioned, we use CorpStage schema as the data source that feeds the XML composition process. Therefore, we must import the CorpStage metadata into the current project:

1. Switch to the Data perspective. In the Navigator panel, right-click the **WEB-INF** folder. Select **New → Folder**.
2. In the New Folder window, name the folder `database` and click **Finish**. The new folder is then created.

3. You should already have the database connection defined in the Database Explorer panel. If not, refer to 10.6, “Creating tables on the database server” on page 169, for instructions on how to create the connection to the System i machine.

   In the Database Explorer panel, right-click the existing database connection and select **Refresh**. This refreshes the metadata from the database server. Right-click the connection again and select **Copy to Project**.

4. In the Copy to Project window, navigate to the newly created folder called **database**. Click **OK** and then click **Finish**. The metadata is imported into the database folder as shown in Figure 11-2.

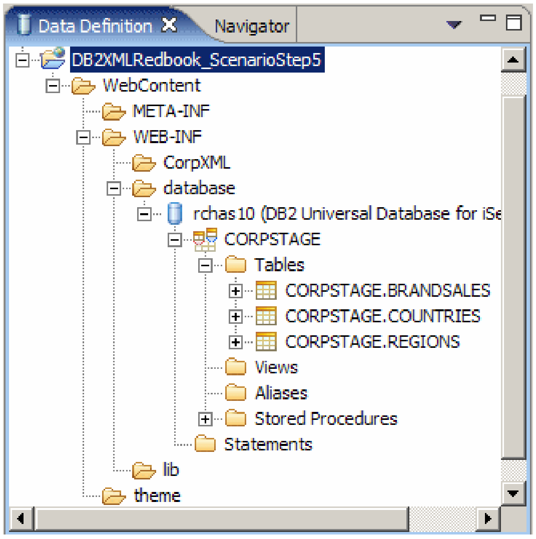


Figure 11-2 Importing the CorpStage metadata

11.2 Analyzing the database structure and outbound XML hierarchy

First we must understand the hierarchy of the XML document requested by headquarters. We use the imported DTD for that purpose. Example 11-1 shows the source code. Notice that the elements in the DTD have been rearranged so that the DTD reflects the hierarchy of an outbound document.

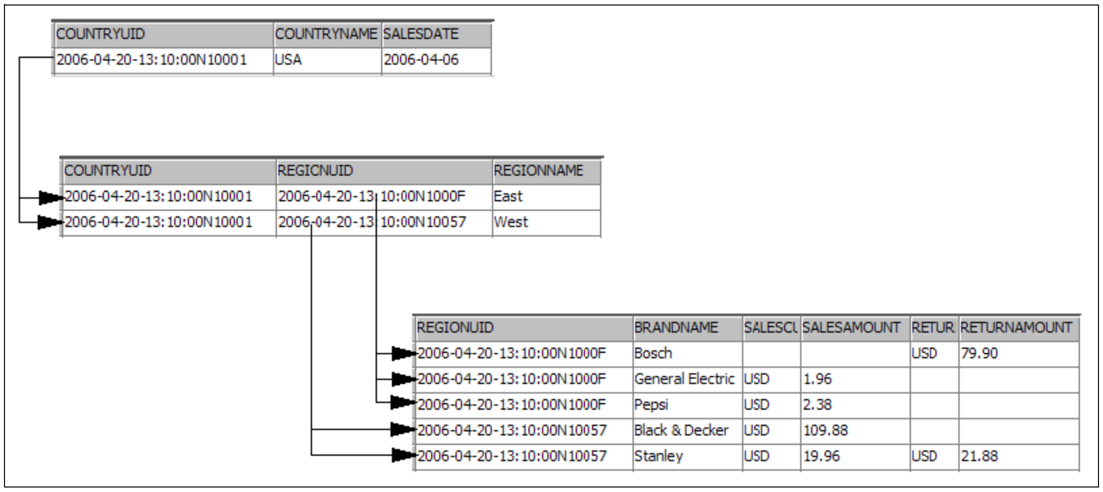
Example 11-1 CorpSales.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Amount (#PCDATA)>
<!ELEMENT CorpSales (CountryInfo,SalesByBrand) > 1
<!ATTLIST CorpSales
  Date CDATA #IMPLIED
>
<!ELEMENT CountryInfo (Name) >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Currency (#PCDATA)>
<!ELEMENT SalesByBrand (Brand+) > 2
<!ELEMENT Brand (Name,Sales?,Returns?) > 3
<!ELEMENT Sales (Currency,Amount) >
<!ELEMENT Returns (Currency,Amount) >
```

In the DTD shown in Example 11-1, there are two wrapper elements, CountryInfo and SalesByBrand (in line **1** in Example 11-1). One element can repeat, which is Brand (in line **2**). There are two optional elements, Sales and Returns (in line **3**).

To populate CorpSales/@Date attribute we must retrieve a specific sales data from the source database. Similarly, we must retrieve the country name to populate the /CorpSales/CountryInfo/Name text node. The sales data is grouped by brand. Consequently there are many Brand elements for any particular sales date and country. To express these concepts in relational database terms, we can say that every unique row that contains sales date and country name will have a number of dependent rows that contain sales data grouped by brand for this particular date and country.

Let us revisit now the CorpStage schema to determine how we can map the existing database structure into CorpSales DTD. Figure 11-3 illustrates the database schema.



COUNTRYUID	COUNTRYNAME	SALESDATE
2006-04-20-13:10:00N10001	USA	2006-04-06

COUNTRYUID	REGIONUID	REGIONNAME
2006-04-20-13:10:00N10001	2006-04-20-13:10:00N1000F	East
2006-04-20-13:10:00N10001	2006-04-20-13:10:00N10057	West

REGIONUID	BRANDNAME	SALESCU	SALESAMOUNT	RETUR	RETURNAMOUNT
2006-04-20-13:10:00N1000F	Bosch			USD	79.90
2006-04-20-13:10:00N1000F	General Electric	USD	1.96		
2006-04-20-13:10:00N1000F	Pepsi	USD	2.38		
2006-04-20-13:10:00N10057	Black & Decker	USD	109.88		
2006-04-20-13:10:00N10057	Stanley	USD	19.96	USD	21.88

Figure 11-3 CorpStage schema

The Countries table contains one row per country and sales date. We can retrieve this data to populate /CorpSales/CountryInfo/Name and /CorpSales/@Date respectively. BrandSales table, however, contains data that is too granular. The corporate headquarters is not interested in sales data by country, region, and brand. Headquarters wants to see the data summarized by country and brand. This can be achieved by creating the view shown in Example 11-2 that rolls-up (summarizes) the region sales data into a single row by country and brand.

Example 11-2 CreateV_Corpsales.sql

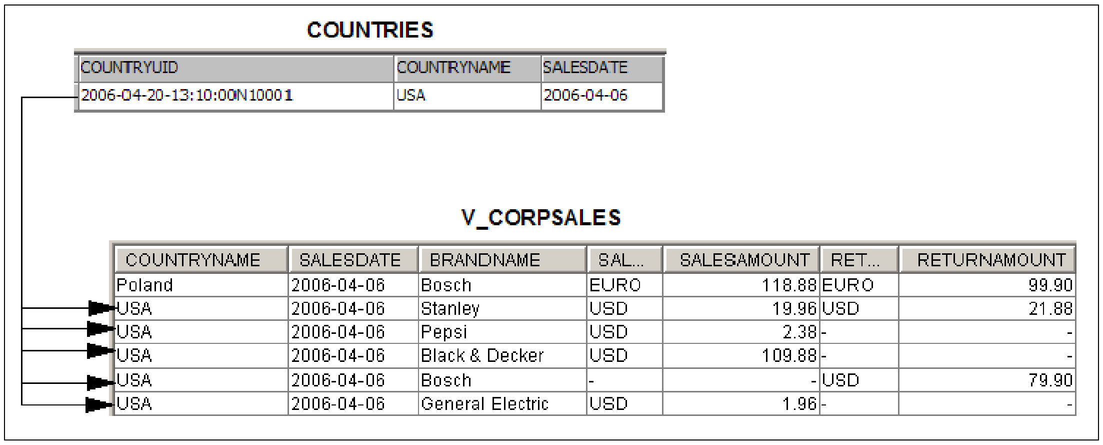
```
CREATE VIEW CORPSTAGE.V_CORPSALES
AS
(
 SELECT COUNTRYNAME, SALESDATE, BRANDNAME,
 SALESCURRENCY, SUM(SALESAMOUNT) AS SALESAMOUNT,
 RETURNCURRENCY, SUM(RETURNAMOUNT) AS RETURNAMOUNT
 FROM CORPSTAGE.COUNTRIES COUNTRIES, CORPSTAGE.REGIONS REGIONS,
      CORPSTAGE.BRANDSALES BRANDSALES
 WHERE COUNTRIES.COUNTRYUID = REGIONS.COUNTRYUID
      AND REGIONS.REGIONUID = BRANDSALES.REGIONUID
 GROUP BY COUNTRYNAME,SALESDATE,BRANDNAME, SALESCURRENCY, RETURNCURRENCY
 );
```

With the new view, the mapping of the remaining unmapped elements that is Brand and its children is quite simple. As shown in the DTD, Brand has three children: Name, Sales, and Returns. None of these elements can occur more than once so one occurrence of a Brand element along with its children can be mapped to a single row of the V_CORPSALES view. Table 11-1 documents the mapping.

Table 11-1 XML hierarchy mapped to relational tables

Location path	Table/view name	Column name	Nullable
/CorpSales/@Date	COUNTRIES	SALESDATE	N
/CorpSales/CountryInfo/Name	COUNTRIES	COUNTRYNAME	N
/CorpSales/SalesByBrand/Brand/Name	V_CORPSALES	BRANDNAME	N
/CorpSales/SalesByBrand/Brand/Sales/Currency	V_CORPSALES	SALESCURRENCY	Y
/CorpSales/SalesByBrand/Brand/Sales/Amount	V_CORPSALES	SALESAMOUNT	Y
/CorpSales/SalesByBrand/Brand/Returns/Currency	V_CORPSALES	RETURNCURRENCY	Y
/CorpSales/SalesByBrand/Brand/Returns/Amount	V_CORPSALES	RETURNAMOUNT	Y

Figure 11-4 shows the DB2 objects that are necessary to populate the outbound XML document. The sample data is provided to illustrate the one-to-many relationship between the rows in the Countries table and the V_CORPSALES view.



COUNTRIES

COUNTRYUID	COUNTRYNAME	SALESDATE
2006-04-20-13:10:00N10001	USA	2006-04-06

V_CORPSALES

COUNTRYNAME	SALESDATE	BRANDNAME	SAL...	SALESAMOUNT	RET...	RETURNAMOUNT
Poland	2006-04-06	Bosch	EURO	118.88	EURO	99.90
USA	2006-04-06	Stanley	USD	19.96	USD	21.88
USA	2006-04-06	Pepsi	USD	2.38	-	-
USA	2006-04-06	Black & Decker	USD	109.88	-	-
USA	2006-04-06	Bosch	-	-	USD	79.90
USA	2006-04-06	General Electric	USD	1.96	-	-

Figure 11-4 Layout of the source database

In Figure 11-4, the arrows indicate that one row in the Countries table has multiple dependent rows in the V_CORPSALES view. This correctly reflects the hierarchy of the outbound XML document where there are multiple Brand elements that are descendents of the root element CorpSales. To preserve the relationships represented by arrows, we join Countries to V_CORPSALES using COUNTRYNAME and SALESDATE as the join columns. Example 11-3 shows the necessary join condition.

Example 11-3 Join condition

```
COUNTRIES.COUNTRYNAME=V_CORPSALES.COUNTRYNAME
AND COUNTRIES.SALESDATE=V_CORPSALES.SALESDATE
```

11.3 Creating the SQL view to project the data

We create the necessary view V_CORPSALES by creating the CreateV_Corpsales.sql script and deploying it on the target DB2 server:

1. In WebSphere Development Studio Client, switch to the Data perspective. Right-click the **CorpXML** folder and select **New** → **SQL/DDL Script**.
2. In the Create SQL Script File window, enter the script name `CreateV_CorpSales.sql`. Then click **Finish**.
3. Copy the source code provided in Example 11-2 on page 184 into the script. Save the content.
4. Right-click the newly created DDL script **CreateV_CorpSales** and select **Deploy**.
5. In the Run Script window, click **Next**.
6. In the Run Script Option window, leave the **Commit changes only upon success** option selected and click **Next**.
7. In the Database Connection window, make sure that the **Use existing connection** option is selected and click **Finish**.

The view is then created in the CorpStage schema on the DB2 server.

11.4 Graphically mapping the relational database to XML (generating DAD)

Similar to the shredding methodology, we use the RDB to XML Mapping wizard to provide the necessary relational database to XML mapping. The wizard uses the source database metadata and DTD as input parameters. It produces a DAD file as output. The DAD file can then be used by XML Extender to generate outbound XML documents.

Before we activate the wizard, we refresh the metadata imported into the current project (refer to 11.1, “Setting up the development environment” on page 182, for details about how to import the metadata):

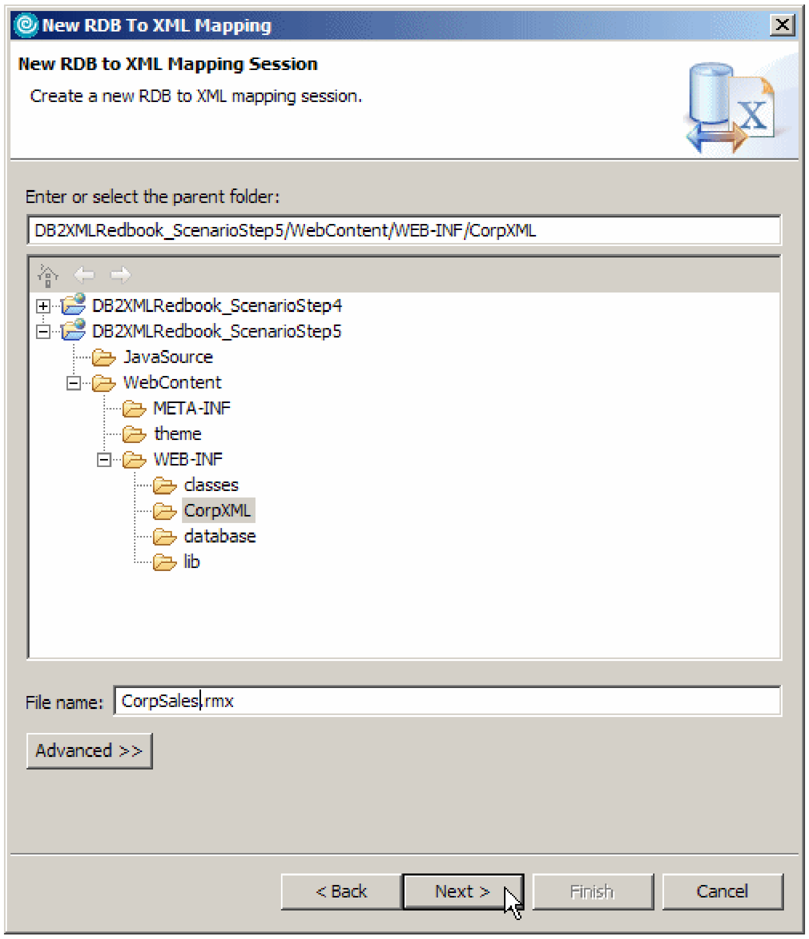
1. In the Database Explorer panel, right-click the database connection for your DB2 server. Select **Refresh**.
2. Right-click the connection again and select **Copy to Project**.
3. In the Copy to Project window, navigate to the folder called **database**. Click **OK** and then click **Finish**. The metadata is then imported into the database folder.

The refreshed metadata contains the information on the newly created V_CORPSALES view. Verify this by expanding the database folder. The view should now appear in the Views folder.

To map using the RDB-to-XML wizard:

1. Switch to the Resource perspective and then select **File** → **New** → **Other** → **RDB-to-XML mapping**. Click **Next**.

2. In the New RDB to XML Mapping Session panel (Figure 11-5), select **database** as the target folder and change the mapping file name to CountrySales.rmx. Click **Next**.



Create a new RDB to XML mapping session.

Enter or select the parent folder:
DB2XMLRedbook_ScenarioStep5/WebContent/WEB-INF/CorpXML

File name: CorpSales.rmx

Figure 11-5 New RDB to XML Mapping Session panel

3. In the Choose Mapping panel, leave the **RDB table to XML mapping** radio button selected and click **Next**.

4. In the Source RDB Tables panel, navigate to the database folder. Select the **CORPSTAGE.COUNTRIES** table and the **CORPSTAGE.V_CORPSALES** view as shown in Figure 11-6. Click **Next**.

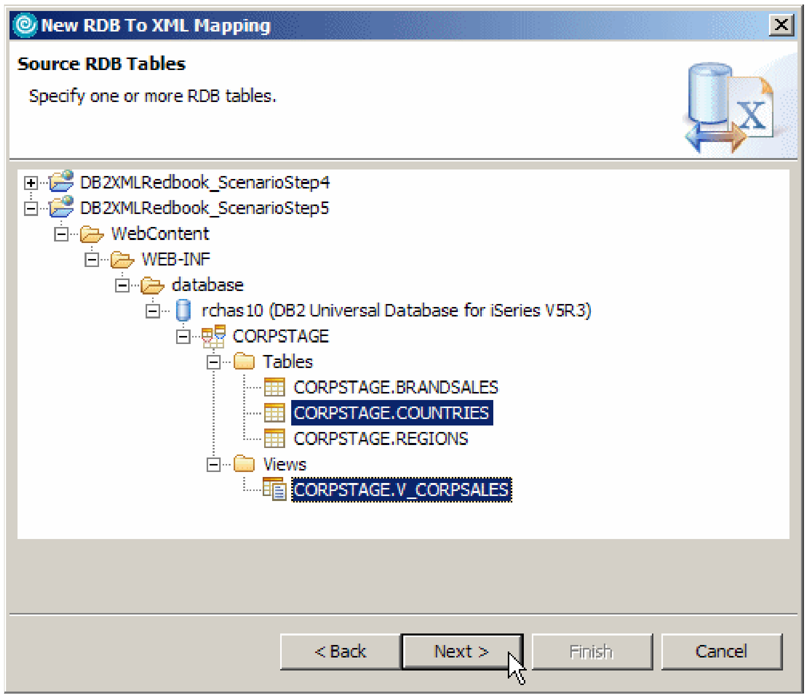


Figure 11-6 Specifying the RDB objects for mapping

5. In the Target DTD File panel (Figure 11-7), navigate to the **CorpXML** folder and select the **CorpSales.dtd** file. Click **Next**.

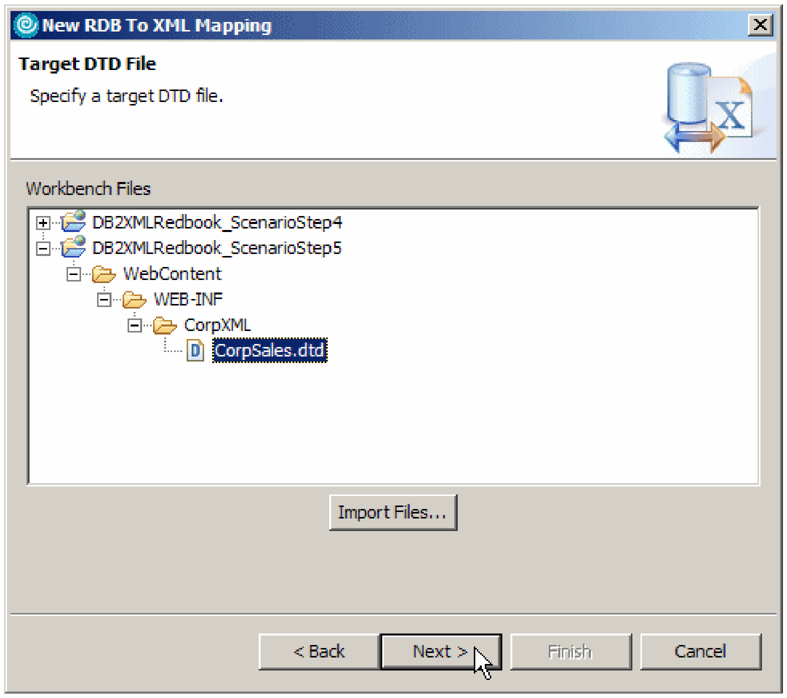
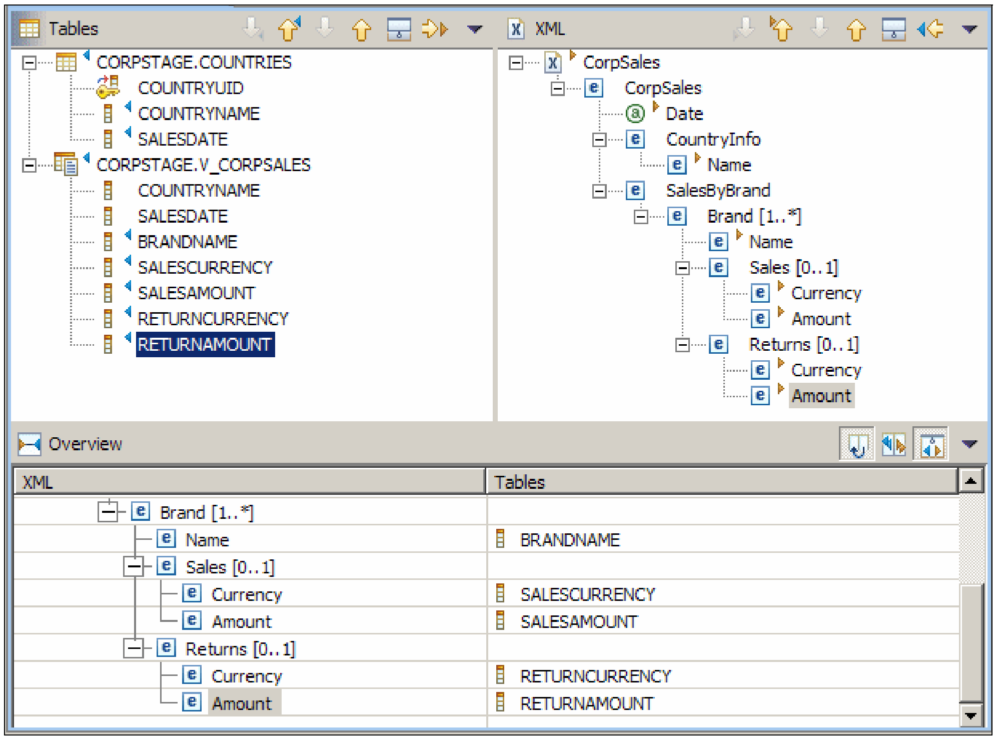


Figure 11-7 Specifying the DTD for mapping

6. In the Root Element window, leave the Root Element as is. Click **Finish**.
7. The graphical RDB to XML mapping editor opens in the main panel. The mapping editor enables you to drag a column from a source table and drop it on an element or attribute in the XML structure. That way, you provide the information about the required mapping.

   Using Table 11-1 on page 185 as a guide, map the columns from the DB2 objects on the left to the elements and attributes on the right. Start with the Countries table and work your way down the hierarchy described in the DTD.

   Note that COUNTRYUID column is not used in the mapping. Similarly, COUNTRYNAME and SALESDATE columns in the V_CORPSALES view are not mapped, because the matching columns are already mapped in the COUNTRIES table. Figure 11-8 shows the final mapping.



XML	Tables
Brand [1..*]	
Name	BRANDNAME
Sales [0..1]	
Currency	SALESCURRENCY
Amount	SALESAMOUNT
Returns [0..1]	
Currency	RETURNCURRENCY
Amount	RETURNAMOUNT

Figure 11-8 Final RDB to XML mapping

8. Select **Mapping** → **Generate DAD**.
9. In the Join Conditions Not Specified window, click **Continue**.
10. In the next window that opens, navigate to the **CorpXML** folder. Leave the DAD name as is and click **Finish**. The CorpSales.dad is then generated.
11. Save the mapping session into the rmx file.

As you might have noticed, we did not specify the join condition in the RDB to XML mapping wizard. We provide the necessary join condition by manually editing the generated DAD. More importantly, we also add the search conditions to the DAD. Note that currently the DAD has no conditions that limit the amount of data selected from the CorpStage schema. XML Extender can generate a separate outbound XML document for every country and every sales date for which the data exists in the CorpStage schema. However, we want the flexibility to generate XML only for a given date or a given country or a combination of the two conditions.

We tweak the DAD source so that it meets all our requirements. Example 11-4 shows the modified DAD.

Example 11-4 Modified CorpSales.dad

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD PUBLIC "dadId" "dad.dtd">
<DAD>
  <dtdid>CorpSales.dtd</dtdid>
  <validation>NO</validation>
  <Xcollection>
    <prolog>?xml version="1.0"?</prolog>
    <doctype>!DOCTYPE CorpSales PUBLIC "CorpSalesId" "CorpSales.dtd"</doctype>
    <root_node>
      <element_node name="CorpSales">
        <RDB_node>
          <table name="CORPSTAGE.COUNTRIES"
           key="SALESDATE, COUNTRYNAME" orderBy="SALESDATE, COUNTRYNAME"/> 1
          <table name="CORPSTAGE.V_CORPSALES" key="SALESDATE, COUNTRYNAME"/> 2
          <condition>
            CORPSTAGE.COUNTRIES.COUNTRYNAME=CORPSTAGE.V_CORPSALES.COUNTRYNAME
            AND CORPSTAGE.COUNTRIES.SALESDATE=CORPSTAGE.V_CORPSALES.SALESDATE 3
          </condition>
        </RDB_node>
        <attribute_node name="Date">
          <RDB_node>
            <table name="CORPSTAGE.COUNTRIES"/>
            <column name="SALESDATE" type="Date"/>
            <condition></condition> 4
          </RDB_node>
        </attribute_node>
        <element_node name="CountryInfo">
          <element_node name="Name">
            <text_node>
              <RDB_node>
                <table name="CORPSTAGE.COUNTRIES"/>
                <column name="COUNTRYNAME" type="VarChar(50)"/>
                <condition></condition> 5
              </RDB_node>
            </text_node>
          </element_node>
        </element_node>
        <element_node name="SalesByBrand">
          <element_node name="Brand" multi_occurrence="YES">
            <element_node name="Name">
              <text_node>
                <RDB_node>
                  <table name="CORPSTAGE.V_CORPSALES"/>
                  <column name="BRANDNAME" type="VarChar(80)"/>
                </RDB_node>
              </text_node>
            </element_node>
            <element_node name="Sales">
              <element_node name="Currency">
                <text_node>
                  <RDB_node>
                <table name="CORPSTAGE.V_CORPSALES"/>
                <column name="SALESCURRENCY" type="VarChar(30)"/>
              </RDB_node>
            </text_node>
          </element_node>
          <element_node name="Amount">
            <text_node>
              <RDB_node>
                <table name="CORPSTAGE.V_CORPSALES"/>
                <column name="SALESAMOUNT" type="Decimal(9,2)"/>
              </RDB_node>
            </text_node>
          </element_node>
        </element_node>
        <element_node name="Returns">
          <element_node name="Currency">
            <text_node>
              <RDB_node>
                <table name="CORPSTAGE.V_CORPSALES"/>
                <column name="RETURNCURRENCY" type="VarChar(30)"/>
              </RDB_node>
            </text_node>
          </element_node>
          <element_node name="Amount">
            <text_node>
              <RDB_node>
                <table name="CORPSTAGE.V_CORPSALES"/>
                <column name="RETURNAMOUNT" type="Decimal(9,2)"/>
              </RDB_node>
            </text_node>
          </element_node>
        </element_node>
      </element_node>
    </element_node>
  </element_node>
</root_node>
</Xcollection>
</DAD>
```

In Example 11-4, in line **1**. we list the first table used in the data retrieval. Along with the table name, we provide two additional attributes, key and orderBy. The key attribute identifies the columns that constitute a unique key. The orderBy attribute is used to define the sorting order for the rows that are retrieved from the database.

In this case, we want the rows to be sorted by SALESDATE and COUNTRYNAME. This, in turn, determines the order in which the outbound XML documents are generated. At line **2**, we specify the view along with its key. At line **3**, the join condition is defined. This condition is used to join rows from COUNTRIES table to rows in the V_CORPSALES view.

At line **4**, we specify an empty condition for the date attribute. This condition is going to be dynamically overridden with an override expression at run time to limit the number of XML documents composed by XML Extender. Refer to 11.7, "Running the deployed application" on page 194, for more details. Similarly, at line **5**, we define an empty condition for the /CorpSales/CountryInfo/Name element.

11.5 Creating the wrapper stored procedure

The last step in the development process is to create a wrapper stored procedure that invokes the XML Extender dxxGen procedure. The wrapper allows us to run the XML composition process from the iSeries Navigator Run SQL Scripts utility. This utility has the ability to return and display the stored procedure's output parameters so that we can monitor the outcome of the XML compose process.

Example 11-5 shows the source code of the GenXMLWrapper stored procedure.

Example 11-5 GenXMLWrapper.sql

```
CREATE PROCEDURE CORPSTAGE.GENXMLWRAPPER (
   IN DAD_FILE VARCHAR(512) ,
   IN RESULT_TABLE_NAME VARCHAR(160),
   IN RESULT_COLUMN_NAME VARCHAR(129),
   IN VALID_COLUMN_NAME VARCHAR(129),
   IN OVERRIDE_TYPE INTEGER ,
   IN OVERRIDE VARCHAR(16336) ,
   IN MAX_NUM_DOCS INTEGER,
   OUT NUM_DOCS INTEGER ,
   OUT ERRCODE INTEGER ,
   OUT ERRMSG VARCHAR(1024) )
   LANGUAGE SQL
   SPECIFIC CORPSTAGE.GENXMLWRAPPER
   NOT DETERMINISTIC
   MODIFIES SQL DATA
   CALLED ON NULL INPUT
   --SET OPTION DBGVIEW=*SOURCE

   P1 : BEGIN
   DECLARE DAD_BUF CLOB ( 102400 ) ;
   DECLARE RESULTDOC CLOB ( 1048576 ) ;
   DECLARE RESULT_FILEOUT VARCHAR ( 512 ) ;

   SELECT DB2XML.XMLCLOBFROMFILE(DAD_FILE) INTO DAD_BUF FROM SYSIBM.SYSDUMMY1 ; 1
   CALL DB2XML.DXXGENXML(DAD_BUF, RESULT_TABLE_NAME, RESULT_COLUMN_NAME,
   VALID_COLUMN_NAME, OVERRIDE_TYPE , OVERRIDE , MAX_NUM_DOCS, NUM_DOCS, ERRCODE ,
   ERRMSG ) ; 2
   IF ERRCODE < 0 THEN
   SET ERRMSG = 'ErrCode=' || TRIM ( CHAR ( ERRCODE ) ) || ' - ' || ERRMSG ;
   SIGNAL SQLSTATE '70001' SET MESSAGE_TEXT = ERRMSG ;
   END IF ;
END P1  ;
```

The parameters are described as follows:

- **DAD_FILE**: The name of the fully qualified DAD file name; it resides in integrated file system
- **RESULT_TABLE_NAME**: The name of the result table where the generated XML docs are stored
- **RESULT_COLUMN_NAME**: The name of the column in the result table in which the composed XML documents are stored
- **VALID_COLUMN_NAME**: The name of the column that indicates whether the XML document is valid

- **OVERRIDE_TYPE**: A flag to indicate the type of the following override parameter; set to one of the following values:
  - 0: No override
  - 1: Override by an SQL_stmt
  - 2: Override by an XPath-based condition
- **OVERRIDE**: Overrides the condition in the DAD file; the input value is based on the OVERRIDE_TYPE
- **MAX_NUM_DOCS**: The maximum number of rows in the result table
- **NUM_DOCS**: The actual number of generated rows in the result table
- **ERRCODE**: The return code from the stored procedure
- **ERMSG**: The message text that is returned in case of an error

At line 1 in Example 11-5, the DAD file is read into a CLOB variable. At line 2, the XML Extender stored procedure is called to perform the composition. The main purpose of the wrapper is to read the DAD from the integrated file system and pass its content to XML Extender as a CLOB.

11.6 Deploying DAD, SQL script to the integrated file system on the System i machine

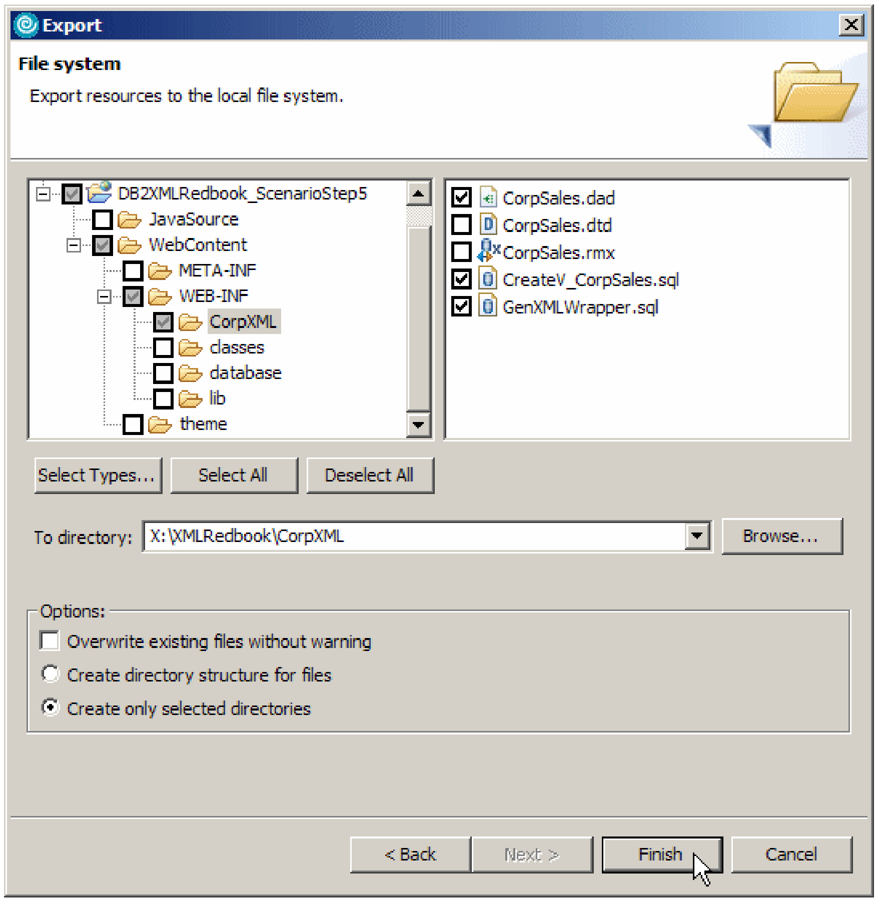
Let us review the application objects that we must deploy on the System i machine to run the XML compose solution:

- **Source database**: In our scenario, the data is retrieved from the CorpStage schema. The tables in the CorpStage schema already exist and are populated by the shredding application described in Chapter 10, “Shredding methodology” on page 155.
- **CorpSales.dad**: The DAD is used to map the relational database into the XML hierarchy of the outbound XML documents.
- **CreateV_CorpSales.sql**: This SQL DDL Script is used to create the V_CORPSALES view that aggregates the region sales data.
- **GenXMLWrapper.sql**: This SQL script is used to create the wrapper stored procedure.

The solution created in this chapter is deployed to the /XMLRedbook/CorpXML integrated file system directory on the target System i machine.

The following steps outline the deployment process:

1. We assume that the integrated file system Root directory on the target System i machine is mapped to a local X: drive on the development workstation. In WebSphere Development Studio Client, right-click the **CorpXML** folder and select **Export**.
2. In the window that opens, select **File System** and click **Next**.
3. In the Export window, select the three files as shown in Figure 11-9. Click **Browse** to navigate to the CorpXML folder in the integrated file system. Click **Finish**.



DB2XMLRedbook_ScenarioStep5 [✓]
JavaSource
WebContent [✓]
META-INF
WEB-INF [✓]
CorpXML [✓]
classes
database
lib
theme

[✓] CorpSales.dad
[ ] CorpSales.dtd
[ ] CorpSales.rmx
[✓] CreateV_CorpSales.sql
[✓] GenXMLWrapper.sql

To directory: X:\XMLRedbook\CorpXML

Options:
[ ] Overwrite existing files without warning
( ) Create directory structure for files
(●) Create only selected directories

Figure 11-9 Exporting the application files

4. In iSeries Navigator, start the Run SQL Scripts utility. Navigate to the Create_VCorpSales.sql script in the X:\XMLRedbook\CorpXML directory. Run the script.
5. Still in the Run SQL Scripts utility, open the **GenXMLWrapper.sql** script and execute it.

The application is now ready to compose outbound XML documents.

11.7 Running the deployed application

When the application is deployed, you can use any SQL interface to call the stored procedure to compose any number of outbound XML documents. We use the iSeries Navigator Run SQL Scripts utility that can display the output parameters. This allows us to monitor the outcome of the composition process. For example, we run the SQL script shown in Example 11-6 to compose multiple documents for all the data currently contained in the CorpStage schema (two sales dates for three different countries).

Example 11-6 Composing XML from the CorpStage schema

```
set path corpstage; 1

DECLARE GLOBAL TEMPORARY TABLE GENRESULTTABLE ( 2
   XMLDOCS DB2XML.XMLCLOB DEFAULT NULL ,
   VALID INTEGER DEFAULT NULL )
   WITH REPLACE
   ;

call genxmlwrapper('/XMLRedbook/CorpXML/CorpSales.dad',3
'SESSION.GENRESULTTABLE',
'XMLDOCS',
'VALID',
0, 4
'',
1024, 0, 0, '');

SELECT DB2XML.CONTENT(xmldocs , 5
'/XMLRedbook/CorpXML/CorpSales' ||
trim(db2xml.extractVarchar(xmldocs, '/CorpSales/CountryInfo/Name')) ||
trim(db2xml.extractVarchar(xmldocs, '/CorpSales/@Date')) ||
'.xml' ,
'UTF-8' )
FROM Session.genresulttable ;

commit;
```

In Example 11-6, at line 1, we set the path so that we can call the stored procedure with the unqualified name. In line 2, we declare a temporary table that will hold the composed documents. Note that this table must contain a column of type DB2XML.XMLClob or DB2XML.XMLVarChar.

In line 3, we call the wrapper stored procedure with the appropriate parameters. In line 4, we set the OVERRIDE_TYPE parameter to 0, which means there is no override. Accordingly, the next parameter OVERRIDE is an empty string.

At line 5, we use the XML Extender function, called Content, to write the generated XML documents into the integrated file system. We use the extractVarChar function to retrieve the country name and the sales dates information from the generated document. We then use this information to dynamically create the outbound file names. DB2 automatically disposes of the temporary table when the Run SQL Scripts session is ended.

The stored procedure returns the following messages:

```
Return Code = 0
Output Parameter #8 = 6
Output Parameter #9 = 0
Output Parameter #10 = DXXQ020I  XML successfully generated.
```

Notice that the NUM_DOCS output parameter returned 6, which means that a single call to the stored procedure generated six XML documents, one document per each date and country.

Sometimes, however, we want to compose a single XML document. Suppose that we want to retrieve sales data for Poland for 6 April 2006. With the flexibility built into our DAD, we can easily accomplish this task by adding the appropriate override on the stored procedure call as shown in Example 11-7.

Example 11-7 Using Override

```
call genxmlwrapper('/XMLRedbook/CorpXML/CorpSales.dad',
'Session.GENRESULTTABLE',
'XMLDOCS',
NULL,
2, 1
'/CorpSales/CountryInfo/Name=''Poland'' AND /CorpSales/@Date=''2006-04-06''', 2
 1024, 0, 0, '');
```

In line 1 of Example 11-7, we set OVERRIDE_TYPE to 2, which means that we request an override by an XPath-based condition. In line 2, we provide the override condition. Refer to the following DB2 Information Center article for more details about the syntax of override expressions:

http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/t0008569.htm

Tip: Make sure that the literals in the override expression are contained in *single* quotation marks.

12

Advantages and disadvantages of the middleware approach

In Part 3, “Middleware approach” on page 107, we describe the second approach to integrating XML into DB2 for i5/OS, which is the middleware approach. In this methodology, we use middleware products that help to store and retrieve XML data into DB2 for i5/OS. One of the products that we use is the DB2 UDB XML Extender for iSeries. The advantage of the middleware approach is that the development can be reused and it is flexible.

In this chapter, we discuss the advantages and disadvantages of the middleware approach.

12.1 Advantages

The middleware approach to integrating XML and DB2 has several advantages, including:

- It requires no knowledge of XML parsing because it is handled by the middleware and the developer can concentrate more on the business requirements.
- It does not require high-level language (HLL), such as CLI, RPG, C, COBOL, or Java, database coding techniques.
- It can use any SQL interface for interaction with the middleware.
- There are toolsets for design such as WebSphere Development Studio Client and iSeries Navigator.
- It is template-driven, which offers a quick setup using the provided engine.
- If offers cross-platform implementation of the same middleware, so you can do a one-time setup and run it from anywhere.
- There is less maintenance since there is only one initial setup.

12.2 Disadvantages

The middleware approach to integrating XML and DB2 has the following disadvantages:

- You are required to buy a product to use the middleware.
- Knowledge of the IBM document access definition (DAD) standard is required.
- This approach involves longer setup times.
- Some limitations, such as size of the documents or functionality, are introduced.

Part 4

Moving forward

In this part, we discuss some of the futures technologies related to XML. This part includes Chapter 13, “A look into the future” on page 201.

13

A look into the future

In this chapter, we look into the future of XML in general and some of the cutting edge XML technologies, including:

- XQuery
- SQL/XML
- Hybrid database management systems
- XLink
- XML Encryption

13.1 Future XML technologies

While much can be done on the System i platform with XML already, there are some capabilities that the System i platform does not yet have. The future of XML is wide open. In this section, we briefly cover a few items where we think the future of XML might go. Some of these are already accepted standards, while others are still in the proposal stages and not completely accepted.

13.1.1 XQuery

XQuery is a query language that is built on top of XPath that allows for information to be queried quickly from an XML document. XQuery does not simply scan the XML document. It uses an underlying data model of the document called XQuery Data Model (XDM). XQuery is similar to SQL for relational databases. We need to provide a source (from clause), a list of data items to select (fields), and restrictions (where clause).

Let us consider the simple XML document shown in Example 13-1.

Example 13-1 Books.xml sample XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<Books>
   <Book>
      <Title>My First Book</Title>
      <Price>20.00</Price>
   </Book>
   <Book>
      <Title>My Second Book</Title>
      <Price>30.00</Price>
   </Book>
</Books>
```

First we use the **doc** function to determine from which XML document we are selecting. We can select the titles of all the books in the document by combining the **doc** function with the XPath statement such as:

```
doc(“Books.xml”)/Books/Book/Title
```

This returns:

- `<Title>My First Book</Title>`
- `<Title>My Second Book</Title>`

We were hoping for the titles of the books, but what went wrong? If we look back to 1.1.4, “XPath and Location Path” on page 9, and use the XPathTester java program in Example 1-6 on page 11, we can see that we forgot to add the text function to get just the text. We can change our statement to:

```
doc(“Books.xml”)/Books/Book/Title/text()
```

This then results in:

- My First Book
- My Second Book

We can also add in selection criteria. For example, if we want books that are over US$25, then we can use:

```
doc("Books.xml")/Books/Book[Price>25]/Title/text()
```

This results in “My Second Book”.

While XQuery commands can be simple, one-line statements, they can also be expanded to a format known as FLWOR (For, Let, Where, Order by, and Return). This allows a statement to have selection, ordering, and formatting of the output results. Using the same criteria as before, we can rewrite the XQuery as:

```
for $x in doc("Books.xml")/Books/Book
where $x/Price>25
return $x/Title/text()
```

This yields the same result. However, for more complex statements, it is easier to expand them into the FLWOR format.

For more information about this standard, see:

http://www.w3.org

13.1.2 SQL/XML

SQL/XML is an emerging American National Standards Institute (ANSI) and International Organization for Standardization (ISO) standard that allows for access to XML from an SQL interface. SQL/XML provides methods for storing XML documents with relational databases. In addition to storing, the relational data can then be generated into an XML document. If this sounds familiar, it is because much of this functionality is provided by the XML Extender product. However, SQL/XML provides a standard interface to do these items that crosses multiple platforms.

In addition to providing methods for storing XML documents in relational databases and composing XML documents from relational data, there are definitions that allow for the integration of XPath and XQuery. We cover XQuery in 13.1.1, “XQuery” on page 202, and XPath in 1.1.4, “XPath and Location Path” on page 9.

For more information about the proposed standards, see:

- ISO

  http://www.iso.org

- ANSI

  http://www.ansi.org

13.1.3 Hybrid and multistructured database management systems

Chapter 1, “Introduction to XML integration with DB2 for i5/OS” on page 3, we describe the two types of data models in consideration here. The first of these is the relational data model that is used by DB2. The second is the hierarchical data model that is used by XML documents. We took the position that in today’s technology world, the choice becomes which one to use, a relational database management system or a hierarchical database management system. What if we could combine these to models together?

A new technology is that of a hybrid or multistructured database management system (DBMS). The thought is to have one common DBMS that has access to data stored in both relational and hierarchical structures. Queries into the DBMS can be in the either SQL, SQL/XML, or XQuery. The DBMS then accesses the appropriate data and returns the results. Figure 13-1 shows a high-level view of this type of architecture.

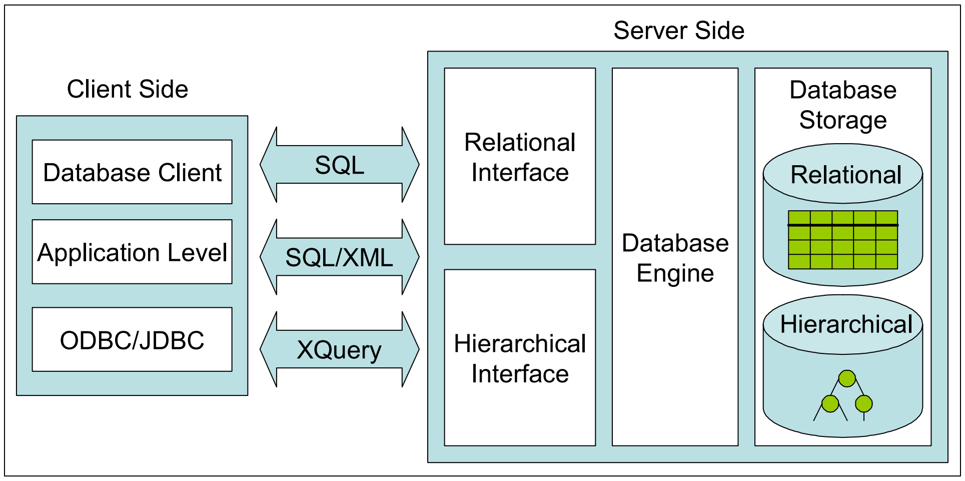


Figure 13-1 Hybrid DBMS overview

13.1.4 XLink

We have mentioned several times that XML is plain text data. What if you want to embed an image file in the XML document? Can you? The answer is yes and no. Using a technology called Scalable Vector Graphics (SVG), you can represent an image in terms of describing shapes, sizes, and positions in text. Therefore the text that describes the image can be embedded in an XML document and then drawn by an SVG viewer program. However, JPEGs, GIFs, MP3s, and any other type of binary file are not allowed.

The solution to this problem is XLink. XLink is a recommendation for linking objects outside of an XML document to the XML document. This is the equivalent of a hypertext reference in HTML. XLink describes the formatting of elements to link to other items.

For more information about this recommendation, see:

http://www.w3.org

13.1.5 XML Encryption

While one strength is that an XML document is plain text, there needs to be consideration for security. Since the document is plain text, security needs to happen at a layer above the document itself.

The goal of the XML Encryption workgroup is to define a standard for how an XML document can be encrypted in its entirety or that certain elements of the document are encrypted. Imagine a transaction file that contains credit card numbers. While we may not need the entire document encrypted, we prefer not to have the charge card numbers in plain text. This technology is still in the workgroup phase and has no proposed standards yet.

For more information about this workgroup, see:

http://www.w3.org

13.2 Future XML technologies on the System i platform

As new standards are adopted across the industry, the development team at IBM considers adopting these for the System i platform. Some of these technologies may be added while others are not. As the state of the art adjusts, so will the System i platform.

A

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG247258`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG247258.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
XMLRedbook.zip	Zipped code samples

The XMLRedbook.zip file contains the files listed in Table A-1. This same table also describes the content of the download packages.

Table A-1   Contents of the download packages

Location	Sample name	Description	References
/XMLRedbook/ CountryXML	CountryByRegionABCYYYYMMDD.xml, where ABC is the county and YYYYMMDD is the date	CountrySalesByRegion XML documents generated from SQL mapping document access definition (DAD)	Section 8.4.4, "Executing the SQL mapping DAD composition" on page 137
/XMLRedbook/ CountryXML	CountrySalesByRegion.dad	SQL mapping DAD to compose the CountrySalesByRegion XML documents	Section 8.4, "SQL mapping DAD composition example" on page 129
/XMLRedbook/ CountryXML	CountrySalesByRegion.xsl	XSL stylesheet that shows how to insert unique ID attributes to an inbound XML document	Section 10.3, "XSL Transform to insert unique keys for repeating elements" on page 161
/XMLRedbook/ CountryXML	StoreSales.dad	XML Column DAD to archive the StoreSales XML documents intact	"XML column example" on page 117
/XMLRedbook/iSeriesLibraries	xmlredbook.savf	An i5/OS save file **Note**: You must send this file by FTP to a System i machine, and then restore the library XMLRedbook	Chapter 5, "Using RPG for XML processing" on page 59
/XMLRedbook/ SQL Source	Setup.sql	SQL script to set up databases for the example given	Chapter 2, "Scenario overview" on page 21
/XMLRedbook/ SQL Source	Cleanup.sq	SQL script to clean up databases for the example given	Chapter 2, "Scenario overview" on page 21
/XMLRedbook/ SQL Source	Create GenStoreXML.sql	SQL script to create the GenStoreXML stored procedure	Section 3.3, "Coding the SQL stored procedure" on page 36
/XMLRedbook/ SQL Source	Run GenStoreXML.sql	SQL script to execute the GenStoreXML stored procedure	Section 3.5, "Running the SQL stored procedure" on page 40
/XMLRedbook/ SQL Source	Run GenCountryXML.sql	SQL script to execute the SQL mapping DAD composition with XML Extenders	Section 8.4.4, "Executing the SQL mapping DAD composition" on page 137
/XMLRedbook/ StoreXML	Store#DD.xml (where # is store number and DD is date)	StoreSales XML document that is composed by the SQL stored procedure	Section 3.5, "Running the SQL stored procedure" on page 40
/XMLRedbook/ StoreXML	Store#DD.xml.sql	SQL script that is generated to decompose the StoreSales XML document into the CountryXXX database	Section 4.1, "Decomposing XML using XSLT, Java, and SQL" on page 44
/XMLRedbook/ StoreXML	StoreSales.dtd	Document type definition to validate the StoreSales XML documents	Section 4.1, "Decomposing XML using XSLT, Java, and SQL" on page 44
/XMLRedbook/ StoreXML	StoreSales.xsd	XML Schema Definition to validate the StoreSales XML document	Section 1.1.3, "XML schema" on page 6

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	100 MB minimum
Operating System:	Windows XP Professional with the latest SP
Processor:	1.5 GHz
Memory:	1 GB

How to use the Web material

Create a subdirectory (folder) on your workstation, and copy the contents of the XMLRedbook subdirectory into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 211. Note that some of the documents referenced here may be available in softcopy only.

- *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503
- *XML for DB2 Information Integration*, SG24-6994

Other publications

These publications are also relevant as further information sources:

- *IBM DB2 Universal Database for i5/OS XML Extender Administration and Programming,* SC18-9179
- *WebSphere Development Studio ILE COBOL Programmer’s Guide*, SC09-2540
- *WebSphere Development Studio ILE RPG Language Reference*, SC09-2508
- *WebSphere Development Studio ILE RPG Programmer’s Guide,* SC09-2507

Online resources

These Web sites are also relevant as further information sources:

- iSeries Information Center

  `http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp`
- DB2 XML Extender product support

  `http://www.ibm.com/software/data/db2/extenders/xmlext/support.html`

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

A
advantages of programmatic approach 106
ancestor 10
ancestor-or-self 10
attribute 10

C
cascading stylesheet (CSS) 14
CGI (Common Gateway Interface) 60
child 10
child element 4
children 4
comment 10
commitment control 144
Common Gateway Interface (CGI) 60
Composition Stored Procedure 128
containment 18
CorpSales 26, 28
CorpStage 26
CountrySalesByRegion 27
CountryXXX 25
Create GenStoreXML.sql 39
CSS (cascading stylesheet) 14

D
DAD (document access definition) 110
data management in XML columns 122
database enablement 114
database model 24
DB2 UDB XML Extender for iSeries 19
DB2DXX_QUICKPOOL_MALLOC 149
DB2XML schema 114
DB2XML.XML_USAGE table 116
DEBUG parameter 54
Decomposition Stored Procedure 128
descendant 10
descendant-or-self 10
DFTRDBCOL 54
disadvantages of programmatic approach 106
document (root 10
document access definition (DAD) 110
document node 10
Document Object Model (DOM) 18
document type definition (DTD) 5
DTD (document type definition) 5
dvalidate 126
DVALIDATE UDF 126
dxxGenXML 128, 140
dxxGenXMLCLOB 128
dxxGenXMLClob 140
DXXInsertXML 140
dxxInsertXML 128
dxxInsertXML100MB 140
dxxRetrieveXML 128, 140
dxxRetrieveXMLCLOB 128
DXXSAMPLES 112
DXXShredXML 140
dxxShredXML 128
dxxShredXML100MB 140

E
element 10
entity 38
Extensible Markup Language (XML) 3
Extensible Stylesheet Language (XSL) 14, 43
Extensible Stylesheet Language Transformation (XSLT) 14, 126
external stored procedure 34
extractChar 125
extractChars 125
extractCLOB 125
extractDate 125
extractDouble 125
extractInteger 125
extractReal 125
extractSmallint 125
extractTime 125
extractTimestamp 125
extractVarchar 125

F
facet 7
FLWOR 203
following 10
following-sibling 10
foreign key 16

G
Guaranteed Universal Unique Identifier (GUUI) 53
GUUI (Guaranteed Universal Unique Identifier) 53

H
hierarchical data model 17
hierarchical structure 4
hybrid database management systems 203

I
ILE RPG compiler 60
Inserting the XML documents 120

J
Java Transform Class 81
journal management 144
journaling 144

L
Location Path 9, 13

M
Migration 143
multi-structured database management systems 203

N
namespace 10
node test 10

O
optional elements 18

P
Panel Group (PNLGRP) 44
parent 4, 10
performance 147
PNLGRP (Panel Group) 44
pool memory management 148
preceding 10
preceding-sibling 10
predicates 10
processing-instruction 10
Program Status Data Structure (PSDS) 66
programmatic approach
  advantages 106
  disadvantages 106
PTF quick reference 149

Q
QCMDEXC API 83
QRTVJOBI prototype 66
QZXMSAMP1 save file 112
QZXMSAMP2 save file 112

R
RDB node mapping 128
Redbooks Web site 211
  Contact us xii
referential integrity (RI) 17
relational data model 16
relational database (RDB) node mapping 128
relational database to XML mapping 15
repeating elements 18
restore 144
retrieval of entire XML documents 122
RI (referential integrity) 17
root element 4
RSTOBJ command 72
RTVJOB API 66
Run GenStoreXML.sql 40
Run SQL Scripts utility 39
RUNJVA command 52
RUNSQLSTM command 52

S
same named elements 18
save 144
SAVOBJ command 72
SAX (Simple API for XML) 18
Scalable Vector Graphics (SVG) 204
self 10
siblings 4
side table 116, 120
Simple API for XML (SAX) 18
SQL
  mapping scheme DAD 128
  naming convention 143
  stored procedure 34
  view 131
SQL Communications Area (SQLCA) 70
SQL Persistent Stored Module (SQL PSM) 34
SQL PSM (SQL Persistent Stored Module) 34
SQL/XML 203
SQLCA (SQL Communications Area) 70
SQLFILE 53
stored procedure 34
  deployment 41
  external 34
StoreSales 27
StoreXXX 24
strategies
  middleware 19
  programmatic 18
  XML Extender 19
svalidate 126
SVALIDATE UDF 126
SVG (Scalable Vector Graphics) 204

T
text 10
Toolbox JDBC driver 99

U
UDF (user-defined function) 110
UDT (user-defined type) 110
Update UDF 126
user-defined function (UDF) 110
user-defined type (UDT) 110

W
W3C XSLT standard 9
wrapper element 18, 184
wrapper stored procedure 176
WRKRDBDIRE 113

X
XDM (XQuery Data Model) 202
XLink 204
XML 3
  composition via an SQL stored procedure 36
XML (Extensible Markup Language) 3

XML Collection method 127
XML column
  data management 122
  enablement 120
XML document
  models 27
  retrieval 122
  storage 122
  transformation 126
  update of 125
  validation 126
XML Encryption 204
XML Extender 110
  DAD 115–116, 118, 127
  database enablement 114
  default schema 112
  DTD repository table 114
  DTD_REF 114, 118, 143
  error messages 145
  installation 111
  migration 143
  performance 148
  prerequisites 111
  sample programs 112
  storage and access methods 114
  trace facility 145
  UDF 122
  very large documents 139
  XML Collection method 115
  XML Column method 115–116
  XML_USAGE 114, 116, 128, 143
XML Path Language (XPath) 14
XML schema 6
XML Schema Definition (XSD) 8
XML to RDB mapping 17, 115
XML to relational database mapping 15
XMLCLOB 114
XMLCLOBFromFile 122
XMLFILE 53
XMLFileFromCLOB 122
XMLFileFromVarchar 122
XML-FO (XSL Formatting Object) 14
XMLVARCHAR 118
XMLVarcharFromFile 122
XPath (XML Path Language) 14
XPath axis 10
XQuery 202
XQuery Data Model (XDM) 202
xs
  boolean 6
  date 6
  decimal 6
  integer 6
  string 6
  time 6
XSD (XML Schema Definition) 8
XSL (Extensible Stylesheet Language) 14, 43
XSL Formatting Object (XML-FO) 14
XSL transform document 48
XSLFILE 53
XSLT (Extensible Stylesheet Language Transformation) 14, 126
XSLTransformToClob 127
xxRetrieveXMLClob 140

IBM Redbooks

The Ins and Outs of XML and DB2 for i5/OS

(0.2”spine)
0.17”<->0.473”
90<->249 pages

The Ins and Outs of XML and DB2 for i5/OS

Understand how XML and DB2 work together

Master XML and relational database mapping

Leverage current programming skills to process XML

XML represents a fundamental change in computing. It allows applications to move away from proprietary file and data formats to a world of open data interchange. XML has become ubiquitous not only because of its range of applications, but also because of its ease of use.

Although XML solves many problems by providing a standard format for data interchange, some challenges remain. In the real world, applications need reliable services to store, retrieve, and manipulate data. These services have traditionally been offered by DB2 for i5/OS.

In this IBM Redbook, we discuss the challenges of representing XML hierarchies in the relational database model. We provide an in-depth explanation of the three most popular approaches to bridge the hierarchy, the relational model dichotomy:

- Programmatically process the XML documents and map their hierarchy into a relational database.
- Use database middleware to handle the XML parsing and XML-to-relational database mapping.
- Use Extensible Stylesheet Language (XSL) Transformation to transform inbound XML documents directly to SQL scripts.

We also share best practices and techniques aimed at streamlining the XML and DB2 for i5/OS integration.

SG24-7258-00 ISBN 0738496901